2405 Betriebssysteme
VII. Hauptspeicherverwaltung

Thomas Staub, Markus Anwander
Universität Bern

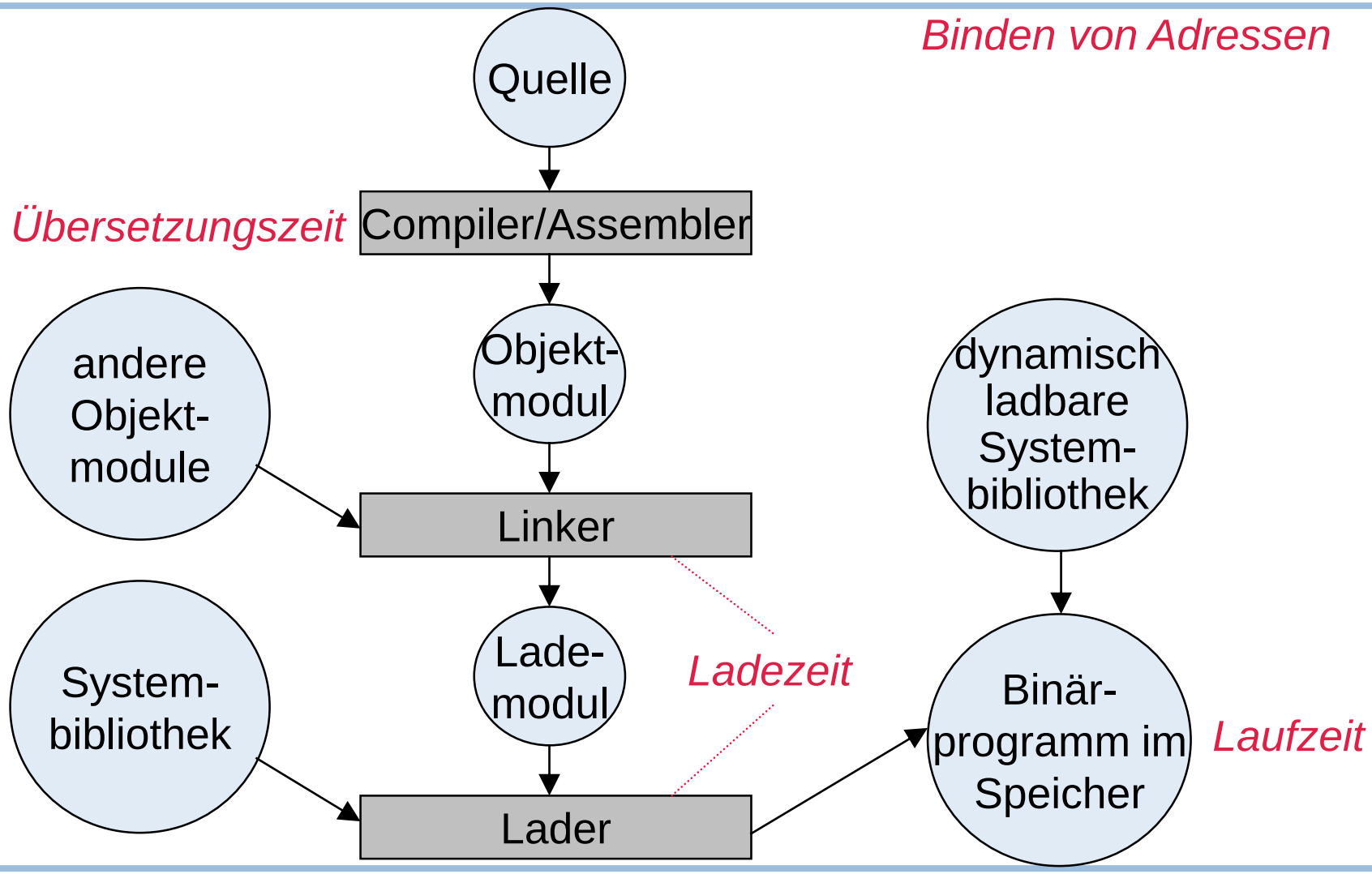
b
UNIVERSITÄT
BERN

Inhalt

1.1.1 Binden von Speicheradressen

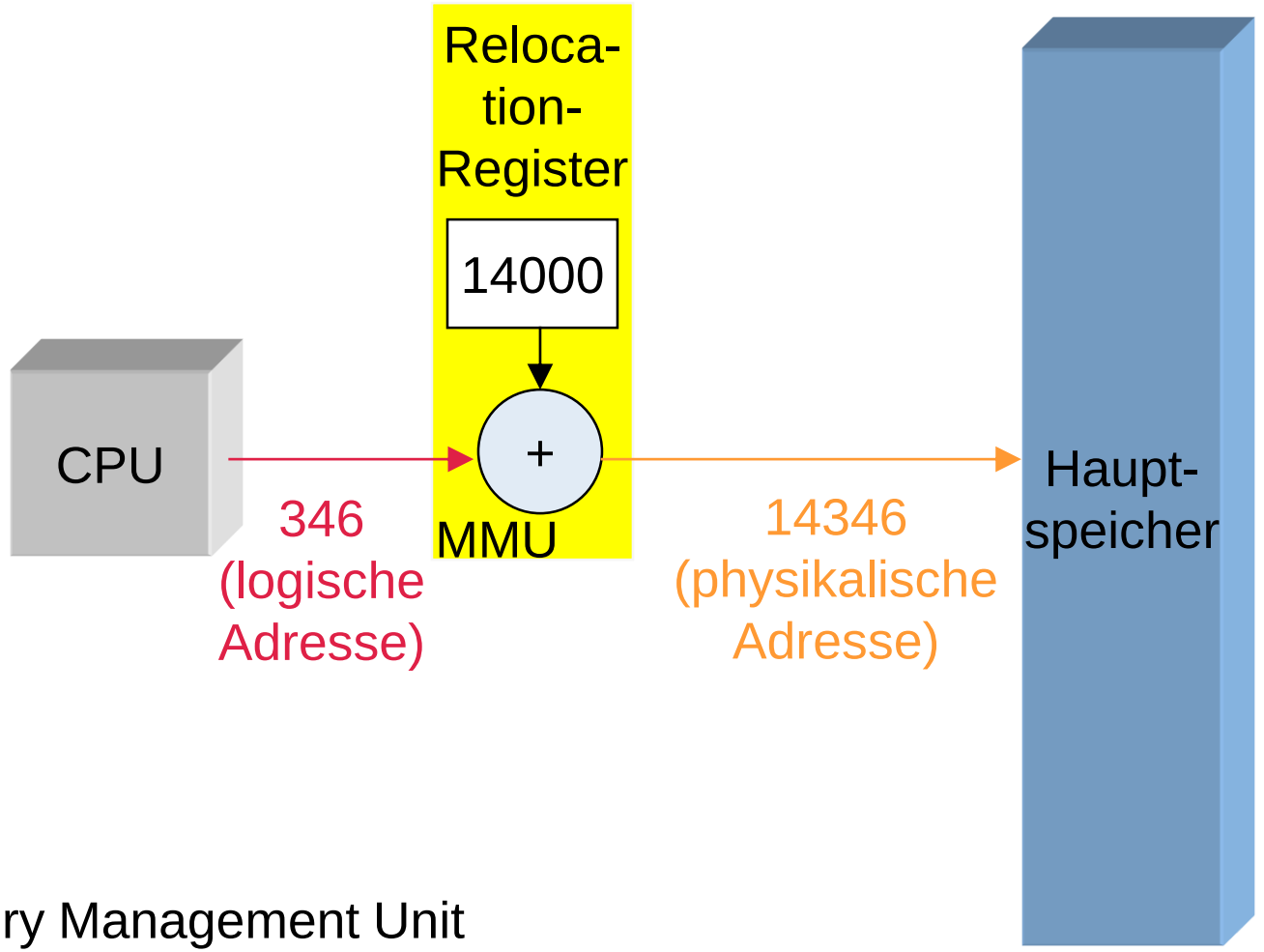
- Zur Ausführung eines Programms muss dies in den Hauptspeicher geladen werden.
- Binden der Speicheradressen zur
  - Übersetzungszeit
    - absoluter Code
  - Ladezeit
    - verlagerbarer Code, z.B. durch relative Adressierung
    - Anpassung der Adressen beim Laden
  - Ausführungszeit / Laufzeit
    - Prozesse können während Ausführung im Hauptspeicher verschoben werden.
    - virtueller Speicher (erfordert Hardware-Unterstützung)

1.1.2 Binden von Speicheradressen



1.2 Logische und Physikalische Adressen

Laden des Relocation-Register bei Prozesswechsel



MMU: Memory Management Unit

1.3 Dynamisches Laden

- Nicht benötigte Routinen werden nicht geladen.
- Routinen werden erst bei Aufruf geladen.
- Aufrufende Routine prüft, ob aufgerufene Routine geladen ist.
- Falls nicht, wird Lader beauftragt, Routine in Hauptspeicher zu laden und Adresstabellen zu modifizieren.
- nützlich, wenn grosse Codesegmente nur selten benötigt werden
- Implementierung durch Benutzerprogramm ohne Betriebssystemunterstützung möglich (ggf. Unterstützung durch Bibliotheken)

