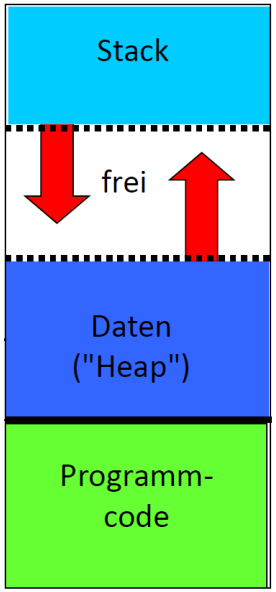
Kapitel 4
Hauptspeicher-Verwaltung

1. Grundlagen
   a) Anforderungen
   b) Adressberechnung
   c) Einfache Zuweisungs- und Freigabeverfahren
   d) Implementierungsaspekte
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Anforderungen an die Hauptspeicherverwaltung

- Automatische Zuweisung und Freigabe von Hauptspeicher für Prozesse
- Verschiebbarkeit (Relokation) von Programmcode im Hauptspeicher *(→ Adressberechnung)*
- Kapselung jedes Prozesses *(Schutz und Zugriffskontrolle → getrennte Adressräume)*
- Automatische Auslagerung von Prozessen *(Scheduling!)*



Kapitel 4
Hauptspeicher-Verwaltung

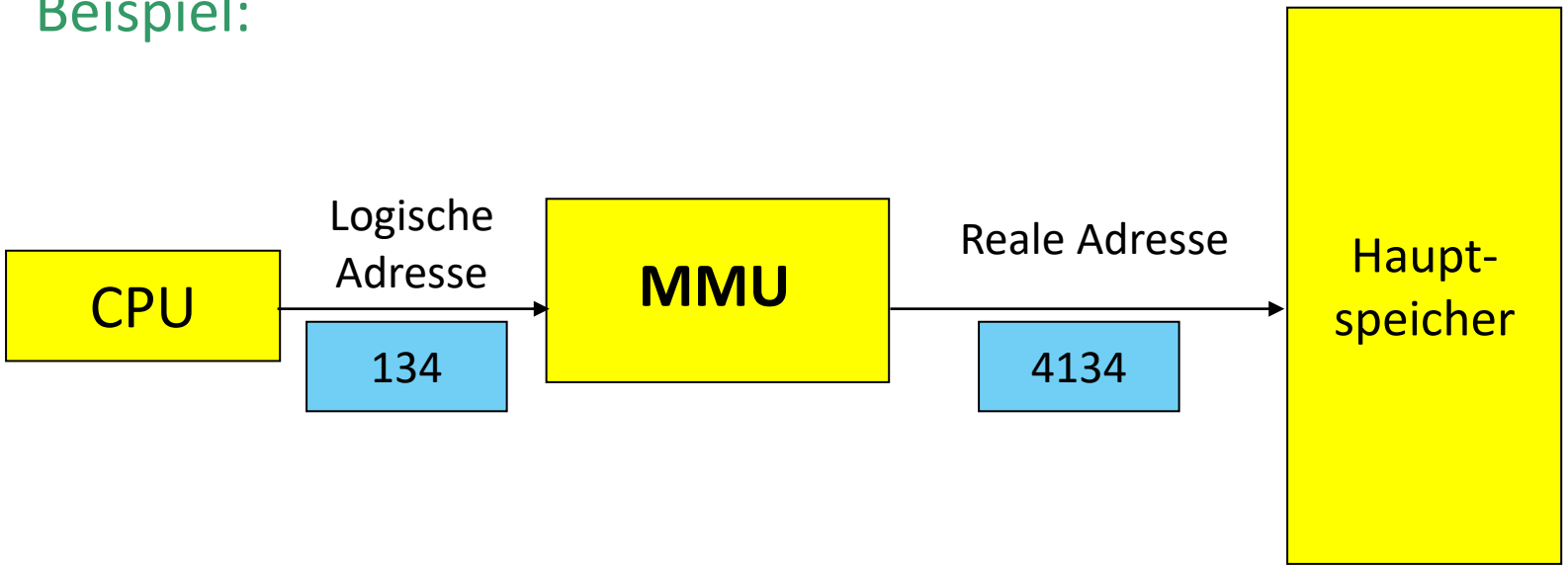
1. Grundlagen
   a) Anforderungen
   b) Adressberechnung
   c) Einfache Zuweisungs- und Freigabeverfahren
   d) Implementierungsaspekte
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Adressen

- Logische Adresse (virtuelle Adresse)
  - Referenz auf einen Speicherplatz, ohne dass dessen reale (absolute) Hauptspeicheradresse bekannt ist
  - Eine „Übersetzung“ muss vom System (Betriebssystem oder Hardware) vorgenommen werden
- Reale Adresse
  - Die absolute Adresse eines Speicherplatzes (Bytes) im physikalischen Hauptspeicher

Dynamische Adressumsetzung zur Laufzeit

- Abbildung einer logischen Adresse auf die reale Hauptspeicher-Adresse zum Zeitpunkt der Programmausführung:
  Aufgabe der **MMU** (Memory Management Unit)
- Beispiel:



Die CPU kennt nur die logischen Adressen!

Einfache Adressumsetzungsmethode *(historisch!)*

- Basis-Register *(„Relocation“-Register)*
  - Enthält die Startadresse des Prozess-Speicherbereichs
  - Berechnungsverfahren:
    Logische Adresse + Wert des Basis-Registers = Reale Adresse
- Limit-Register
  - Endadresse des Prozess-Speicherbereichs
  - **Wenn Reale Adresse > Limit-Register ➔ Fehler!**
    (Schutzverletzung)

Diese Register werden …

- gesetzt, wenn der Prozess geladen oder verschoben wird
- von der MMU zur Adressberechnung zur Laufzeit verwendet

Compilieren, Linken und Laden eines Programms in den Hauptspeicher

- **Compiler** erzeugen üblicherweise aus den Namen in den einzelnen Programm-Dateien des Quellcodes ("Modulen") logische Adressen. Bezüge auf andere Dateien (über Header-Dateien deklariert) werden vorerst offen gelassen.
- **Linker** fügen die einzelnen Dateien zusammen, indem sie
  - die einzelnen Teil-Adressräume gegeneinander verschieben, so dass sie sich nicht überdecken,
  - Querbezüge zwischen den Dateien auflösen.
  - Es entsteht ein einheitlicher logischer Adressraum
- Die Adressen dieses logischen Adressraumes werden während der Ausführung des Programmcodes durch die CPU (von der **MMU**) auf die realen Adressen abgebildet.

Beispiel: Compilieren, Linken und Laden eines C-Programms mit zwei Dateien

	Datei1	Datei2
Quellcode:	printf(..)	i = 10;
Compiler:	CALL 34	MOVE [34], R0
Linker:	CALL 34	MOVE [134], R0
Loader:	Basisregister = 4000	
CPU:	CALL 34 (→ 4034)	MOVE [134], R0 (→[4134], R0)

Anmerkungen:

- **Compiler**: Jede Datei definiert ein Modul und wird einzeln übersetzt (Adressen beginnen immer bei 0).
- **Linker**: 100 Byte reichen hier dem Linker zum Verschieben der Adressräume, da im Beispiel angenommen wird, dass die max. genutzte Adresse von Datei1 99 ist.
- **Loader**: Setzt den Basisregister-Wert auf 4000
  (Methode der MMU ist hier die Adressumsetzung mit Basisregister)

Kapitel 4
Hauptspeicher-Verwaltung

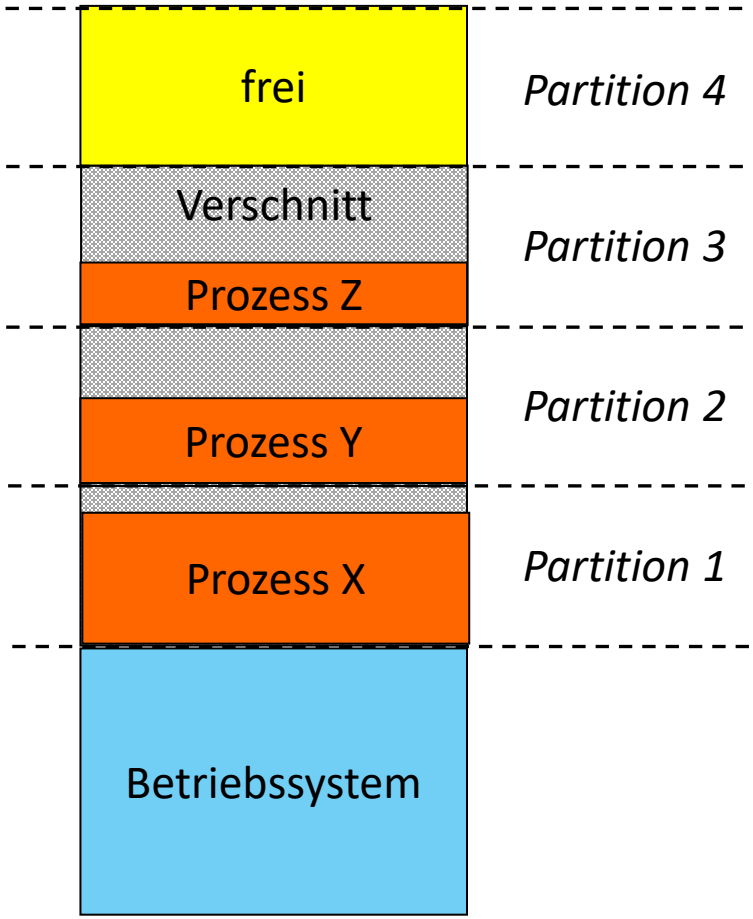
1. Grundlagen
   a) Anforderungen
   b) Adressberechnung
   c) Einfache Zuweisungs- und Freigabeverfahren
   d) Implementierungsaspekte
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Hauptspeicheraufteilung bei Multiprogramming: Feste Partitionierung

