

E. Virtueller Speicher

E.1 Zielsetzungen

- **Scheinbare Hauptspeichervergrößerung:**
 - ◆ Gegenüber einem Programm mehr Speicher vorspiegeln, als physisch installiert ist,
 - ◆ Speicherteile automatisch ein- und auslagern (mit Betriebssystemunterstützung),
 - ◆ Logische Sicht der Programme vs. physikalische Organisation.

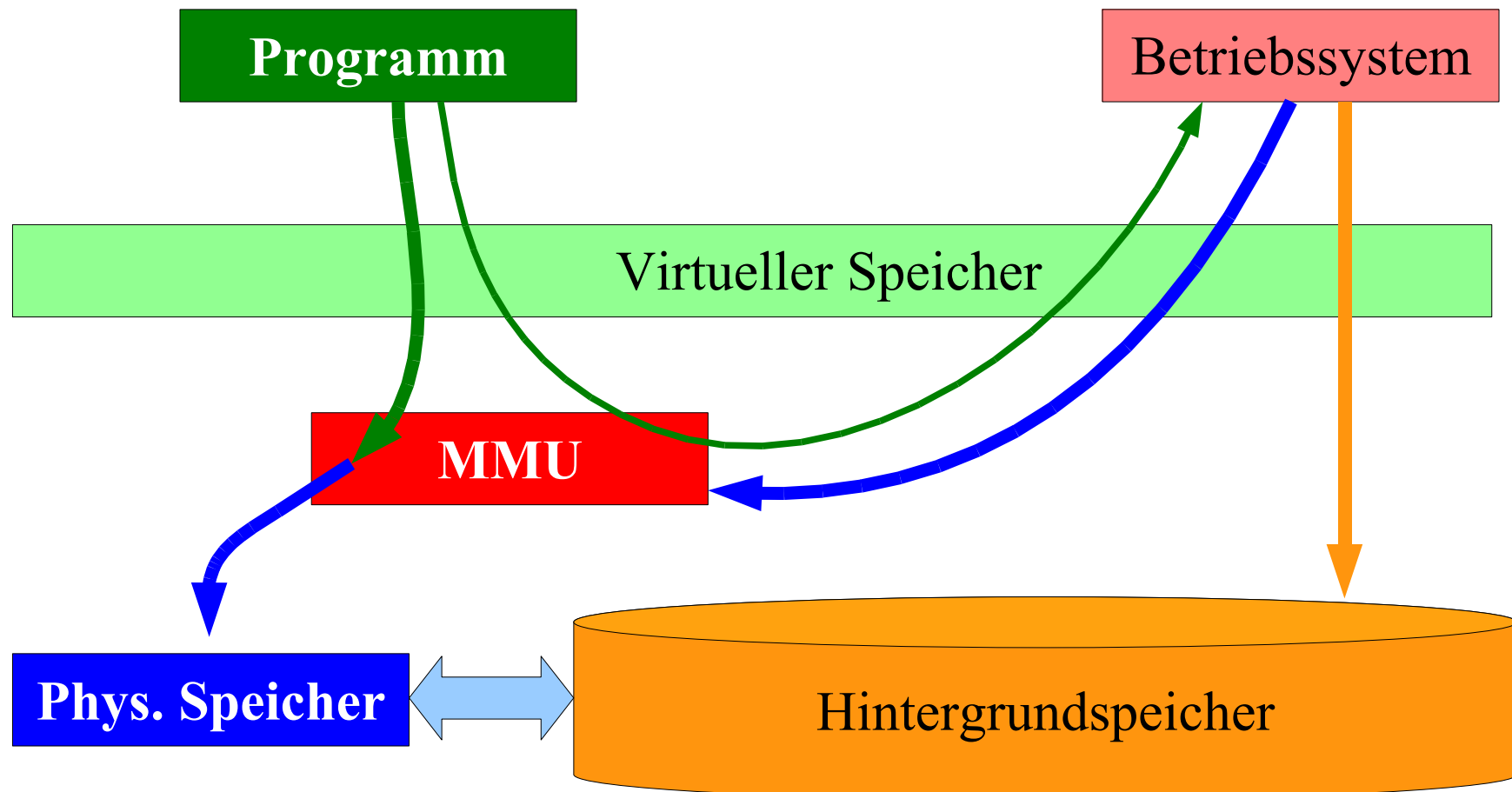
- **Speicherschutz:**
 - ◆ Schutz gegen ungewollte/absichtliche Störungen von Programmen & Betriebssystem,
 - ◆ Zugriff auf Speicher fremder Programme nur mit besonderer Genehmigung,
 - ◆ Unterstützung durch Hardware (z. B. Segmente),
 - ◆ typischere Sprachen von Vorteil (z.B. Java).