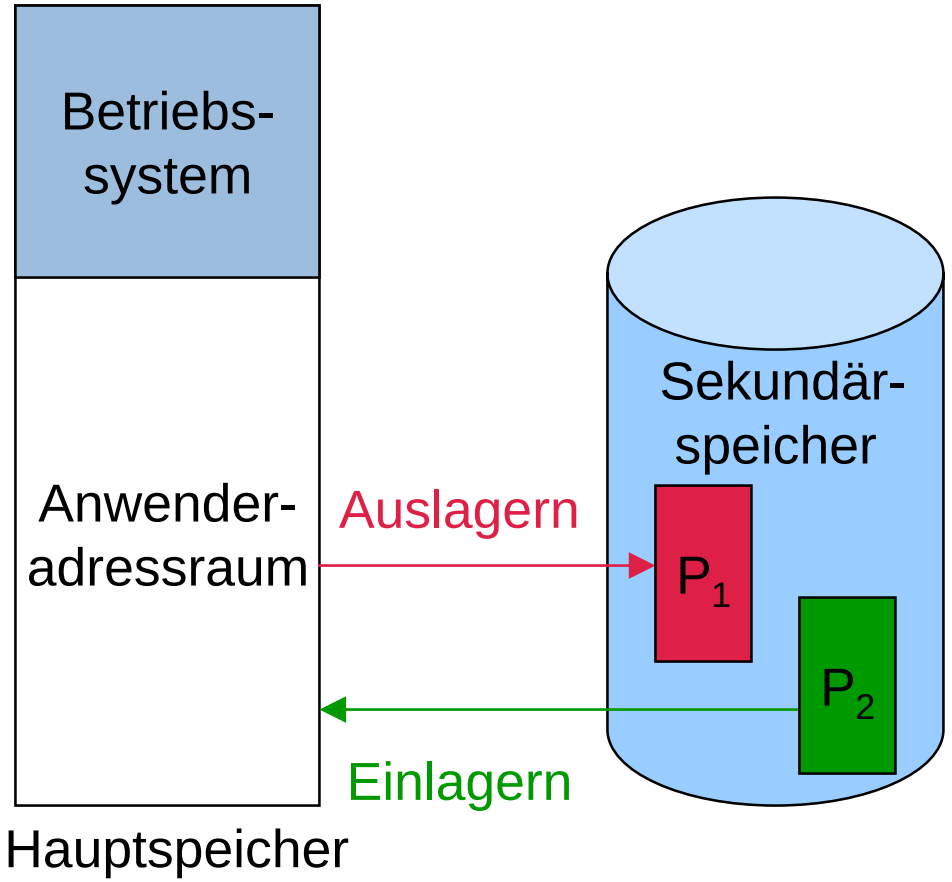
1.4 Dynamisches Binden

- ähnlich wie dynamisches Laden
- *Binden* während der Ausführungszeit
- kleines Codestück (Stub) zum Lokalisieren und Laden der speicherresidenten Bibliotheksroutine
  - prüft Existenz der Routine im Speicher
  - lädt Routine, falls sie sich nicht schon im Speicher befindet
  - ersetzt sich selbst durch Adresse der Routine und führt diese aus
- Bibliotheks-Updates ohne Neuübersetzung / Linken der Anwendung
- Teilen von Codesegmenten
- Shared Library / Dynamic Link Library
- erforderliche Betriebssystemunterstützung:
  Nur Betriebssystem kennt alle geladene Routinen.

1.5 Swapping

- Prozess kann temporär vom Hauptspeicher auf ausreichend grossen Sekundärspeicher (z.B. Festplatte) ausgelagert und später wieder eingelagert werden.
- Swapper lagert gesamten Speicherbereich eines Prozesses ein bzw. aus.
- kann auf Scheduling-Informationen basieren
- erfordert Binden der Adressen zur Laufzeit
- zeitaufwändig (abhängig von Datenmenge)

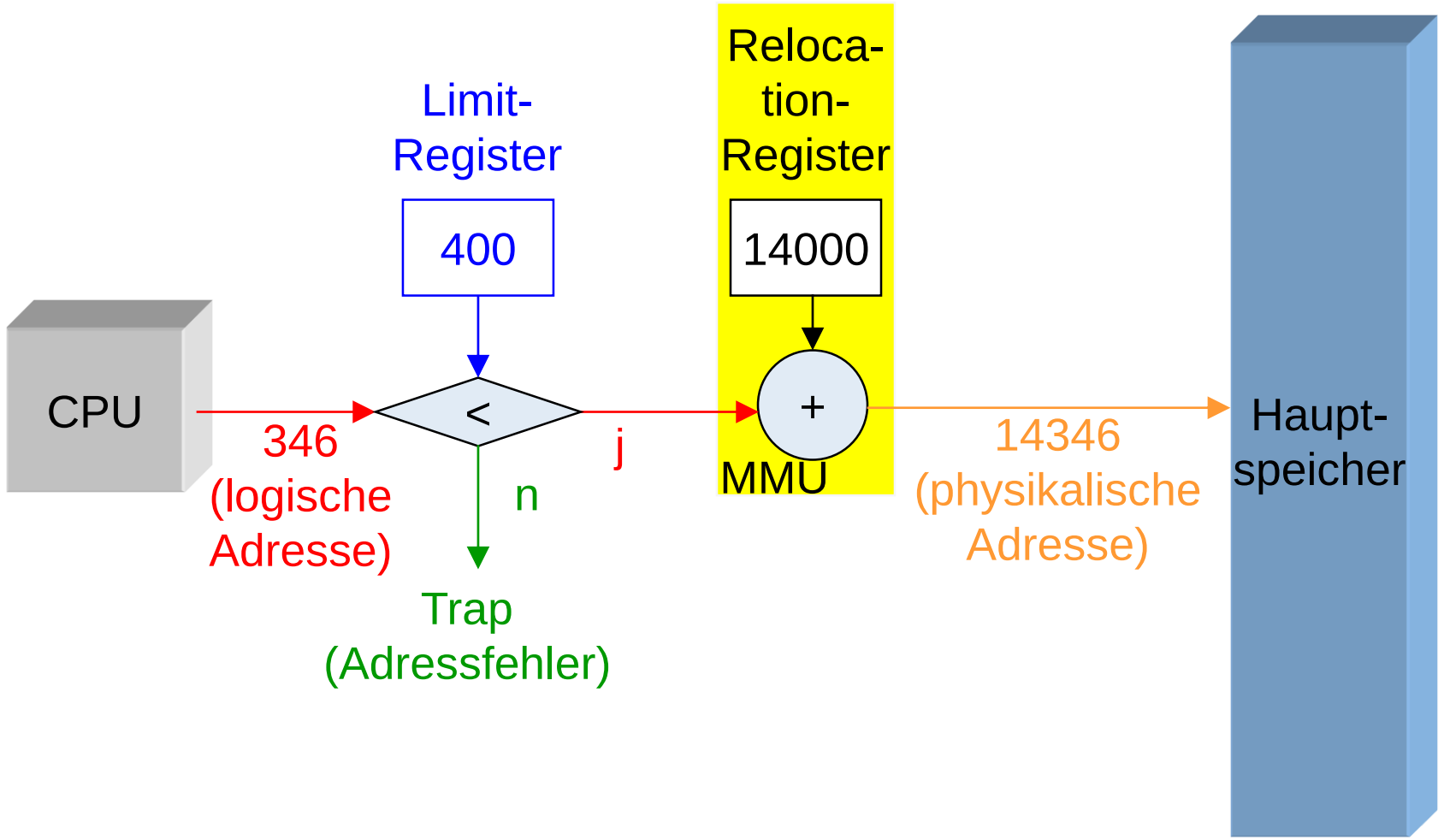
zb blockierter prozess



2. Zusammenhängende Hauptspeicherallokation

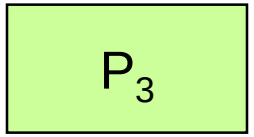
- Aufteilen des Hauptspeichers in 2 Partitionen
  - Betriebssystem
  - Benutzerprozesse
- Annahme: Prozess belegt einen zusammenhängenden Speicherbereich (vgl. Swapping)
- Ziel: gleichzeitige Verfügbarkeit von Benutzerprozessen im Hauptspeicher