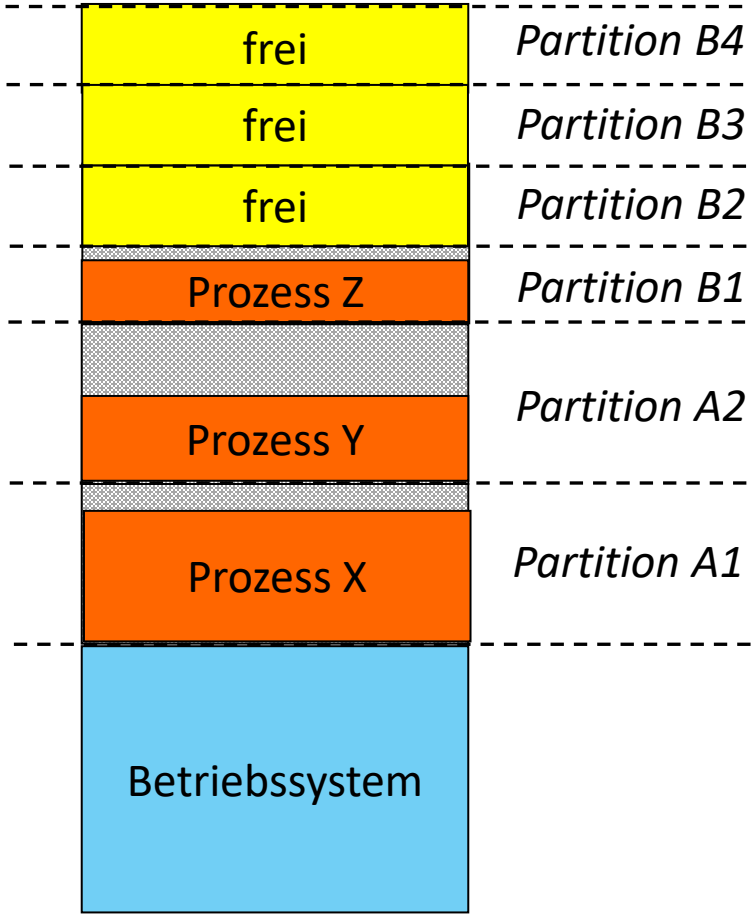
- Aufteilung in feste Anzahl Partitionen (Teile)
  - Jeder Prozess belegt genau eine Partition
  - Jeder Prozess, dessen Platzbedarf $\leq$ der Größe einer freien Partition ist, kann geladen werden
  - Wenn alle Partitionen voll sind, kann das Betriebssystem einzelne Prozesse leicht aus-/ einlagern
  - Varianten bzgl. der Partitionsgröße:
    - Alle Partitionen haben eine einheitliche Größe
    - Es gibt unterschiedliche Partitionsgrößen
      → Verringerung des nicht-nutzbaren Hauptspeichers („Verschnitt“)

Beispiele: Feste Partitionierung

Feste Hauptspeicher-Partitionen **einheitlicher** Größe:



Feste Hauptspeicher-Partitionen **unterschiedlicher** Größe:

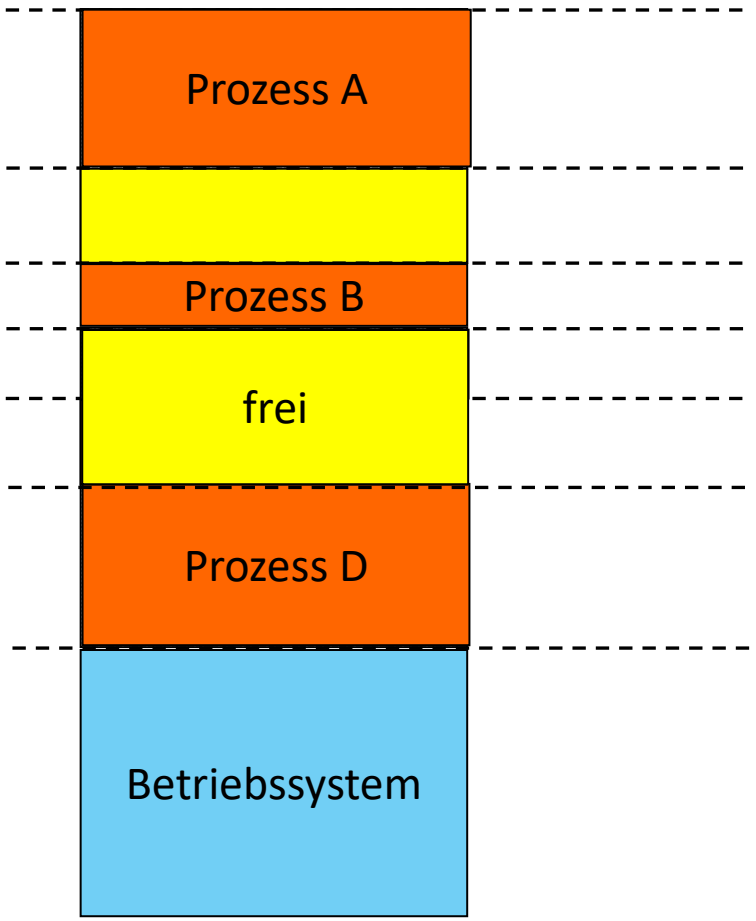


Feste Partitionierung: Probleme

- Ein Programm kann zu groß sein für die Partition
  - der Programmierer muss dann sein Programm aufteilen: **„Overlay“-Technik**
- Der Hauptspeicher wird nicht effizient genutzt
  - jedes Programm belegt eine komplette Partition → „Verschnitt“ → ungenutzer freier Speicher („Interne Fragmentierung“)

Hauptspeicheraufteilung bei Multiprogramming:
Dynamische Partitionierung

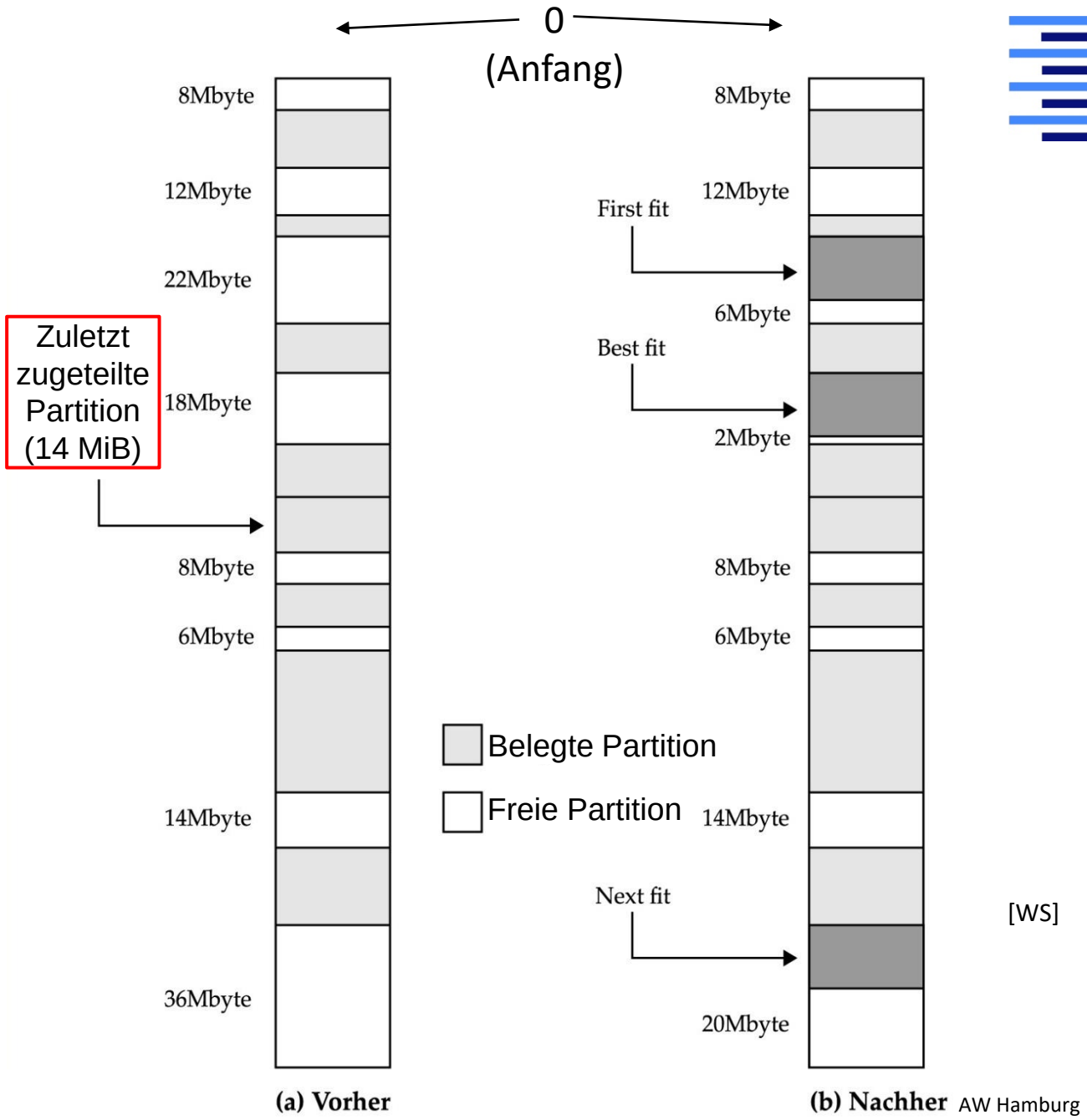
- Es gibt eine variable Anzahl von Partitionen
- Partitionen haben unterschiedliche Größen
- Die Partitionen werden an die Prozessgröße angepasst
  - Nach Zuweisung einer Partition zu einem Prozess wird der restliche freie Platz eine neue Partition
  - Zusammenfassen von freien Partitionen ist möglich