- **Punktuell gemeinsame Nutzung von Speicherbereichen:**
 - ◆ Kooperation auf gemeinsamen Daten (Shared Memory),
 - ◆ vermeiden von Code-Redundanz (mit Zugriffskontrolle).

- **Interne und externe Fragmentierung minimieren.**

E.2 Grundprinzip

- Dem Programm wird ein größerer Hauptspeicher vorgespiegelt, als physikalisch vorhanden ist:
 - ◆ Das Programm arbeitet nur mit logischen bzw. virtuellen Adressen,
 - ◆ Hardware, Compiler und OS übersetzen diese in physische Adressen:



■ MMU - Memory Management Unit im Prozessor:

- ◆ HW-Einrichtung zur Übersetzung von virtuelle Adressen in physische,
- ◆ erzeugt nötigenfalls Seiten- und Segment-Fehler,
- ◆ verwendet Segment- und/oder Seitentabelle.

■ Virtueller/Logischer Speicher:

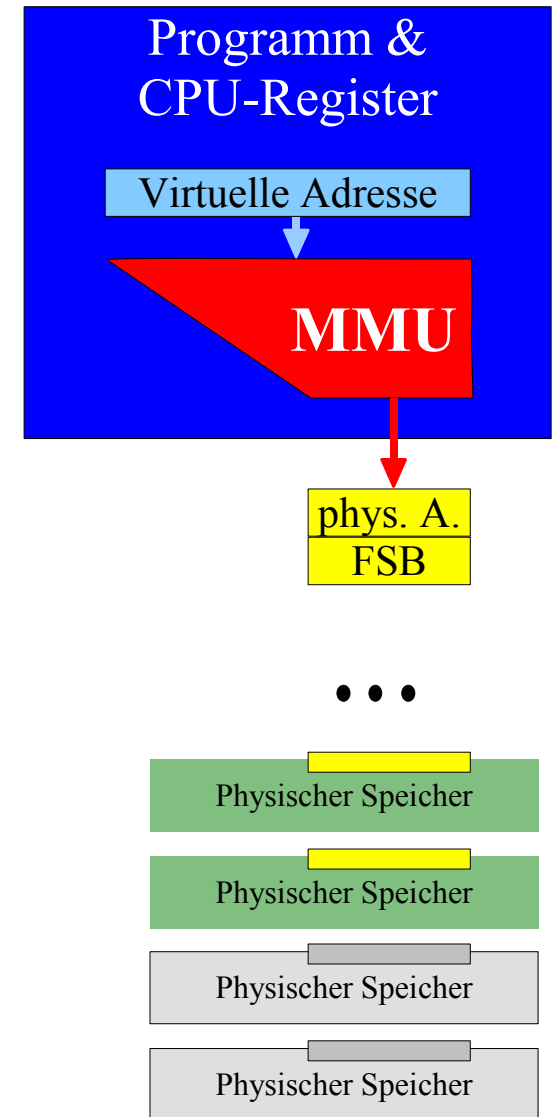
- ◆ maximale Größe ist abhängig von der MMU-Hardware,
- ◆ 32/45-Bit bei IA32 CPUs und 48/64-Bit bei AMD64,
- ◆ variabel große Segmente = Segmentierung,
- ◆ oder gleich große Seiten (pages) = Paging,
- ◆ oder auch beides kombiniert (z.B. Intel).

■ Physischer Adressraum:

- ◆ Physisch installierter Speicher evtl. kleiner als Adressraum,
- ◆ Die Größe ergibt sich aus der Bitbreite der Speicheradressierung.
- ◆ Intel IA32 Rechnersysteme bieten 32-Bit (4 GB, evtl. 36 Bit),
- ◆ AMD64/Intel-64 CPUs bieten 40Bit und mehr.