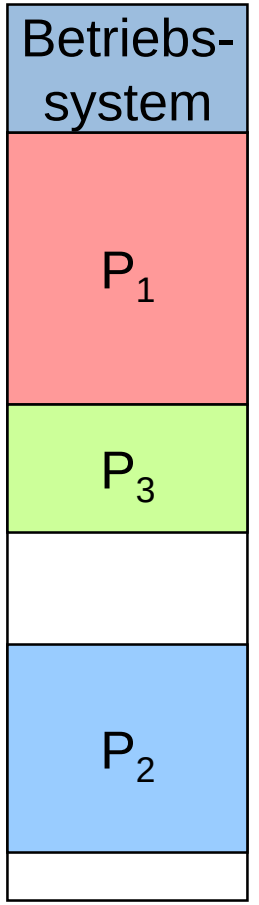
2.1 Speicherschutz



2.2 Hauptspeicherallokation

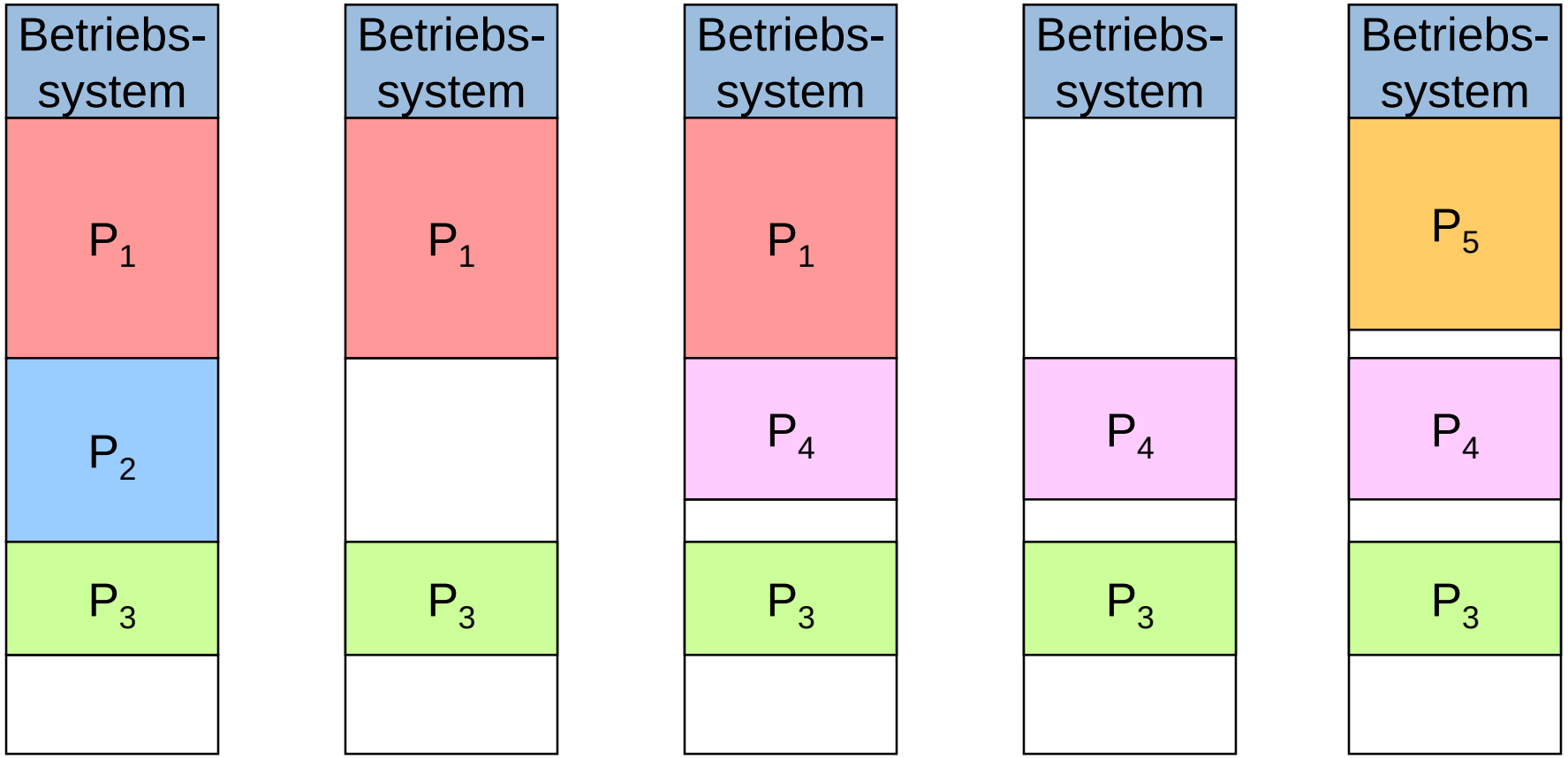
- Partitionen = zusammenhängende, belegte Speicherbereiche
- Einfacher Ansatz (wenig effizient):
  - Partitionen mit fester Grösse, 1 Partition pro Prozess
- Alternative
  - Löcher = freie Hauptspeicherteile verschiedener Grösse
  - Betriebssystem hält Tabelle, die freie und belegte Hauptspeicherteile beschreibt.
  - Anfangs wird der Speicher als ein grosses Loch interpretiert.
  - Betriebssystem wählt einen Prozess aus und ordnet dem Prozess eine Partition aus den Löchern zu: Loch → Partition + kleineres Loch
  - Speicherfreigabe bei Terminierung von Prozessen: Partition → Loch, ggf. Verschmelzen von Löchern
  - Betriebssystem kann Input Queue so ordnen, dass möglichst viele Anforderungen erfüllt werden.





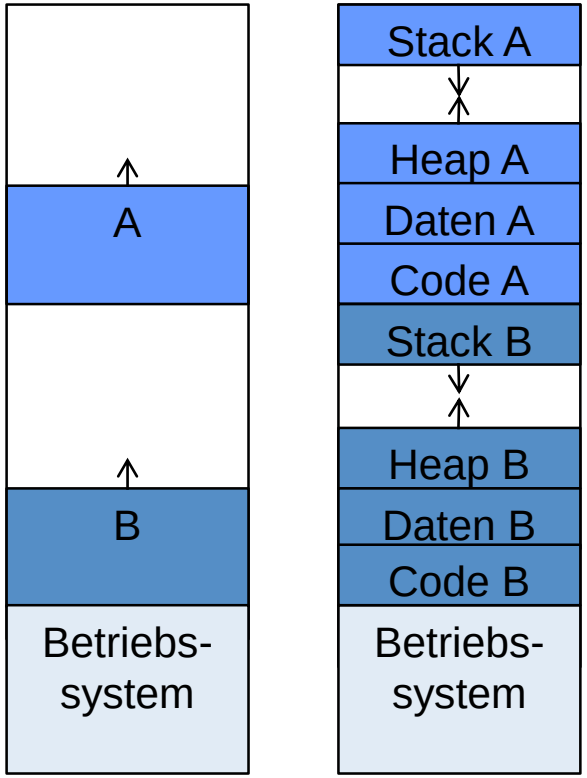
2.3 Hauptspeicherallokation und Scheduling

Round Robin (Quantum 1): P_1(10 ZE), P_2(5), P_3(11), P_4(8), P_5(15)

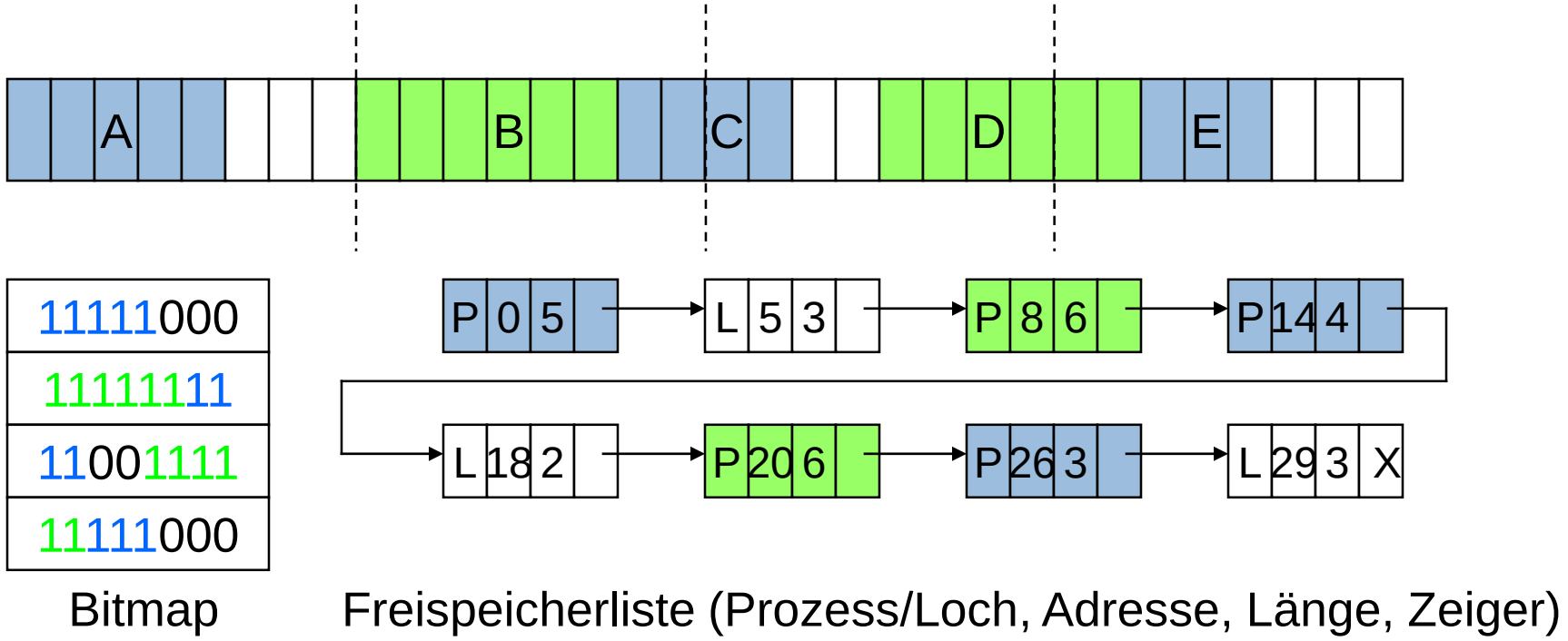


2.4 Wachsen von Prozessen

Unterstützen von einem oder zwei wachsenden Speicherbereichen, z.B. Stack und Heap



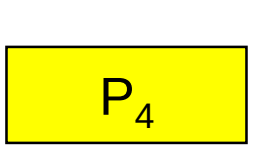
2.5 Hauptspeicherverwaltung mit Bitmaps und verketteten Listen

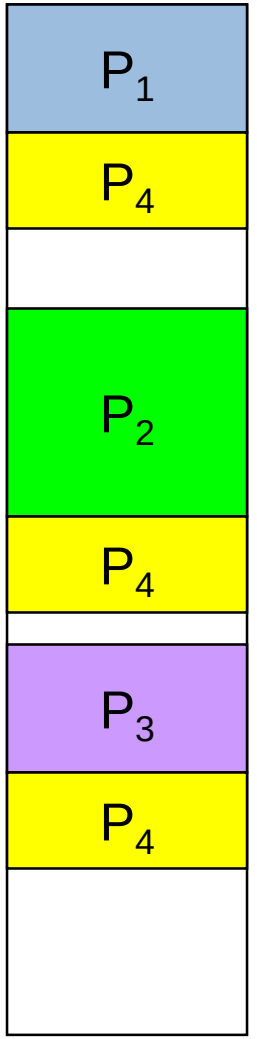


teilen speicher in frames auf.