Dynamische Partitionierung: Platzierungsstrategien

- Das Betriebssystem muss entscheiden, welche freie Partition welchem Prozess zugewiesen wird
- Algorithmen:
  - First-Fit
    - Sucht von vorne die nächste freie Partition, die passt
  - Next-Fit
    - Sucht ab der zuletzt belegten Partition die nächste freie Partition, die passt
  - Best-Fit
    - Auswahl der freien Partition, bei der am wenigsten Platz verschwendet wird

Beispiel für Platzierungs-strategien: Platzierung eines 16 MiB großen Prozesses



[WS]

Dynamische Partitionierung: Bewertung der Platzierungsalgorithmen

- Best-Fit:
  - Schlechtestes Ergebnis!
  - Aufwendigste Suche
  - Weil immer kleine Speicherreste bleiben, muss das Betriebssystem am häufigsten umsortieren
- First Fit:
  - Schnellstes Verfahren!
  - Viele Prozesse im vorderen Speicherbereich
  - Meist hinten noch Platz für große Prozesse
- Next-Fit:
  - Belegt Speicher gleichmäßiger als First Fit, nachteilig für große Prozesse
  - Die größte freie Partition wird eher verwendet (liegt meist hinten)
  - Umsortieren, um wieder Platz für große Prozesse zu erhalten, ist oft nötig
  - Leichter Nachteil gegenüber First Fit!

Hauptspeicheraufteilung bei Multiprogramming: Dynamische Partitionierung

Probleme

- Der Hauptspeicher wird immer noch nicht effizient genutzt
  - es entstehen „Löcher“ im Speicher durch kleine Partitionen („externe Fragmentierung“)
    - Abhilfe: Das Betriebssystem könnte die Partitionen regelmäßig umkopieren (ist aber sehr aufwändig)
- Ein Programm kann zu groß sein für den gesamten verfügbaren Hauptspeicher → **„Overlay“-Technik** nötig

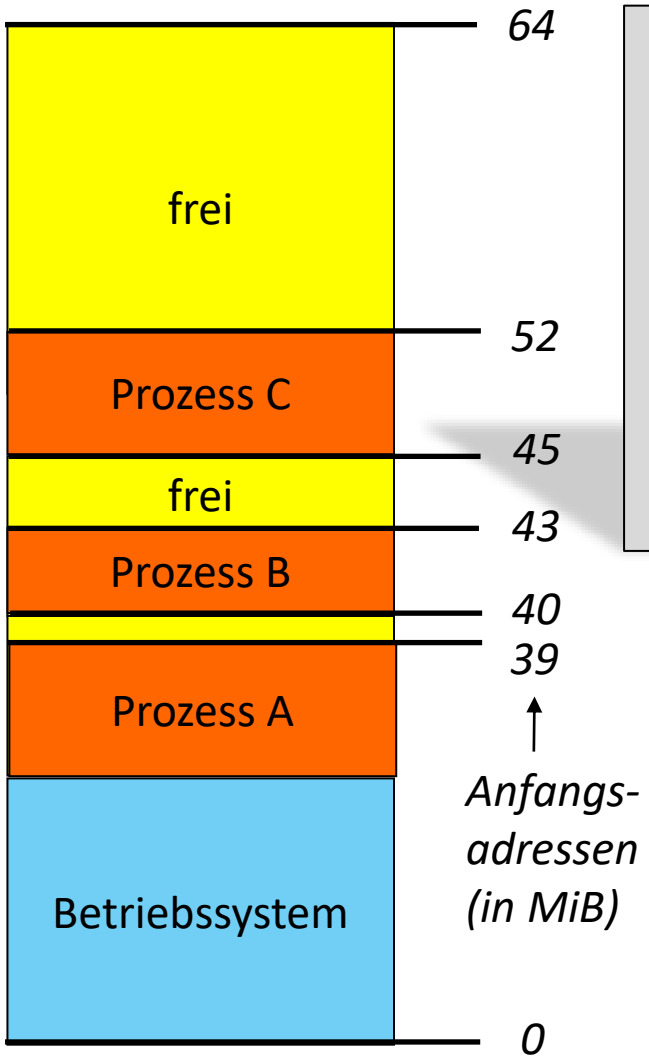
Kapitel 4
Hauptspeicher-Verwaltung

1. Grundlagen
   a) Anforderungen
   b) Adressberechnung
   c) Einfache Zuweisungs- und Freigabeverfahren
   d) Implementierungsaspekte
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

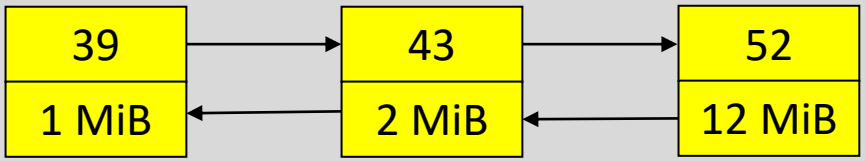
Implementierung der Speicherverwaltung

- **Speicherverwaltung mit Bitmaps**
  - Jeder Speichereinheit (z.B. Partition) entspricht ein Bit in einem Speicherwort („Bitmap“). Dessen Wert signalisiert, ob die Einheit frei ist.
  - Problem: Die Suche nach mehreren zusammenhängenden Einheiten ist relativ aufwändig (Wortgrenzen!)
- **Speicherverwaltung mit verketteten Listen („Freibereichslisten“)**
  - In einer doppelt verketteten Liste spezifiziert jedes Element einen freien Bereich im Hauptspeicher durch Angabe von realer Anfangsadresse und Länge

Beispiel für eine Freibereichsliste



Zugehörige Freibereichsliste (Anfangsadresse / Länge):



Auslagerung von Prozessen

- Falls nicht **alle** Prozesse im Hauptspeicher gehalten werden können: Auslagerung von Prozessen auf Platte (ggf. temporär)!
- Ansätze:
  - Ein- und Auslagern **kompletter** Prozesse: **Swapping**
  - Dynamische Ein- und Auslagerung von Programm**teilen**:
    - Laden von Bibliotheksprozeduren auf Anforderung: **„dynamic loading“**
    - Ein- und Auslagerung einzelner vom Programmierer manuell aufgeteilter Programmstücke: **„Overlays“**
    - Automatische Aufteilung von Prozessen und dynamisches Ein- und Auslagern einzelner Prozessteile auf Anforderung: **„Virtueller Speicher“**

Kapitel 4
Hauptspeicher-Verwaltung

1. Anforderungen und Grundlagen
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Anforderungen an einen idealen Speicher aus Programmierersicht

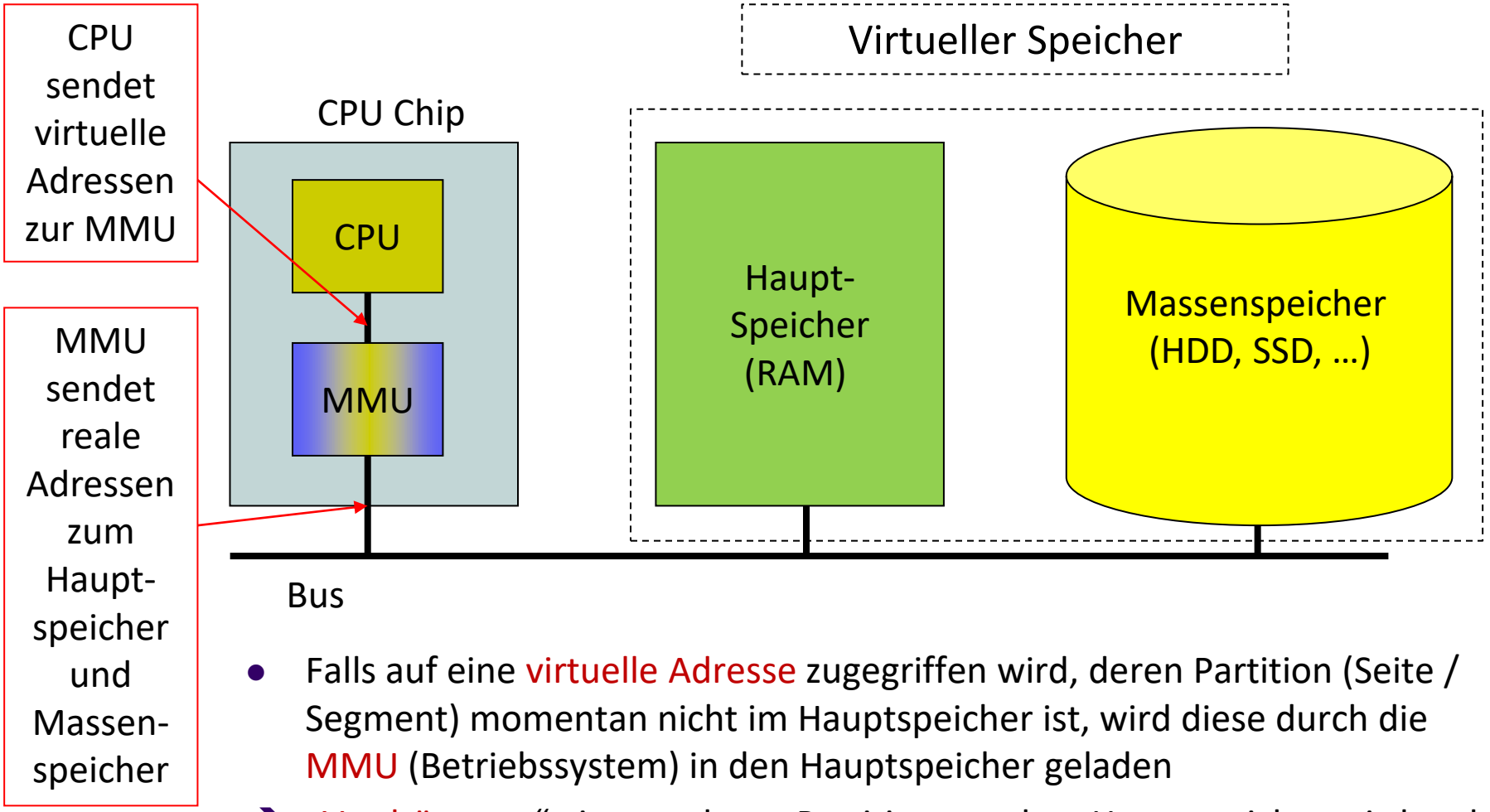
- **Unbeschränkte Größe**, so dass jedes beliebig große Programm ohne zusätzlichen Aufwand geladen und verarbeitet werden kann
- **Einheitliches Adressierungsschema** für **alle** Speicherzugriffe (keine Unterscheidung von Speichermedien)
- **Direkter Zugriff** auf den Speicher (ohne Zwischentransporte)
- **Schutz vor fremden Zugriffen** in den eigenen Speicherbereich