■ Hintergrundspeicher:

- ◆ Dient dem Ein- und Auslagern inaktiver Segmente/Seiten.
- ◆ Langsamer, preisgünstiger, größer (Festplatte, Netz ...),
- ◆ Steuerung über Betriebssystem und Interrupts.



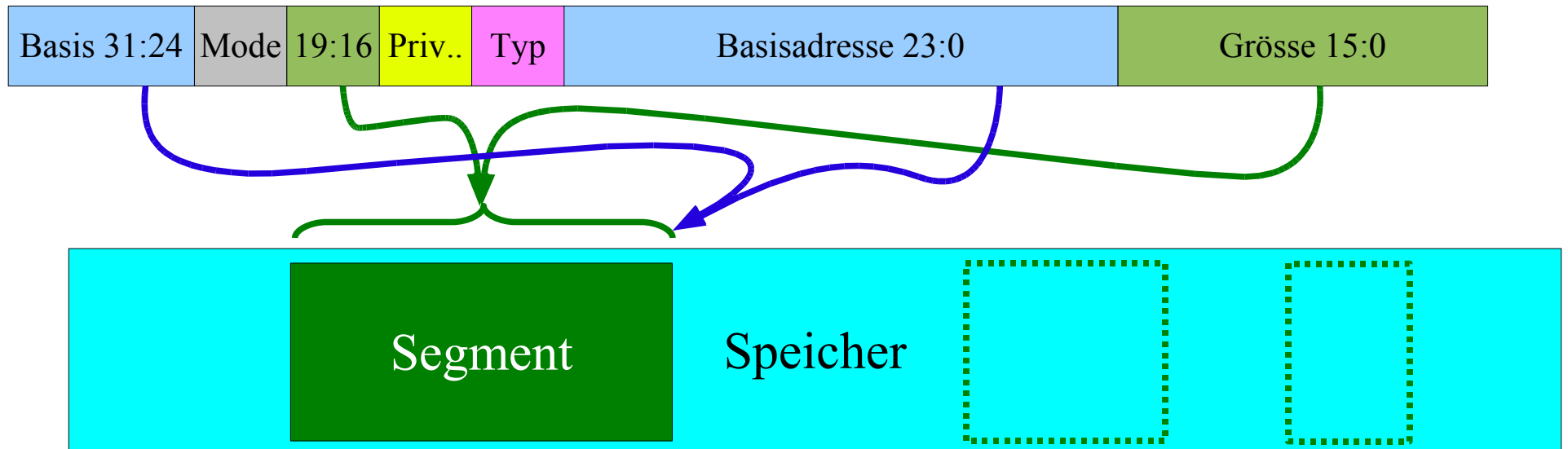
E.3 Segmentierung im Intel 32-Bit Modus

E.3.1 Unterteilung in variabel große Segmente (kein Paging):

■ Segmentdeskriptoren:

- ◆ historisch gewachsen aus der Absicht eine High-Level-Language Maschine zu schaffen,
- ◆ enthalten Ausführungssemantik (Task-Switch, Real/Protected Mode, Call, Interrupt ...),
- ◆ enthalten Präsenzbit, Zugriffsprivilegien und Typ (Code, Data, Stack, ...),
- ◆ enthalten Basisadresse und Grösse,
- ◆ Gedanke einer Capability,
- ◆ Viele Formate.

■ Format eines Datensegment-Deskriptors für Intel i86 CPUs (8 Bytes):



E.3.2 Segmentdeskriptor-Tabellen:

■ Indizierung in die Tabellen über **Segmentselektoren**:

- ◆ die alten Segmentregister wirken als Segment-Selektoren,
- ◆ 13 Bit als Tabellenindex (0..8195),
- ◆ 1 Bit für globale/lokale Tabelle,
- ◆ 2 Bit für Privilegierung,
- ◆ liegen in DS, SS, CS ...