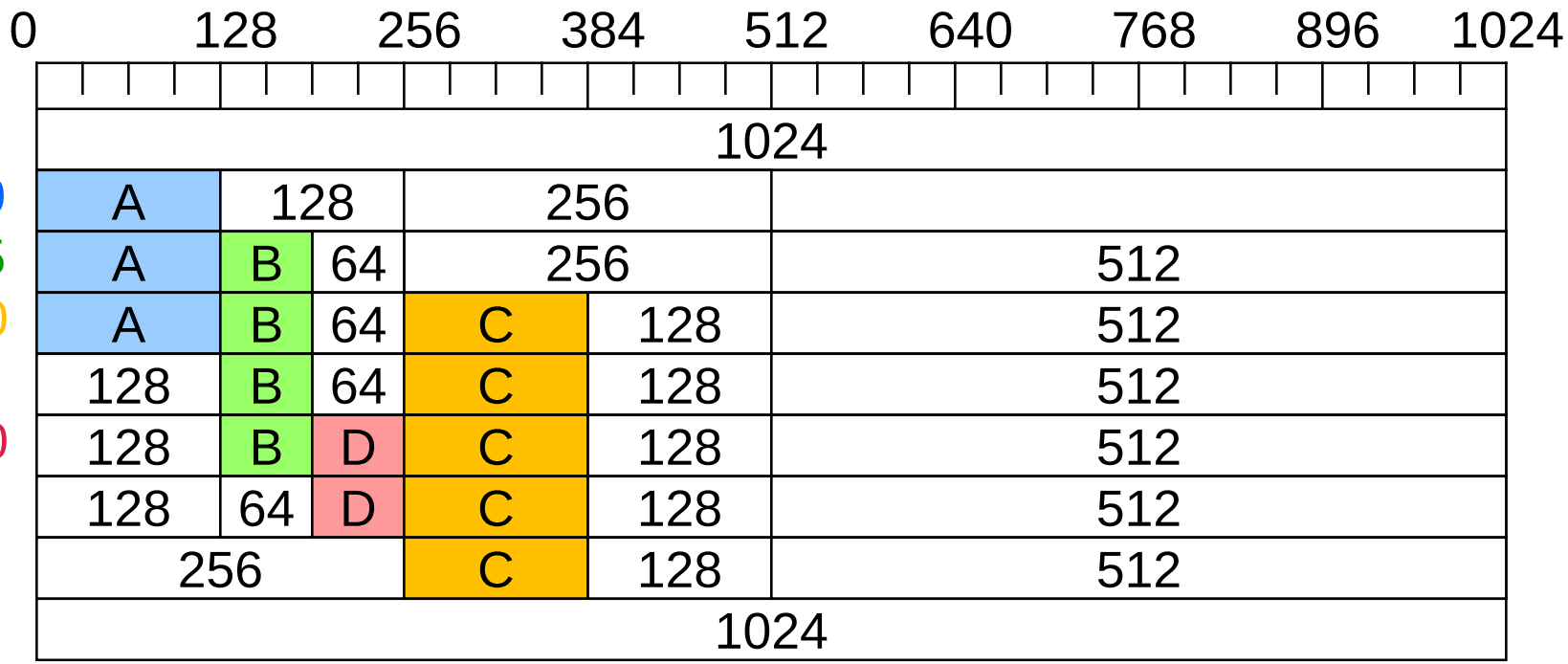
2.6 Dynamische Hauptspeicherallokation

- Annahme: Verwaltung über verkettete Listen
- Problem:
  Wie soll eine Speicherallokationsanforderung der Grösse N bedient werden ?
- Kriterien:
  - Geschwindigkeit (Durchsuchen des gesamten Speichers !)
  - Verschnitt
- Strategien
  - First-fit: alloziere Partition aus dem ersten Loch, das gross genug ist
  - Next-fit: wie first-fit, Suche wird da fortgesetzt, wo sie zuletzt beendet wurde
  - Best-fit: alloziere Partition aus dem kleinsten Loch
  - Worst-fit: alloziere Partition aus dem grössten Loch
  - Quick-fit: Listen mit Löchern verschiedener üblicher Grössen
- Alternative: Buddy-System
  - Nutzung zur Verwaltung von Kern-Speicher





2.6.1 Buddy System



halbieren bis nicht mehr reinpasst

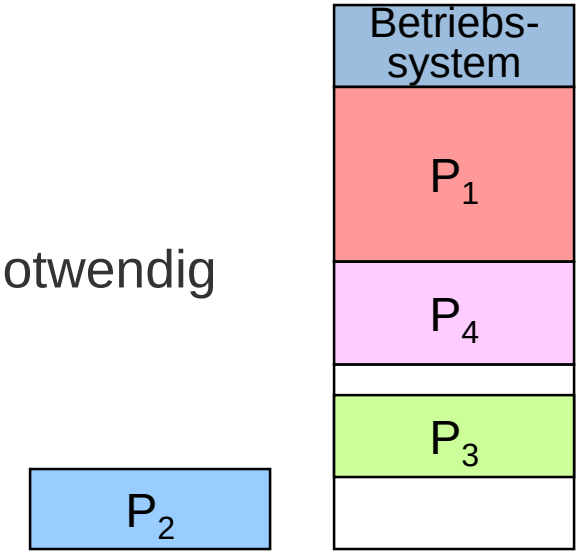
Notation:

X: +Z Prozess X fordert Speicher der Grösse Z an.

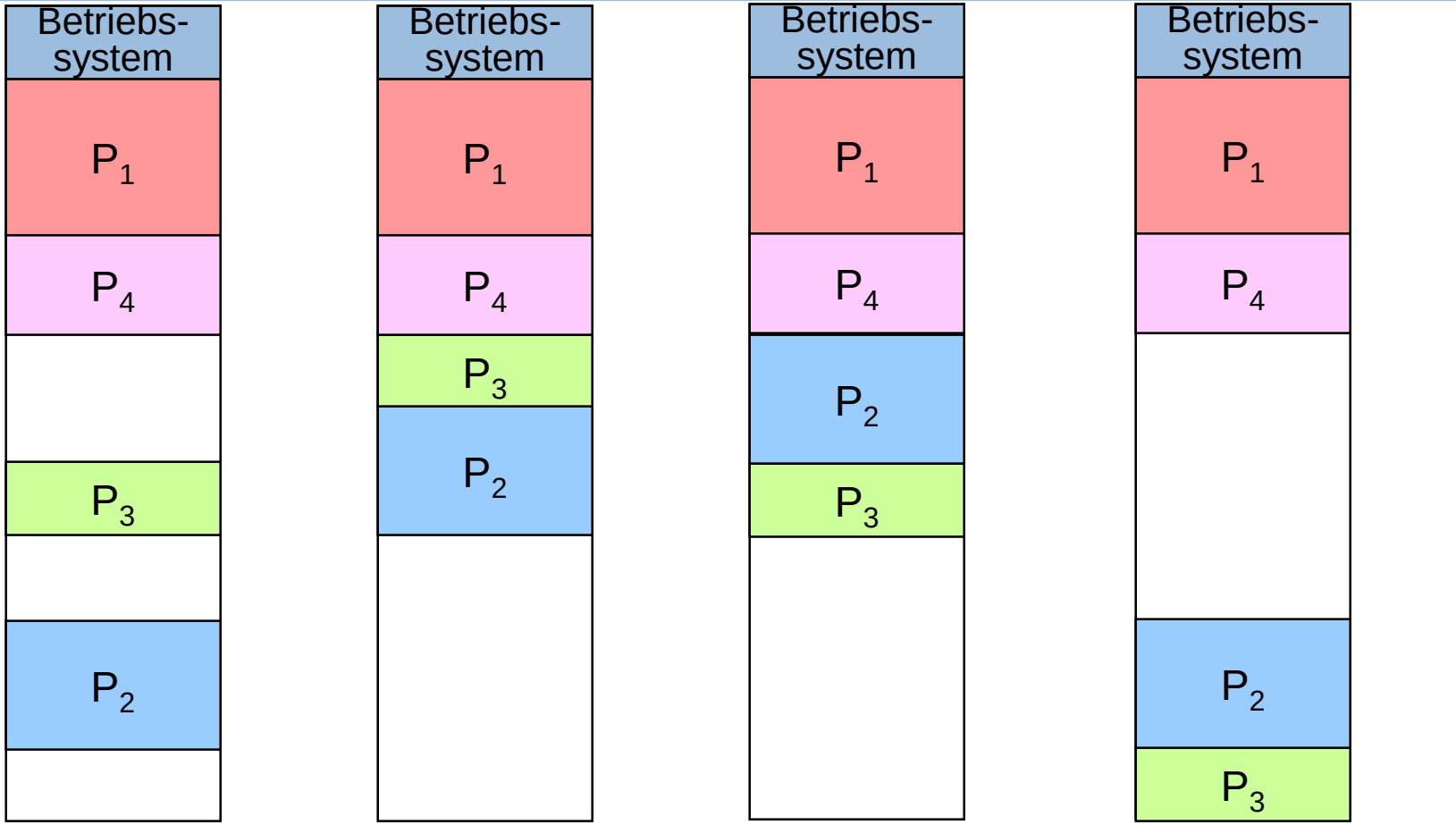
X: -Z Prozess X gibt Speicher der Grösse Z frei.