➔ **Ziel: „Virtueller Speicher“ mit idealen Eigenschaften!**

Realisierung eines virtuellen Speichers

- Aufteilung des Hauptspeichers in viele (kleine) Partitionen
  - feste Größe: **Seiten** (→ Paging)
  - unterschiedliche Größe: **Segmente**
- **Jeder Prozess wird auf mehrere Partitionen verteilt!**
- Partitionen können einzeln auf die Platte ausgelagert werden
- Jeder Prozess benutzt eigene logische („virtuelle“) Adressen
- Jeder virtuelle Adressraum kann größer als der physikalische Hauptspeicher sein (nur durch Plattenkapazität beschränkt)

Abbildung der virtuellen Adressen auf reale Adressen durch die MMU



- Falls auf eine virtuelle Adresse zugegriffen wird, deren Partition (Seite / Segment) momentan nicht im Hauptspeicher ist, wird diese durch die MMU (Betriebssystem) in den Hauptspeicher geladen

➔ „Verdrängung“ einer anderen Partition aus dem Hauptspeicher wird evtl. notwendig

Kapitel 4
Hauptspeicher-Verwaltung

1. Anforderungen und Grundlagen
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Grundprinzip des Paging

- Linearer Adressraum mit virtuellen Adressen
  - Virtuelle Adressen …
    - … beginnen für *jeden* Prozess bei 0
    - … werden fortlaufend durchnummeriert
    - … täuschen einen virtuellen Speicher vor
- Aufteilung des virtuellen Adressraums in Seiten
  - Seite = Partition fester Größe *(„Page“)*
  - Jeder Seite wird ein zusammenhängender realer Speicherbereich zugeordnet, auch *Seitenrahmen („Page Frame“)* genannt
  - Eine Seite kann im Hauptspeicher oder auf der Platte liegen, muss bei Zugriff aber in den Hauptspeicher geladen werden
- Abbildung von virtuellen auf reale Adressen (virtuelle Seiten → Seitenrahmen) durch eine Seitentabelle

Abbildung virtuelle Adresse → reale Adresse

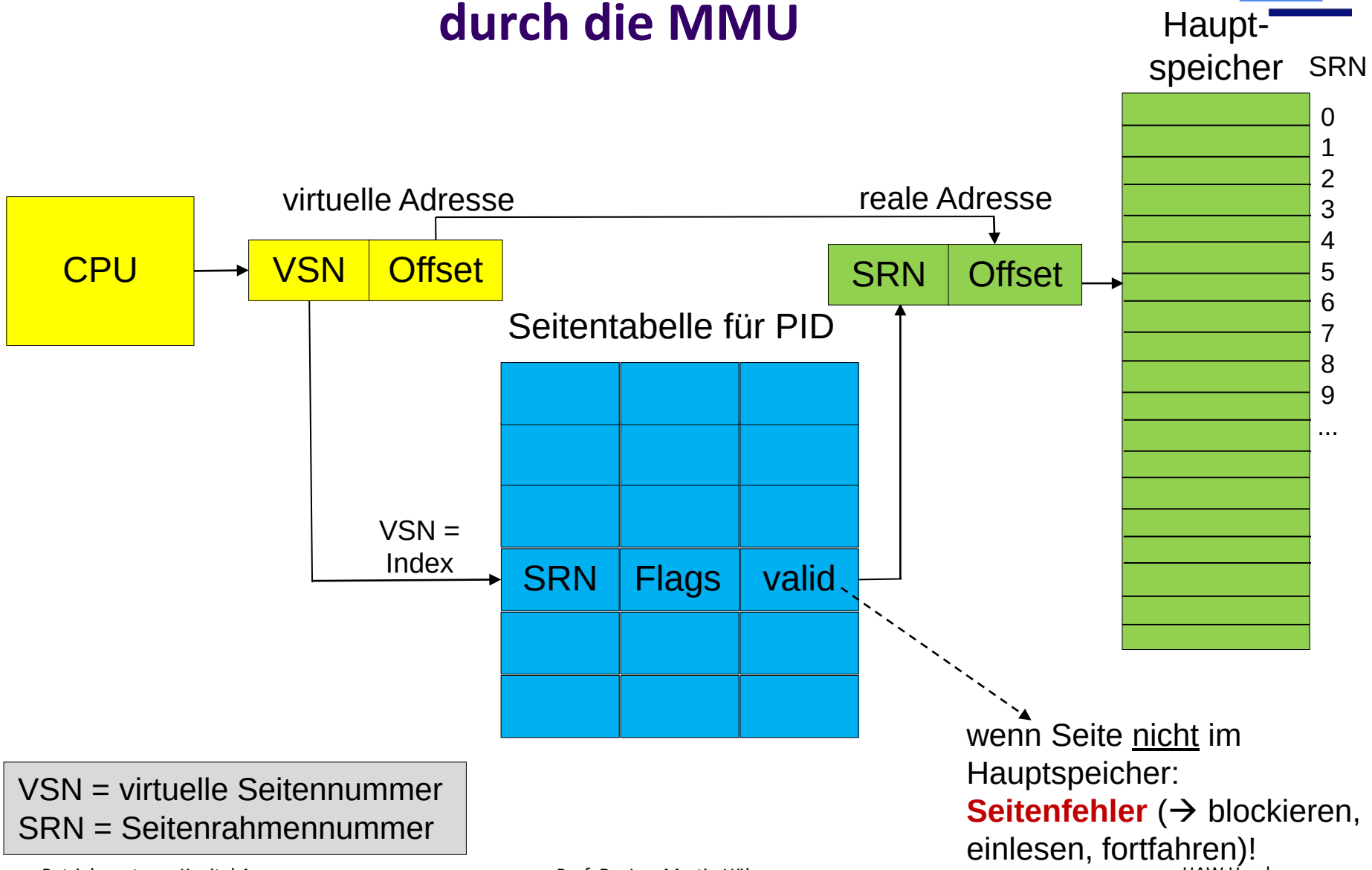
Virtuelle Adresse = Virtuelle Seitennummer * Seitengröße + Offset

1. Ermittlung der virtuellen Seitennummer:
   Virtuelle Seitennummer = floor (virtuelle Adresse / Seitengröße)
2. Ermittlung der Seitenrahmennummer durch Verwendung einer **Seitentabelle** (Index = virtuelle Seitennummer)
3. Ermittlung der realen Adresse (im Hauptspeicher):
   Reale Adresse = Seitenrahmennummer * Seitengröße + Offset
   (Offset = Virtuelle Adresse modulo Seitengröße
   = *„Positionsnummer" innerhalb der Seite*)

Berechnungstrick:

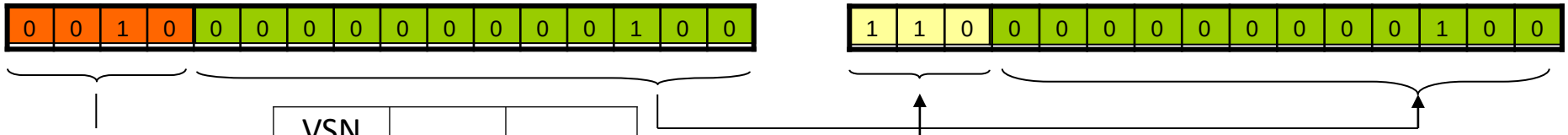
- Verwendung von Zweierpotenzen für Seitengrößen
- Bitweise Aufteilung der virtuellen Adresse für die Adressberechnung, z.B. bei einer virtuellen Adresse von 32 Bit und Seitengröße 4 KiB = 2^{12} Byte: 20 Bit für Seitennummer + 12 Bit für Offset
- Statt Division / Addition kann bitweises Trennen / Aneinanderhängen verwendet werden!