2.7 Verschnitt

- extern
  - Angeforderter Speicherplatz existiert,
    aber nur in kleineren, nicht zusammenhängenden Löchern.
  - first-fit: statistisch: N/2 Verschnitt bei N allozierten Blöcken
- intern
  - Zur Vereinfachung der Algorithmen wird mehr Speicher alloziert als notwendig (z.B. Buddy).
  - Vermeiden kleiner Löcher zur Reduktion des Verwaltungsaufwands
- Vermeiden von externem Verschnitt
  - Compaction
  - Paging: nicht zusammenhängender physikalischer Adressraum



2.7.1 Compaction

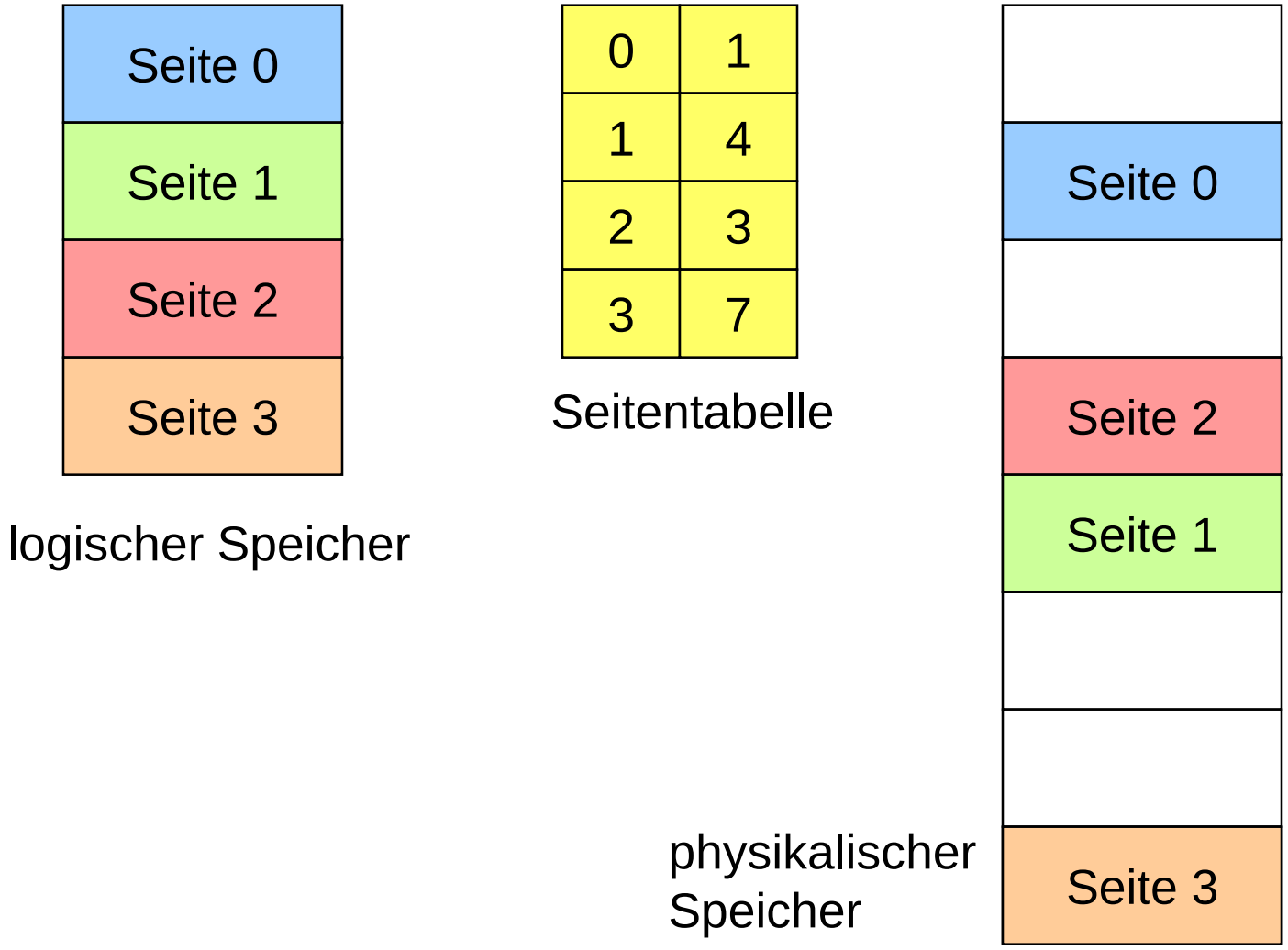


Ausgangs-situation | Verschieben von 2 Prozessen | Verschieben des grösseren Prozesses | Verschieben des kleineren Prozesses

3. Paging

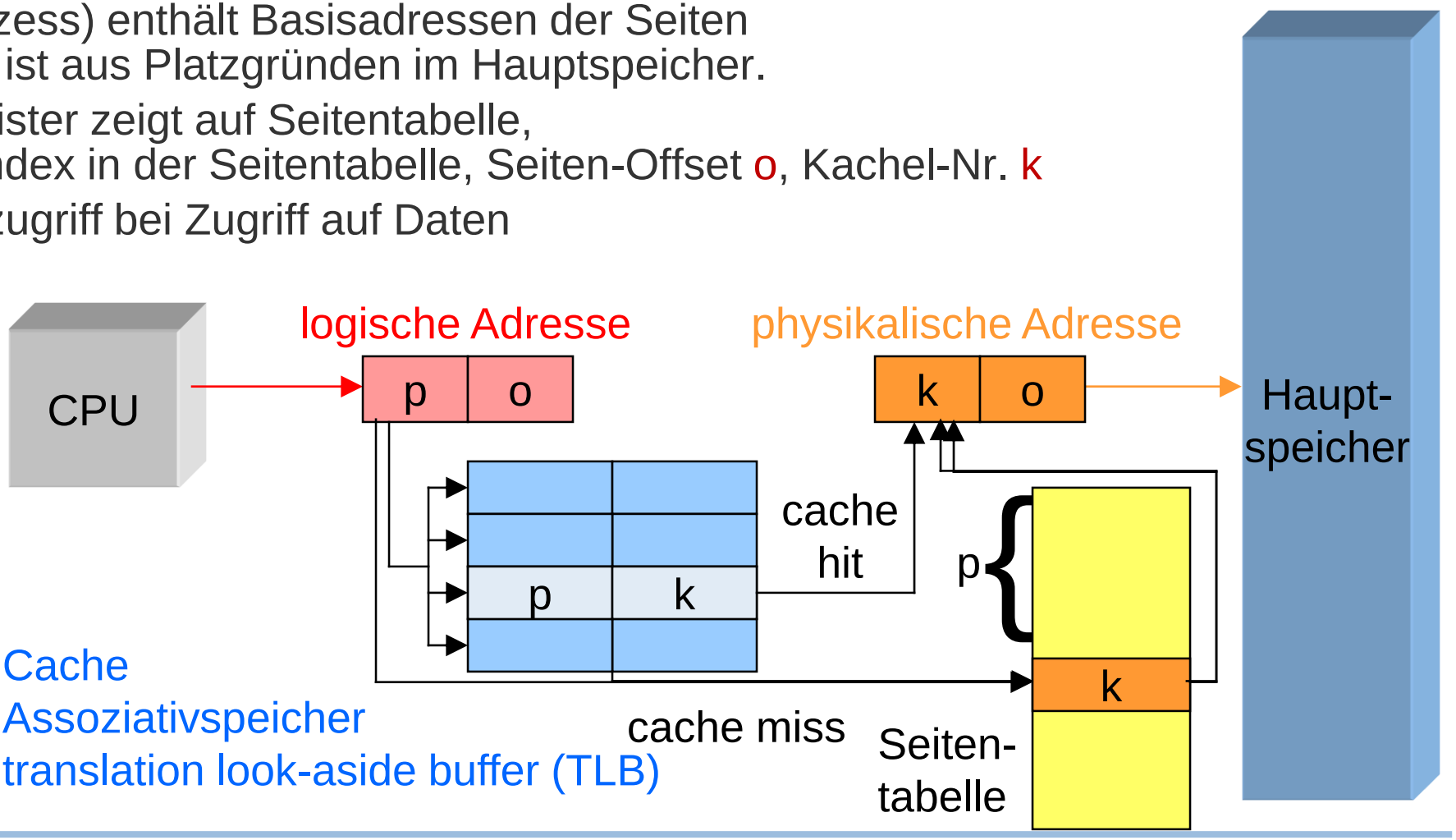
- nicht zusammenhängender physikalischer Adressraum
- Unterteilung
  - physikalischer Speicher in Kacheln (Frames, 2^x Bytes, z.B. 512 oder 8192 Bytes)
  - logischer Speicher in Seiten gleicher Grösse
- Programm mit Speicherbedarf von N Seiten benötigt N freie Kacheln.
- Zuordnung Seiten/Kacheln über Seitentabelle
- interner Verschnitt

3.1 Beispiel: Paging



3.2 Adressumsetzung

- Seitentabelle (pro Prozess) enthält Basisadressen der Seiten im Hauptspeicher und ist aus Platzgründen im Hauptspeicher.
- Page Table Base Register zeigt auf Seitentabelle, Seitennummer p als Index in der Seitentabelle, Seiten-Offset o, Kachel-Nr. k
- zusätzlicher Speicherzugriff bei Zugriff auf Daten



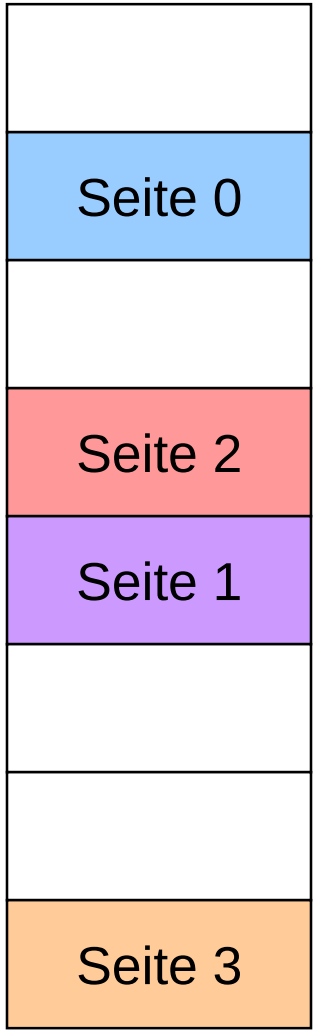
bei cache miss seitentabelle

3.3 Speicherzugriffsschutz

logischer Speicher

0	1	v
1	4	v
2	3	v
3	7	v
4	0	i
5	0	i
…	…	…

Seitentabelle

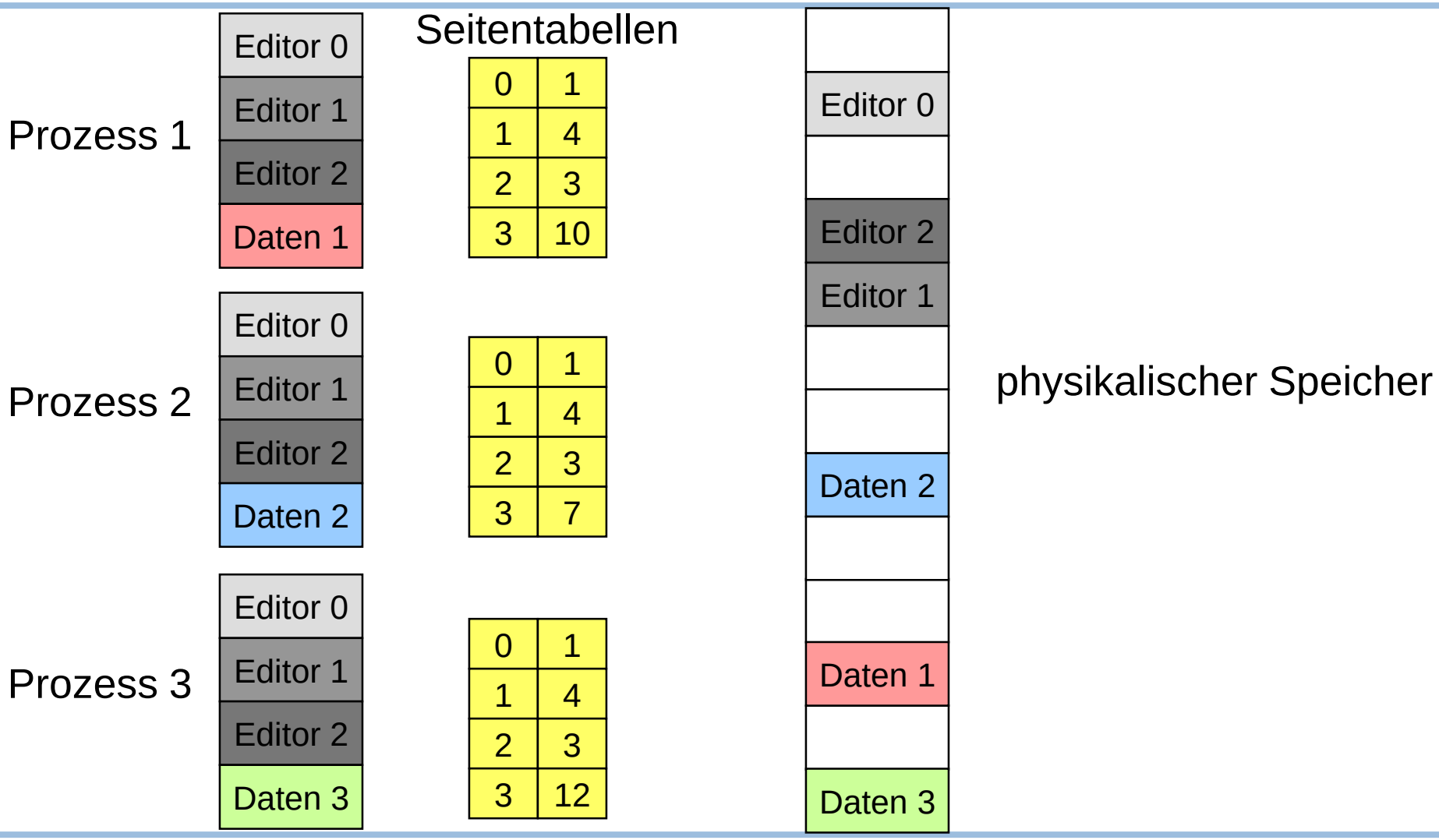


physikalischer Speicher

valid: Seite ist innerhalb des logischen Adressraums
invalid: Seite ist ausserhalb des logischen Adressraums

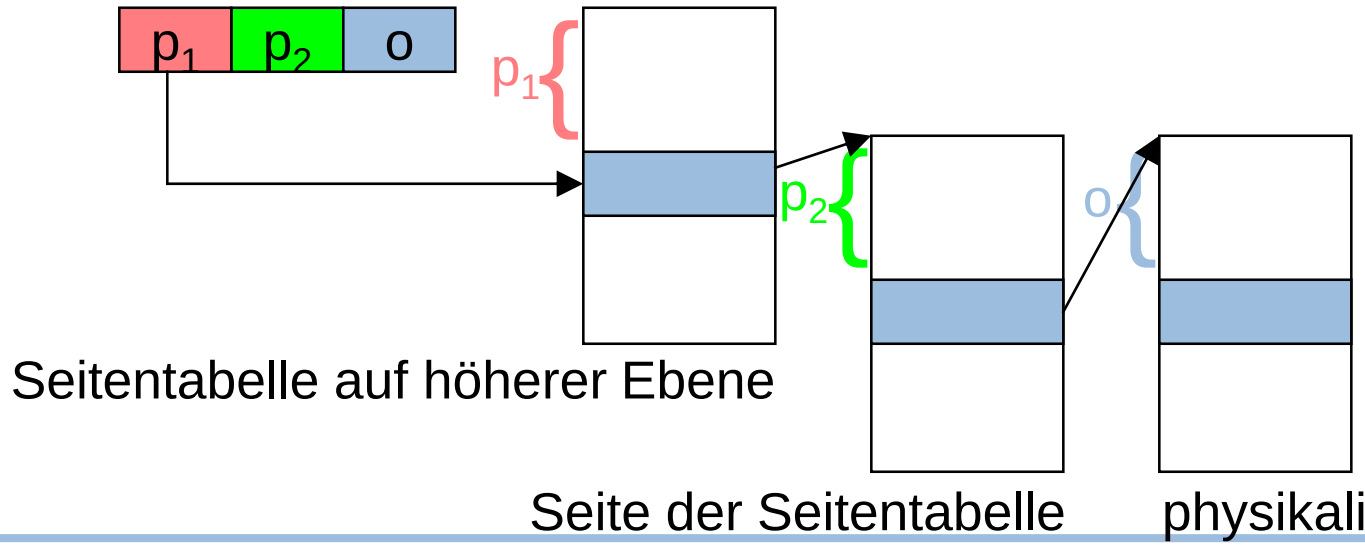
alternativ:
page table length register (PTLR)