Ermittlung einer realen Adresse mittels Seitentabelle durch die MMU



VSN = virtuelle Seitennummer
SRN = Seitenrahmennummer

Beispiel: 64 KiB (=2^{16}) virtueller Adressraum, 32 KiB (=2^{15}) Hauptspeichergröße, Seitengröße 4 KiB (=2^{12})



4-Bit Virtuelle Seitennummer (VSN): **2**

VSN (Index)	SRN	valid
0	010	1
1	001	1
2	110	1
3	000	1
4	100	1
5	011	1
6	000	0
7	000	0
8	000	0
9	101	1
10	000	0
11	111	1
12	000	0
13	000	0
14	000	0
15	000	0

Seitenrahmen-nummer (SRN): **6** aus Seitentabelle

12-bit Offset: **4** (Seitengröße 2^{12}) wird direkt kopiert

Seitentabelle für PID X: 16 (=2^4) virtuelle Seiten

8 (=2^3) physikalische Seitenrahmen im Hauptspeicher vorhanden

Seitentabellen („Page Tables“)

- **Eine** Seitentabelle pro Prozess
- Ein Eintrag beschreibt eine Seite (Index = virtuelle Seitennummer)
- Aufbau eines Eintrags in einer Seitentabelle:
  - Seitenrahmennummer
  - Valid-Bit: 1 = im Hauptspeicher, 0 = auf Platte (→ *Zugriff löst Seitenfehler aus → Interrupt zum Einlesen von der Platte*) *[Present-/Absent-Bit]*
  - Zugriffsrechte (Schreiben/Lesen)
  - Verwaltungs-Flags: Referenced-Bit (R-Bit), Modified-Bit (M-Bit) werden bei Lese-Schreib-Zugriffen (R-Bit) oder Schreib-Zugriffen (M-Bit) auf die Seite von der Hardware gesetzt

Zahlen aus der Praxis

- Größen bei 32-Bit-Systemen:
  - Virtuelle Adresse: 32 Bit
    → max. 2^{32} = 4 GiB Virtueller Speicher adressierbar
  - Übliche Seitengröße: 2^{12} Byte (4 KiB)
  - max. Anzahl Seiten (max. Größe einer Seitentabelle)
    bei 4 KiB Seitengröße: 2^{20} Einträge (~ 1 Million)
- Größen bei 64-Bit-Systemen:
  - Virtuelle Adresse: 64 Bit
    → max. 2^{64} = 16 EiB (~ 1,8 * 10^{19} Byte) virtueller Speicher adressierbar
  - Übliche Seitengrößen: 2^{12} Byte (4 KiB), 2^{21} Byte (2 MiB), 2^{30} Byte (1 GiB)
  - max. Anzahl Seiten (max. Größe *einer* Seitentabelle)
    bei 4 KiB Seitengröße: 2^{52} Einträge
    bei 2 MiB Seitengröße: 2^{43} Einträge
    bei 1 GiB Seitengröße: 2^{34} Einträge (~ 16 Milliarden)

Implementierung von Seitentabellen

- Problem: Größe von Seitentabellen
  (eine Seitentabelle pro Prozess!)
  - effizienter Zugriff
  - effiziente Speicherung
- Lösungsmöglichkeiten (kombinierbar):
  1. Seitentabellen komplett im Hauptspeicher
  2. Caching von Seitentabelleneinträgen
  3. Aufteilung in mehrere Tabellen mit mehrstufigem Zugriff
  4. Invertierte Seitentabellen

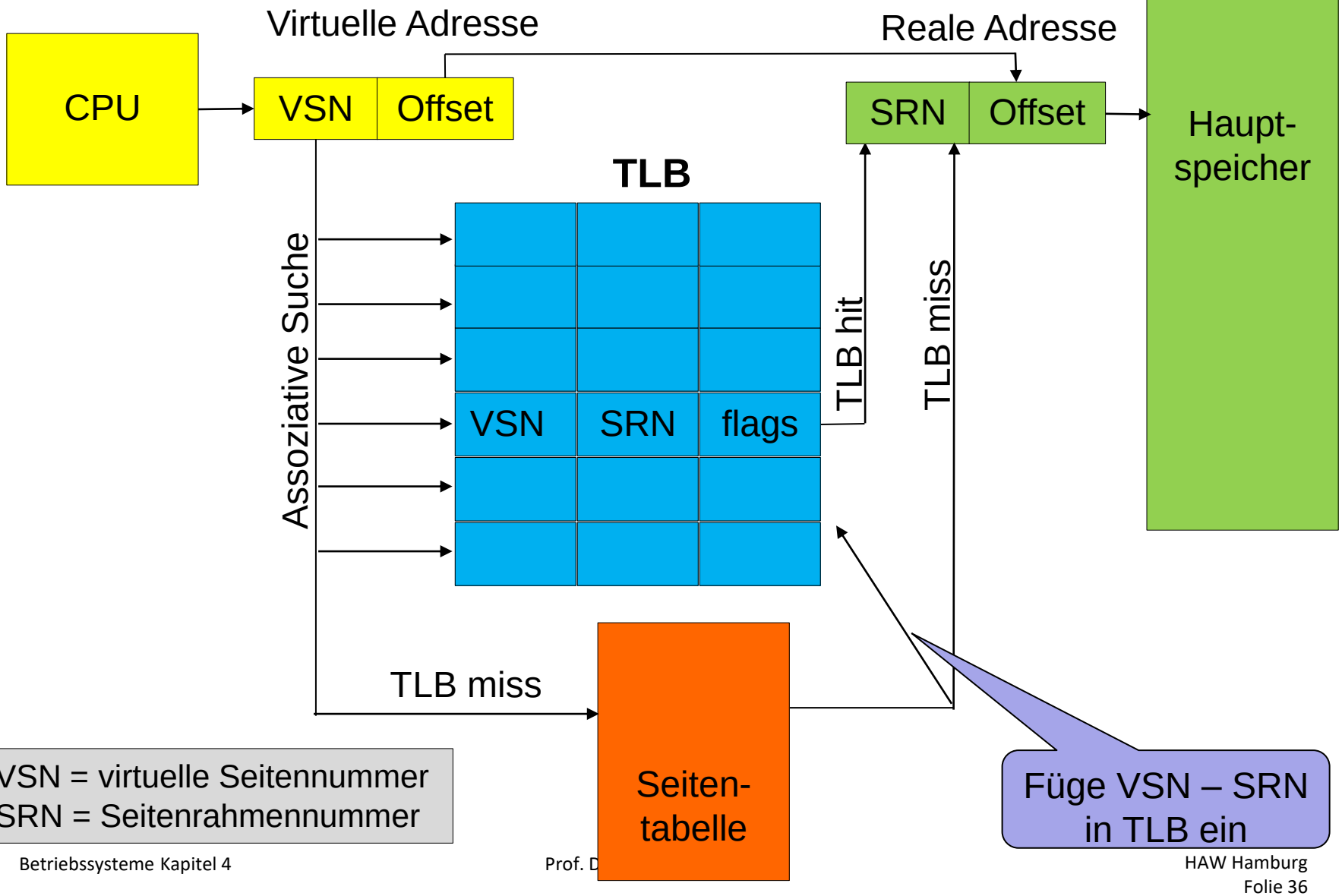
2. Caching von Tabelleneinträgen in Registern

- Einführung eines schnellen Cache in der MMU für die Seitentabellen-Einträge
- Wird **Translation Lookaside Buffer (TLB)** genannt
- Enthält kürzlich benutzte Einträge
- Größe: 64 bis 1024 Einträge
- Ist ein Assoziativ-Speicher: Parallele Suche nach VSN mittels Hardware - Unterstützung
- Trefferquoten in der Praxis: 80% - 98%

Translation Lookaside Buffer (TLB): Zugriffsvorgang

- Virtuelle Adresse → Hardware - Suche im TLB nach virtueller Seitennummer (VSN)
- Vorhanden: → direkte Umsetzung in die Seitenrahmennr. (SRN)
- Nicht vorhanden: → Hardware durchsucht Seitentabelle
  - Eintrag existiert & Seite ist im Hauptspeicher
    - Aktualisierung des TLB (durch Verdrängung eines alten Eintrags)
  - Sonst: Seitenfehler („Page Fault Interrupt“)
    - Seite laden & Seitentabelle aktualisieren

Ermittlung einer realen Adresse mittels TLB und Seitentabelle durch die MMU

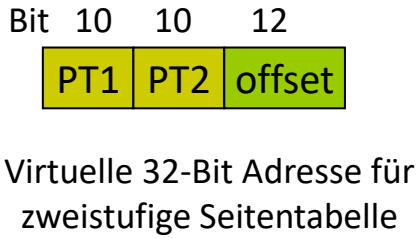


VSN = virtuelle Seitennummer
SRN = Seitenrahmennummer

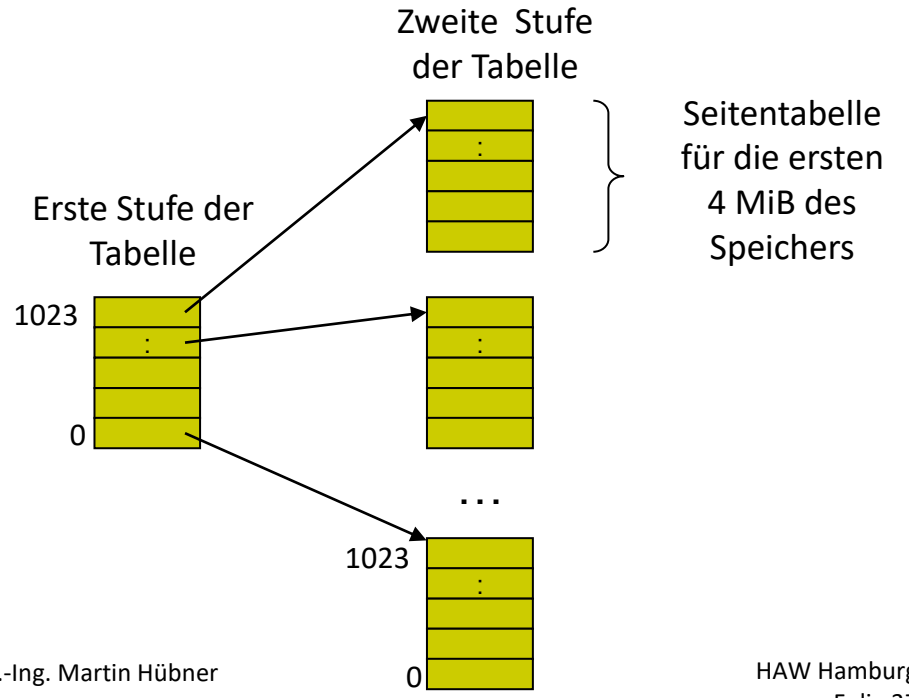
3. Aufteilung einer Seitentabelle in mehrere Tabellen mit mehrstufigem Zugriff

Nur wenige Tabellen müssen zeitgleich im Hauptspeicher geladen sein!