3.4 Geteilte Seiten

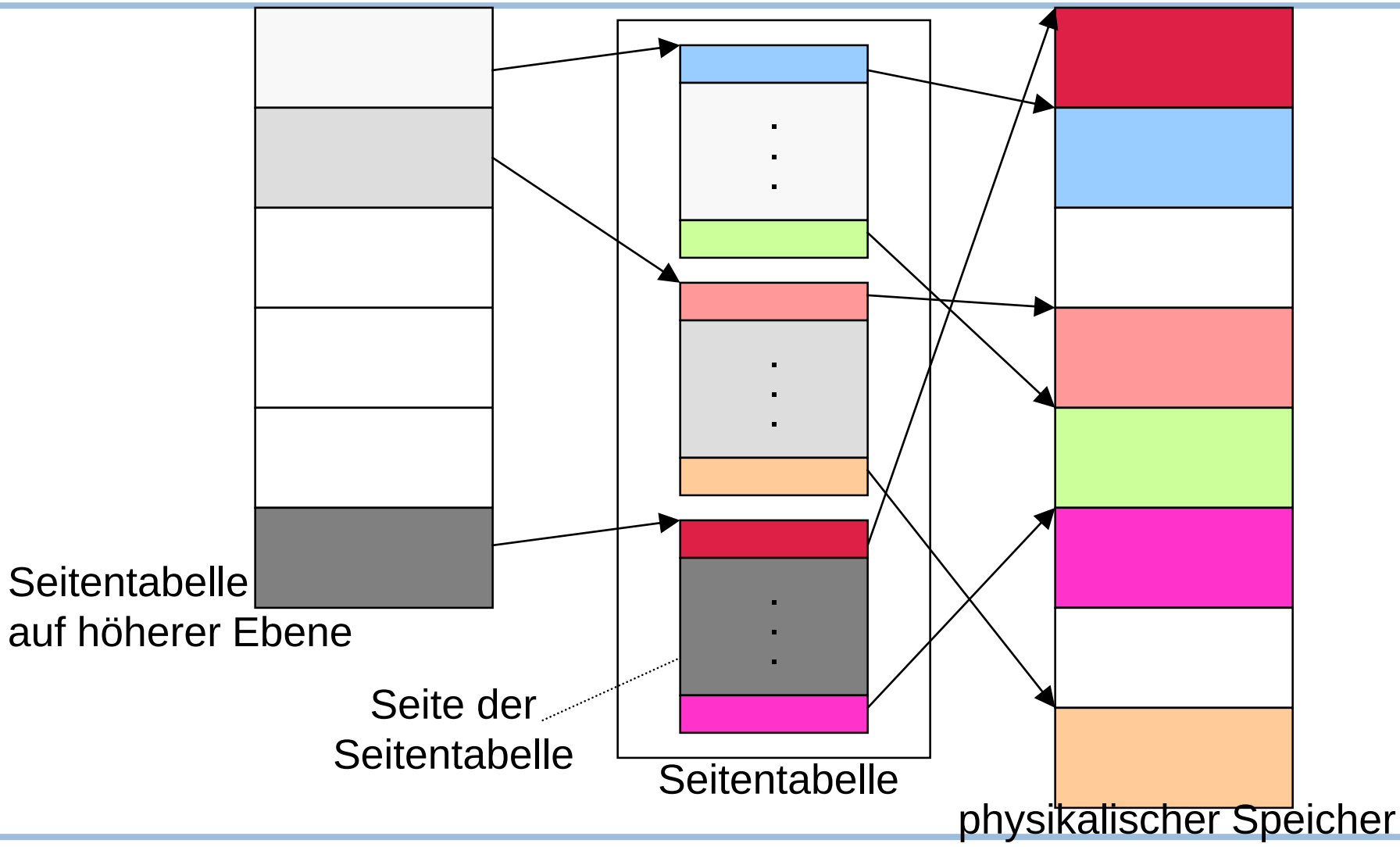


3.5 Multilevel Paging

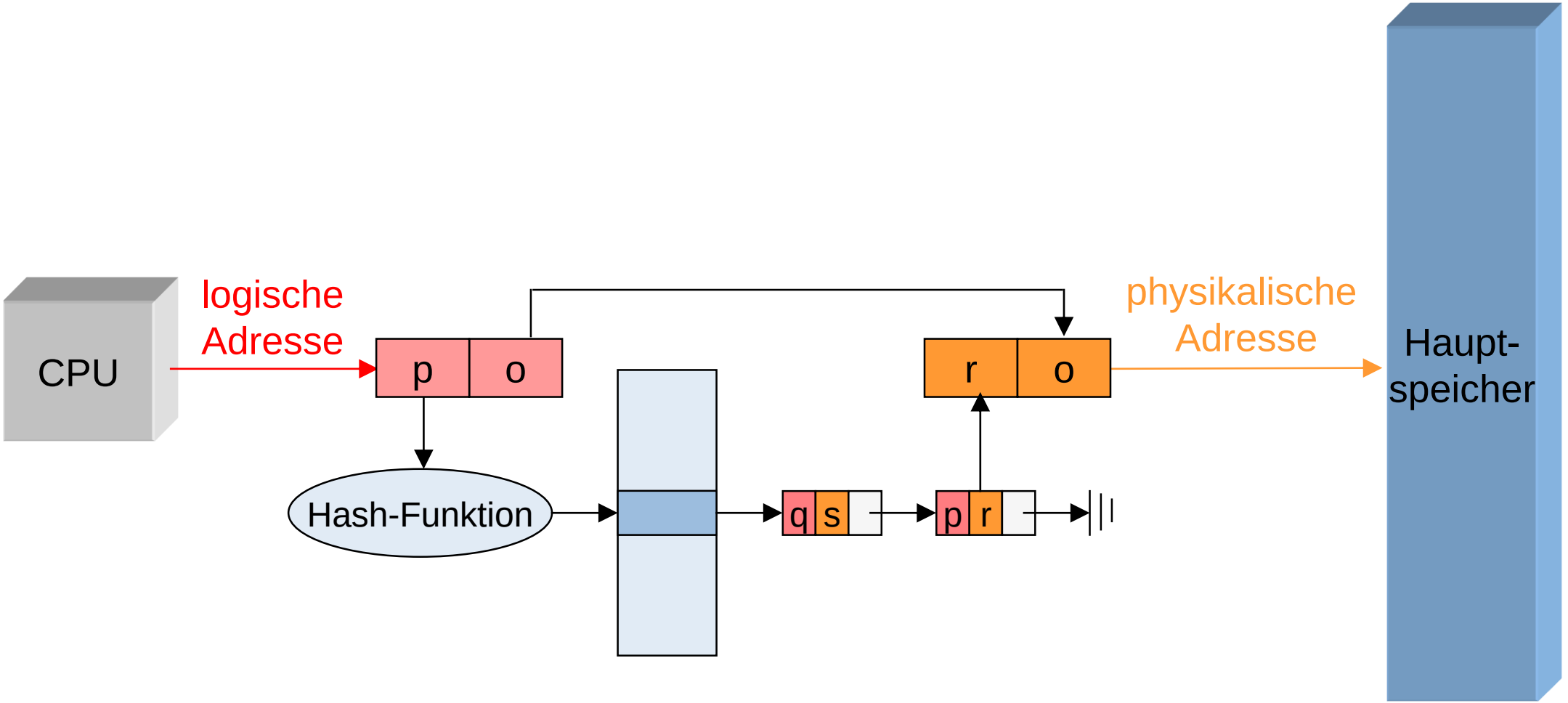
- Problem:
  - Adressraum 2^{32} Bytes, Seitengrösse 4 KB (2^{12}) $\rightarrow$ 2^{20} Seiten, 4 Bytes pro Eintrag: 4 MB für Seitentabelle
- Lösung:
  - Aufteilen der Seitentabelle in kleinere Einheiten
  - Laden der benötigten Einheiten in den Speicher



3.5.1 Two-Level Paging

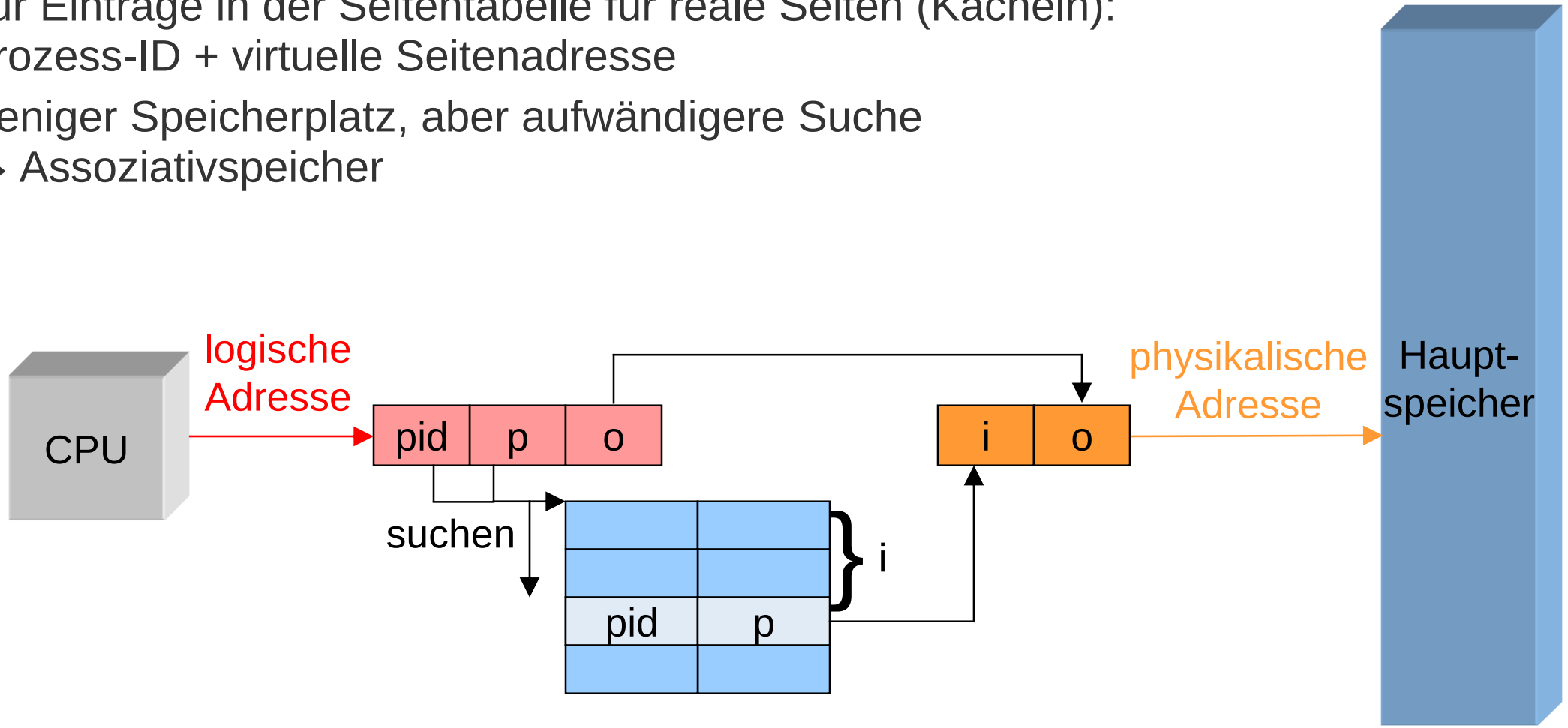


3.6 Seitentabelle mit Hashes



3.7 Invertierte Seitentabellen

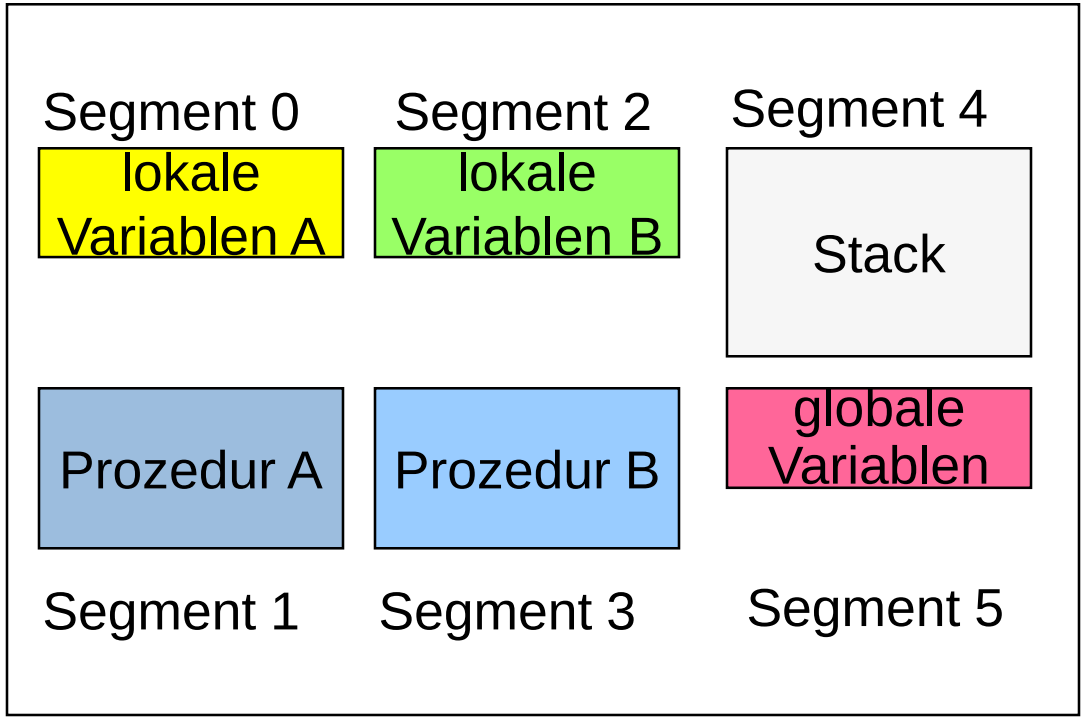
- nur Einträge in der Seitentabelle für reale Seiten (Kacheln): Prozess-ID + virtuelle Seitenadresse
- weniger Speicherplatz, aber aufwändigere Suche
  → Assoziativspeicher



4. Segmentierung

- Compiler erzeugt verschiedene Segmente variabler Länge für ein Programm, z.B.
  - Code
  - globale Variablen
  - Heap
  - Stacks für Threads
  - Standard-Bibliothek
- Segmentierung unterstützt die Benutzersicht auf den Speicher, d.h. logischer Adressraum als Sammlung von Segmenten
- logische Adresse: <Segmentnummer, Offset>

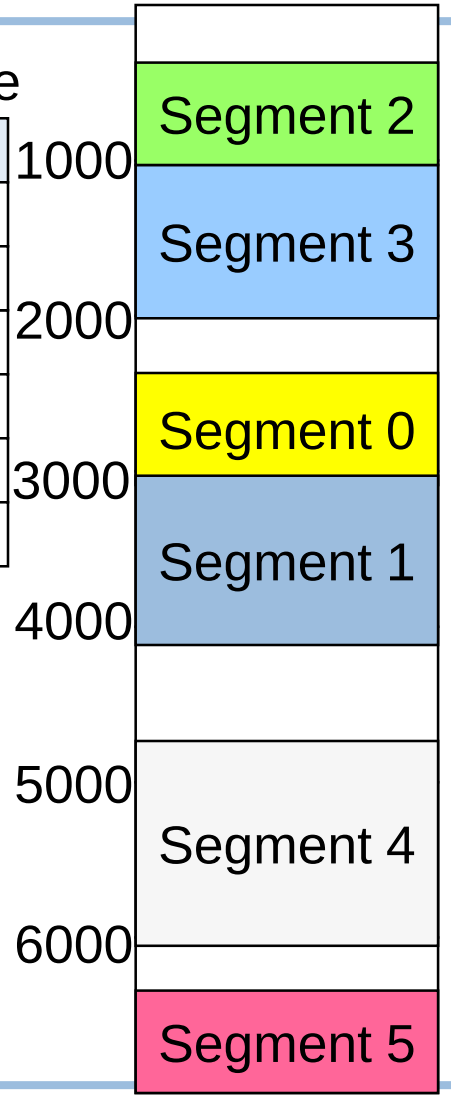
4.1 Beispiel: Segmentierung



logischer Adressraum eines Programms

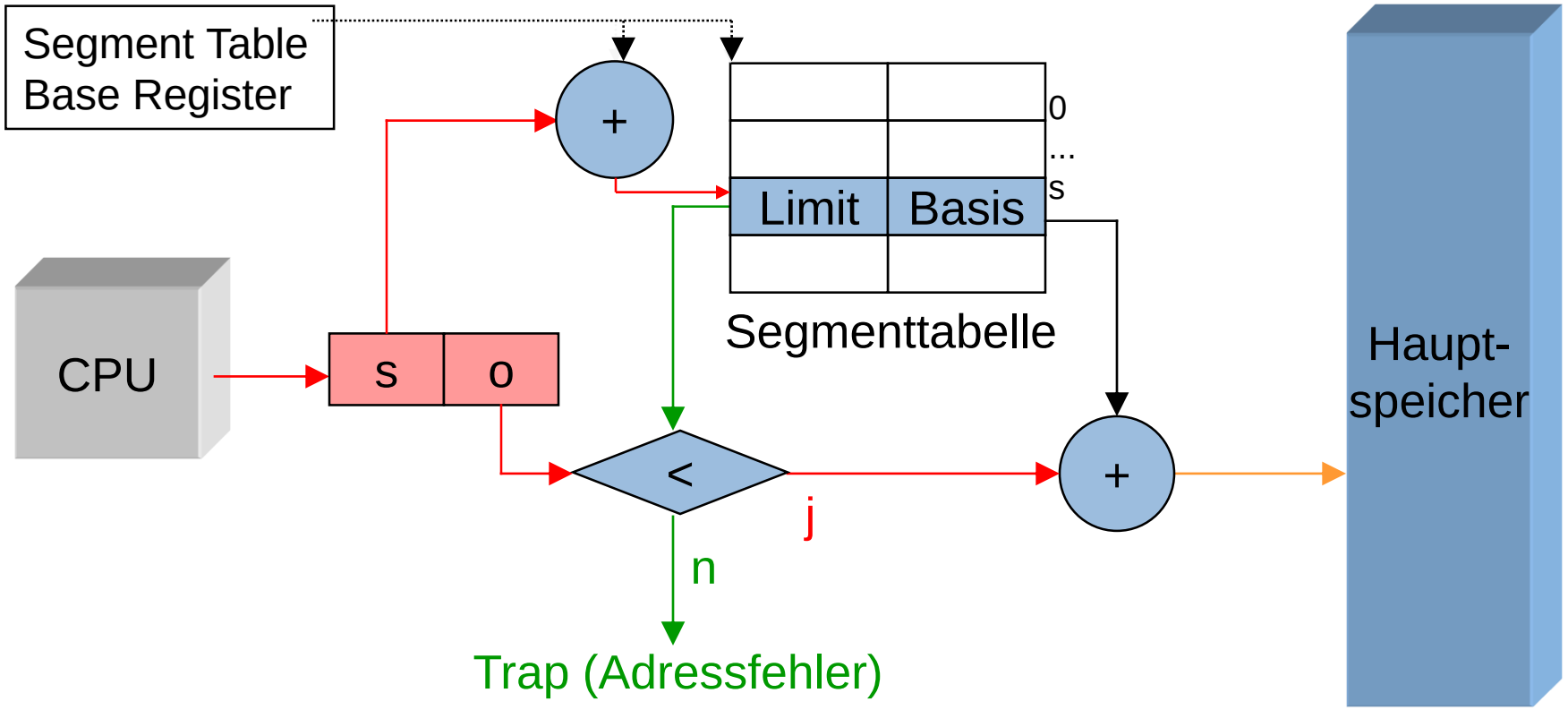
Segmenttabelle

Limit	Basis
700	2300
1000	3000
700	300
1000	1000
1300	4700
700	6300



4.2 Adressumsetzung

- Abbildung der zweidimensionalen logischen Adresse (Segmentnummer und Offset) auf eindimensionale physikalische Adresse
- Einträge der Segmenttabelle: Segmentbasis und Segmentlänge (Limit)



4.3 Segmentierung mit Paging