- Beispiel: 32-Bit Adresse zusammengesetzt aus 2x10-Bit Adressen für die Seitennummern und 12-Bit Offset: zweistufige Seitentabelle
- PT1-Feld ist Index für oberste Seitentabelle
- PT2-Feld ist Index für ausgewählte Seitentabelle der 2ten Stufe



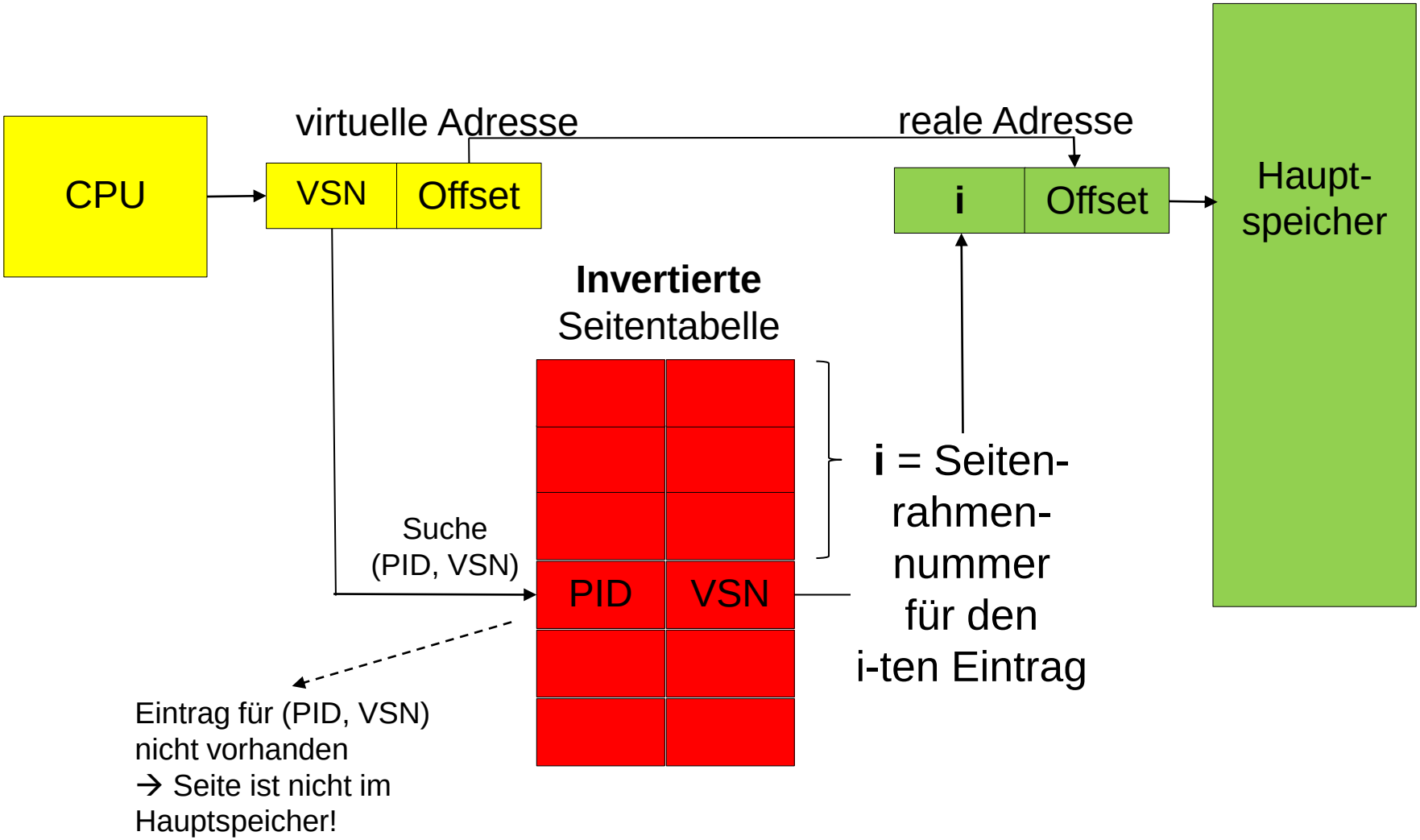
Virtuelle 32-Bit Adresse für zweistufige Seitentabelle



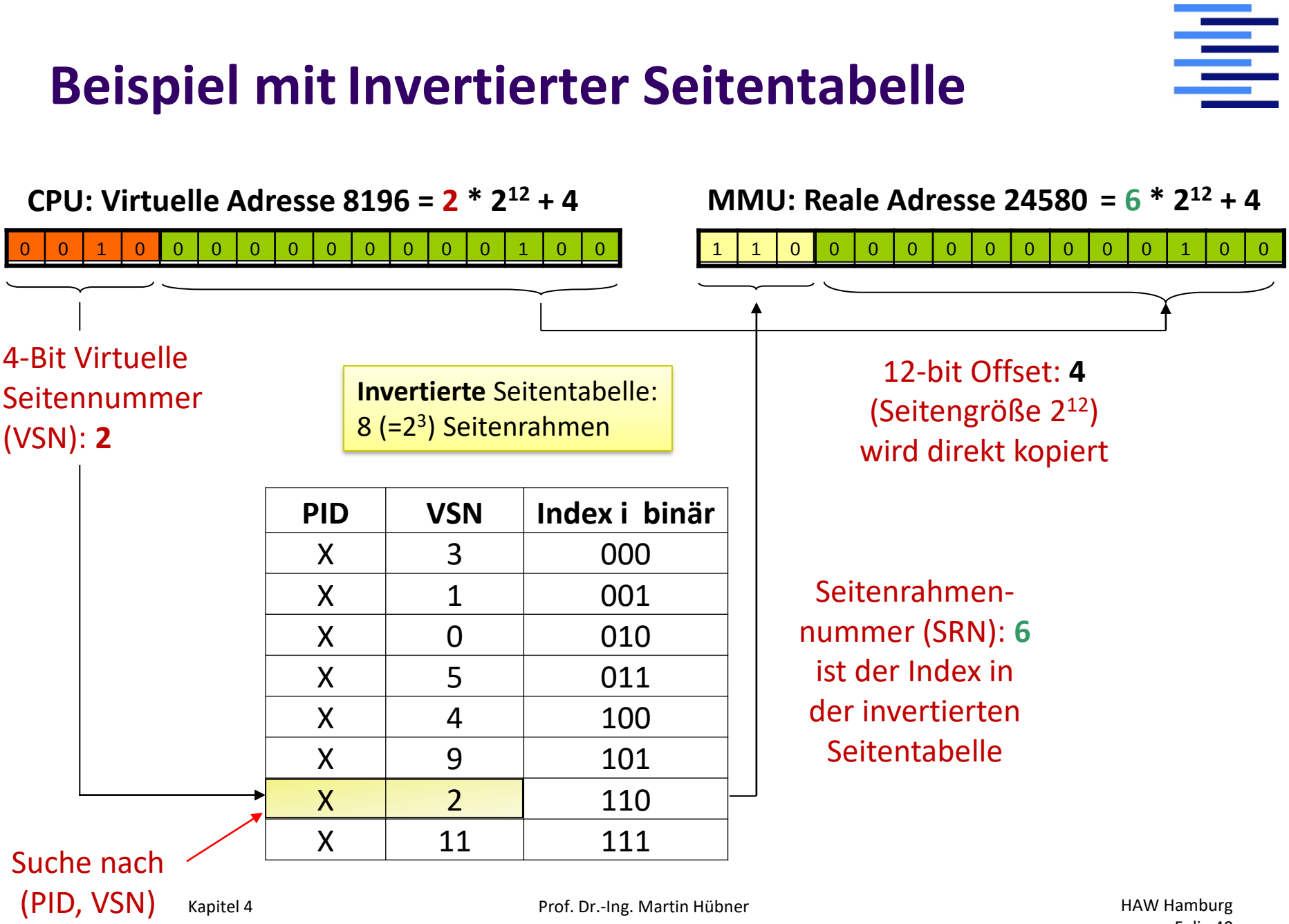
4. Invertierte Seitentabellen

- Idee: Zusätzlich zum TLB **eine** Tabelle für **alle** physischen Seitenrahmen des Hauptspeichers verwalten!
- Eigenschaften:
  - Für jeden Seitenrahmen im Hauptspeicher genau ein Eintrag
    - Prozess (**PID**) [„Besitzer“ des Seitenrahmens]
    - Virtuelle Seitennummer (**VSN**)
  - Unter Tabellenindex **i** steht der Eintrag für den Seitenrahmen mit der Seitenrahmennummer **i**
- Adressumsetzung: Suche (PID, VSN) in der invertierten Seitentabelle!
- Vorhanden: Bilde reale Adresse (**i * Seitengröße + Offset**)
- Nicht vorhanden: „normale“ Seitentabelle → Seitenfehler
- Problem: Suche in invertierter Seitentabelle aufwändig
  → Hash-Tabelle für virtuelle Seitennummern einsetzen!

Adressumsetzung mit invertierter Seitentabelle



Beispiel mit Invertierter Seitentabelle

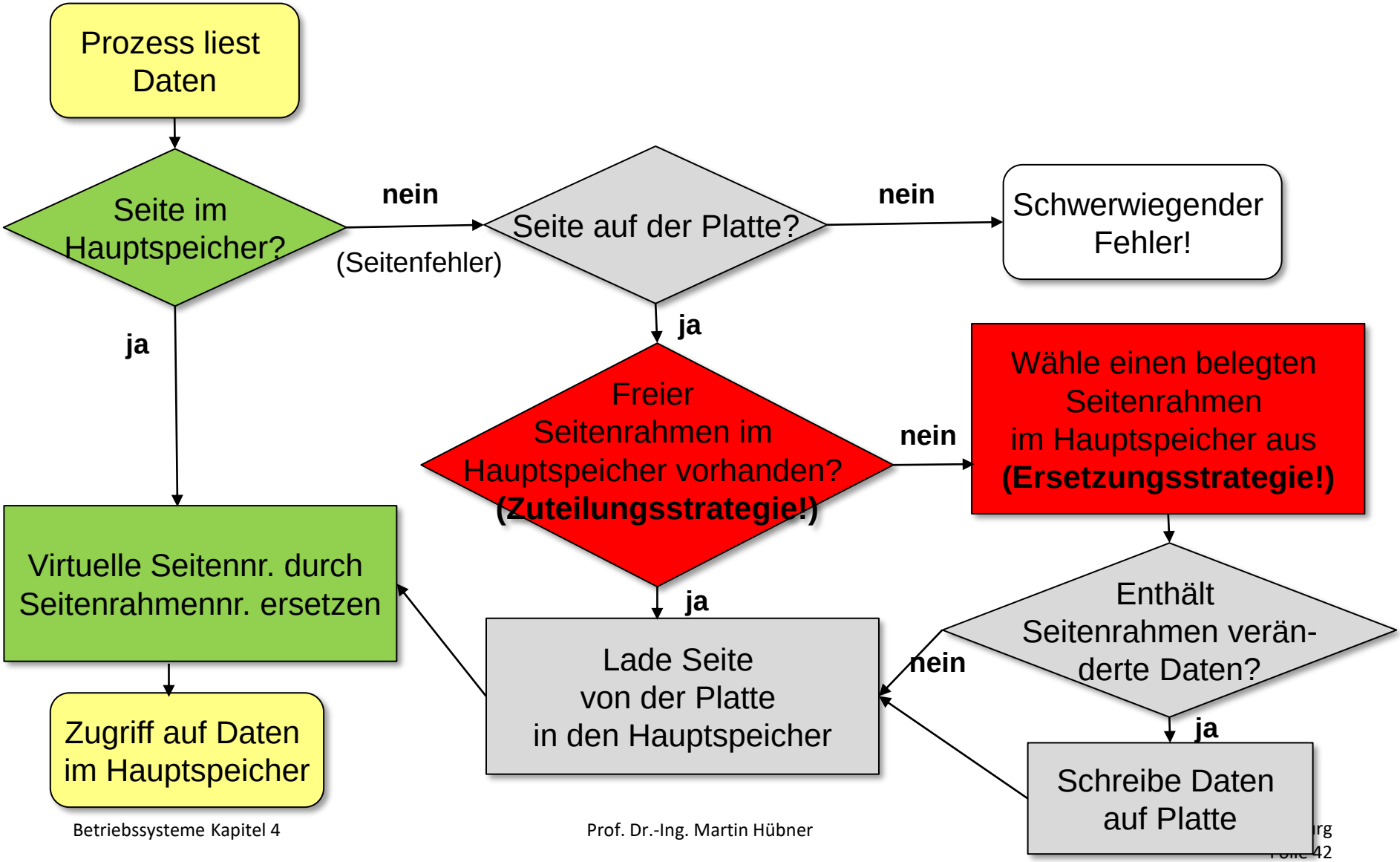


PID	VSN	Index i binär
X	3	000
X	1	001
X	0	010
X	5	011
X	4	100
X	9	101
X	2	110
X	11	111

Kapitel 4
Hauptspeicher-Verwaltung

1. Anforderungen und Grundlagen
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Übersicht Seitentransferprozess (vereinfacht)



Pagingstrategien

- **Ersetzungsstrategie**
  - Welche Seiten werden ersetzt (verdrängt), wenn Hauptspeicherplatz benötigt wird?
- **Speicherzuteilungsstrategie**
  - Wieviele Hauptspeicherseiten bekommt ein Prozess zugeteilt?

Ersetzungsstrategien

- Auftreten eines Seitenfehlers („Page fault“):
  - Laden einer neuen Seite von der Platte
    → Verdrängung einer „alten“ Seite im Hauptspeicher
  - Was passiert mit einer alten Seite?
    - Verändert (M-Bit): Zurückschreiben auf die Platte
    - Nicht verändert: Sofort Überschreiben mit neuer Seite
- **Problem**: Auswahl der „alten“ Seite im Hauptspeicher, die durch die neue Seite ersetzt wird!
- **Optimum**: Die Seite ersetzen, die am längsten in der nächsten Zeit nicht mehr benutzt werden wird

➔ Die optimale Strategie kennt die Zukunft!

- Vorhersage über die genaue Benutzung der Seiten in der Zukunft ist fast immer unmöglich
- Die guten Strategien treffen daher aus dem Verhalten in der jüngsten Vergangenheit Entscheidungen für die nahe Zukunft

Ersetzungsstrategie: First-In, First-out (FIFO)

- Es wird immer die Seite ersetzt, die am längsten im Hauptspeicher ist
- Einfachstes Verfahren
- Verwaltet die Hauptspeicher-Seiten eines Prozesses in einer Liste (Kopf: älteste Seite, Ende: jüngste Seite)
- Berücksichtigt nicht, ob die Seite gerade jetzt häufig benutzt wird
  - Könnte sofort nach Ersetzung wieder gebraucht werden
- Ineffizientes Verhalten

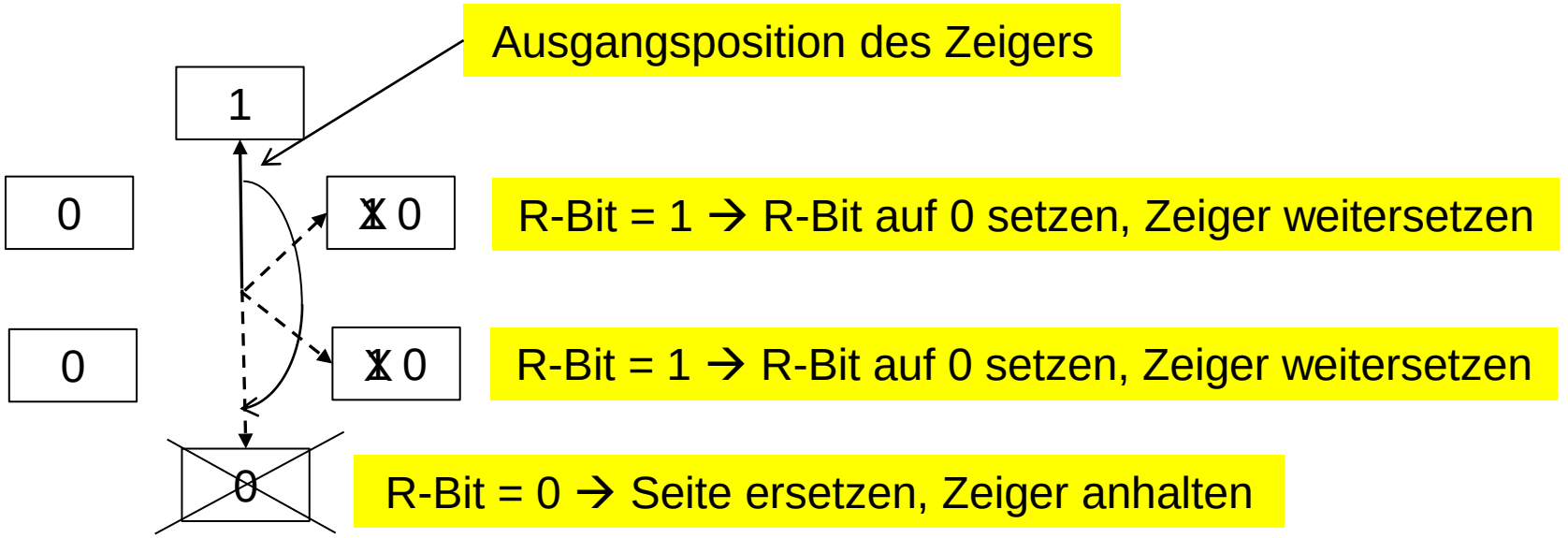
Ersetzungsstrategie: Least Recently Used (LRU)

- Annäherung an die optimale Strategie
- Ersetzt die Seite, die am längsten nicht benutzt wurde
- Nach dem **Lokalitätsprinzip** ist die Wahrscheinlichkeit hoch, dass die Seite auch in Zukunft lange nicht benutzt wird
- Implementierung: Jede Seite müsste mit einem Zeitstempel für den letzten Zeitpunkt der Benutzung versehen werden
  - Das ist in der Realität aber extrem aufwendig!

Ersetzungsstrategie: Clock-Algorithmus

- Ziel: Leicht implementierbare Annäherung an LRU
- Führt eine zirkulare Liste („Clock“) mit Zeigern auf Seitentabelleneinträge für alle Seiten des Prozesses, die sich im Hauptspeicher befinden (mit Speicherung der letzten Zeigerposition)
- Verwendet das Reference-Bit (R-Bit)
  - Wenn die Seite geladen wird: R-Bit = 0
  - Wenn auf die Seite zugegriffen wird: R-Bit = 1 (durch die MMU)
- Seitenfehler → Zeiger durchläuft die Clock-Liste ab der letzten Position (im „Uhrzeigersinn“)
  - Seite mit R-Bit = 1 → R-Bit zurücksetzen auf 0
  - Erste gefundene Seite mit R-Bit = 0 → Seite ersetzen (auch in der Clock-Liste)!

Clock-Algorithmus: Beispiel



Die ersetzte Seite wurde während des gesamten letzten Durchlaufs nicht gebraucht

= Annäherung an „lange“ nicht gebraucht!

Speicherzuteilungsstrategien: Grundlegende Überlegungen (1)

- Je **weniger Platz für den einzelnen Prozess** zur Verfügung steht, desto **mehr Prozesse** können im Hauptspeicher resident sein
  (➔ Seitenzahl pro Prozess minimieren!)
- Stehen **einem Prozess zu wenig Seiten** zur Verfügung, dann wird die **Seitenfehlerrate** trotz Lokalitätsprinzip sehr hoch sein
  (➔ Seitenzahl pro Prozess maximieren!)
- **Über eine bestimmte Größe** hinaus wird sich zusätzlicher Speicher nur **geringfügig** auf die Seitenfehlerrate auswirken
  (➔ Seitenzahl pro Prozess optimieren!)

Speicherzuteilungsstrategien: Grundlegende Überlegungen (2)

- **Verteilung der freien Hauptspeicherseiten auf die existierenden Prozesse:**
  - Einem Prozess wird eine **feste Anzahl** von Hauptspeicherseiten zugebilligt.
  - Einem Prozess werden während seiner Ausführung abhängig von seinem Verhalten **(Seitenfehlerrate)** eine **variable Anzahl** von Hauptspeicherseiten zur Verfügung gestellt.
- **Strategien nach Auftreten von Seitenfehlern:**
  - Bei **lokalen Strategien** muss eine Seite des aktiven Prozesses ersetzt werden.
  - Bei **globalen Strategien** sind alle Seiten im Hauptspeicher Kandidaten für die Ersetzung.

fest variabel lokal global

Speicherzuteilung Variante 1: Feste Seitenanzahl und lokale Strategie

- Einem Prozess steht für seine Ausführung eine feste Anzahl von Hauptspeicherseiten zur Verfügung.
- Tritt ein Seitenfehler auf, dann muss das Betriebssystem entscheiden, welche andere Seite dieses Prozesses ersetzt werden muss
  - Das wesentliche Problem liegt in der Festlegung der Seitenzahl (x % der gesamten Prozessgröße?)

→ In der Regel zu unflexibel!

lokal fest

Speicherzuteilung Variante 2:
Variable Seitenzahl und globale Strategie

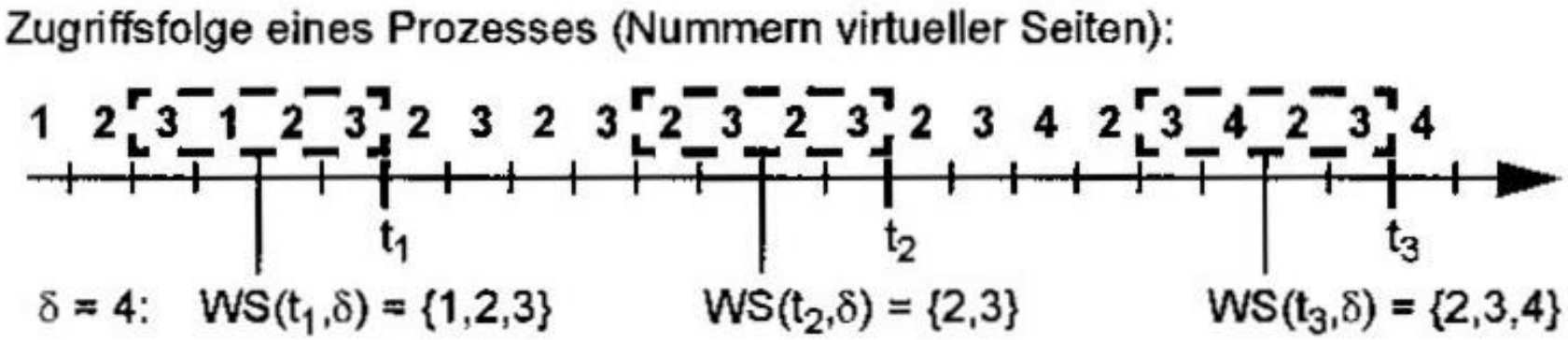
global variabel

- Den im Hauptspeicher befindlichen Prozessen stehen eine variable Anzahl von Seiten zur Verfügung.
- Seitenfehler → Zuweisung einer freien Seite, falls verfügbar
- Keine freien Seiten mehr verfügbar → Seite zur Ersetzung auswählen (beliebiger Prozess!)
- Zu viele Prozesse im Hauptspeicher → u.U. sind die zur Verfügung stehenden Speicherbereiche nicht mehr ausreichend und die Seitenfehlerrate wird sehr groß
- Jeder Prozess, der zusätzliche Seiten benötigt, erhält diese auf Kosten von Seiten anderer Prozesse. Da diese aber ebenfalls noch benötigt werden, werden in immer schnellerer Folge weitere Seitenfehler erzeugt. (Seitenflattern, engl. „**Thrashing**“). Die Prozesse verbrauchen dann für das Seitenwechseln mehr Zeit als für ihre Ausführung.

Speicherzuteilung Variante 3: Variable Seitenzahl und lokale Strategie

Working Set – Modell

- Der **Working Set WS (t,δ)** eines Prozesses zur Zeit t ist die Menge aller Seiten, die er bei den letzten δ Speicherzugriffen angesprochen hat.
- Der Umfang des Working Set ist abhängig von δ. Im Extremfall umfasst der Working Set das gesamte Programm
- Je größer der Working Set, umso geringer die Seitenfehlerrate
- Beispiel:



Variable Seitenzahl und lokale Strategie: Working Set-Strategie

lokal variabel

- Der **Working Set** jedes Prozesses wird beobachtet.
- Seitenfehler → Zuweisung einer freien Seite, falls verfügbar
- Periodisch wird die zugewiesene Seitenanzahl an den Working Set angepasst: diejenigen eigenen Seiten werden aus dem Speicher entfernt, die nicht mehr zum Working Set gehören
- Ein Prozess darf nur dann ausgeführt werden, wenn sein Working Set im Hauptspeicher resident ist

Mögliche Implementierung (grobe Annäherung):

- Periodisch werden diejenigen Seiten aus dem Speicher entfernt, deren Reference-Bit = 0 ist
  (kein Zugriff erfolgt seit letzter Überprüfung).
- Seiten mit Reference-Bit =1 (~ Working Set):
  Reference-Bit zurücksetzen!

Alternative / Ergänzung: Page-Fault Frequency (PFF) - Strategie

- Für die Seitenfehlerrate werden untere und obere Grenzen definiert.
- Seitenfehler → Ersetzung einer eigenen Seite
- Wird die untere Grenze erreicht, dann werden ihm Speicherseiten weggenommen
- Erreicht ein Prozess die obere Grenze, dann werden ihm – wenn möglich – neue Speicherseiten hinzugefügt. Sind keine weiteren Speicherseiten verfügbar, so muss der Prozess suspendiert und ausgelagert werden.

Kapitel 4
Hauptspeicher-Verwaltung

1. Anforderungen und Grundlagen
2. Virtueller Speicher
   a) Einführung und Prinzipien
   b) Paging
   c) Pagingstrategien
   d) Unix / Windows

Virtueller Speicher in UNIX: Paging

- Speicherzuteilung: Variable Seitenzahl und globale Strategie
- Seitenfehler → Zuweisung einer freien Seite
- Page Daemon (Prozess-ID 2)
  - Wacht alle 250 ms auf
  - Falls Anzahl freier Seiten zu klein: Verwendet modifizierten Clock-Algorithmus, um beliebigen Prozessen Seiten zu entziehen
- Swapper
  - lagert Prozesse aus, falls zuviele Seitenfehler auftreten
  - lagert erst wieder ein, wenn genügend freie Seiten zur Verfügung stehen (Scheduling!)

Virtueller Speicher in Windows: Paging (1)

- Speicherzuteilung: Variable Seitenzahl und lokale Strategie
- Eigene Working Set-Definition: Alle Seiten des Prozesses, die sich momentan im Hauptspeicher befinden
- Ein Working-Set hat eine minimale und maximale Grenze (bei Start für alle Prozesse gleich)
- Seitenfehler:
  - **Working Set < Maximum** → neue Seite hinzufügen
  - sonst: Seite aus Working Set ersetzen! (lokal)
  - bei vielen Seitenfehlern: Maximum für den Prozess erhöhen (Kombination mit PFF-Strategie)

Virtueller Speicher in Windows: Paging (2)

- Balance-Set-Manager
  - Wacht einmal pro Sekunde auf
  - Falls Anzahl freier Seiten zu klein: Startet Working-Set-Manager, um Prozessen Seiten zu entziehen
- Working-Set-Manager
  - Untersucht alle Prozesse in definierter Reihenfolge (große Prozesse vor kleinen, Hintergrundprozesse vor Vordergrundprozessen)
  - Falls **Working Set < Minimum** oder bereits **viele Seitenfehler** → ignorieren
  - Sonst: Prozess-Seiten anhand von modifiziertem Clock-Algorithmus aus Hauptspeicher (Working Set) entfernen (*Anzahl? - Berechnung ist komplex!)*

Ende des 4. Kapitels: Was haben wir geschafft?

4. Hauptspeicher-Verwaltung

4.1 Anforderungen und Grundlagen

4.2 Virtueller Speicher

- 4.2.1 Einführung und Prinzipien
- 4.2.2 Paging
- 4.2.3 Pagingstrategien
- 4.2.4 Unix / Windows