■ Globale Deskriptortabelle:

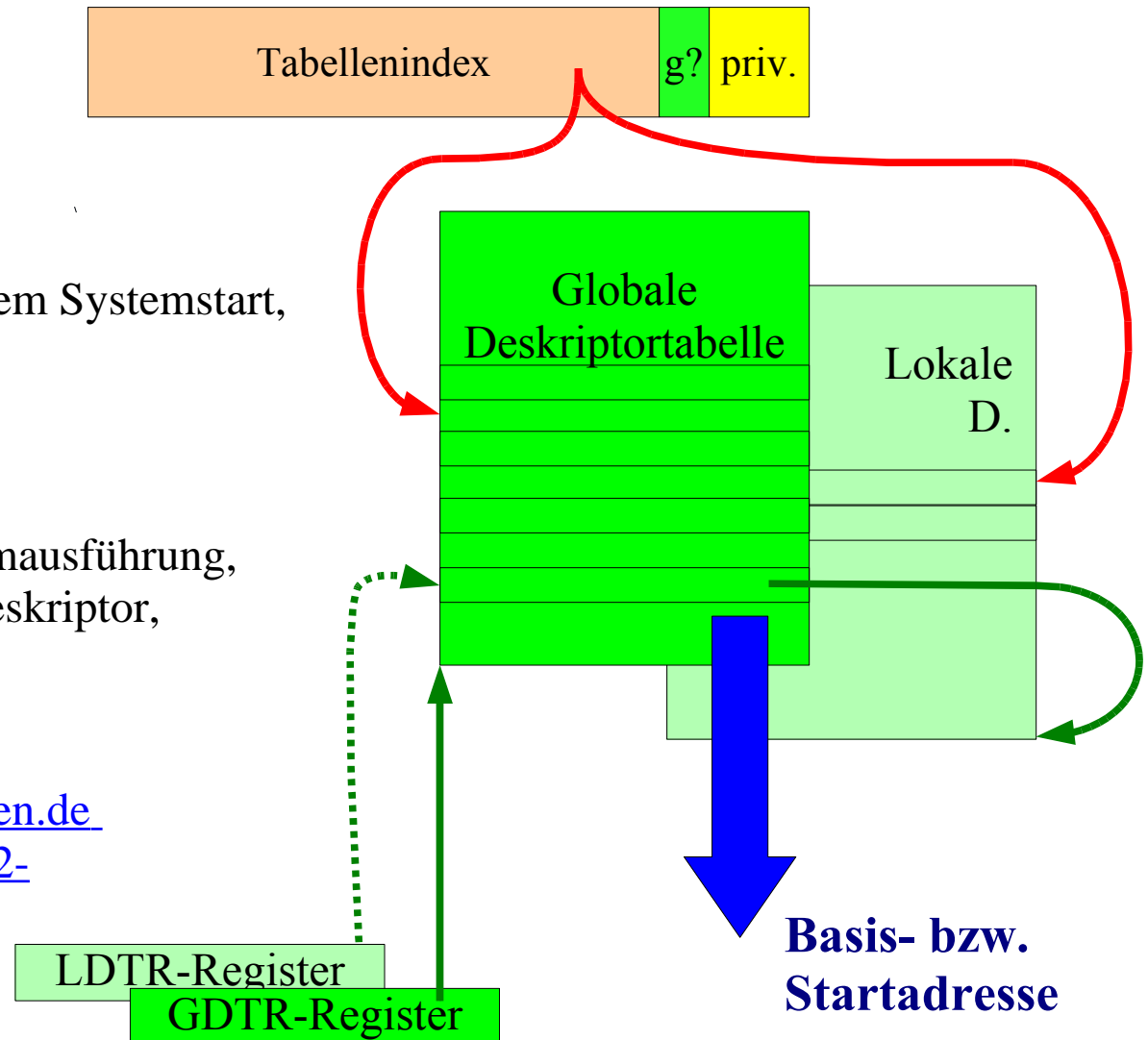
- ◆ typischerweise unverändert nach dem Systemstart,
- ◆ verankert über GDTR-Register,
- ◆ maximal 8196 Einträge.

■ Lokale Deskriptortabelle:

- ◆ veränderbar während der Programmausführung,
- ◆ referenziert über einen globalen Deskriptor,
- ◆ maximal 8196 Einträge.

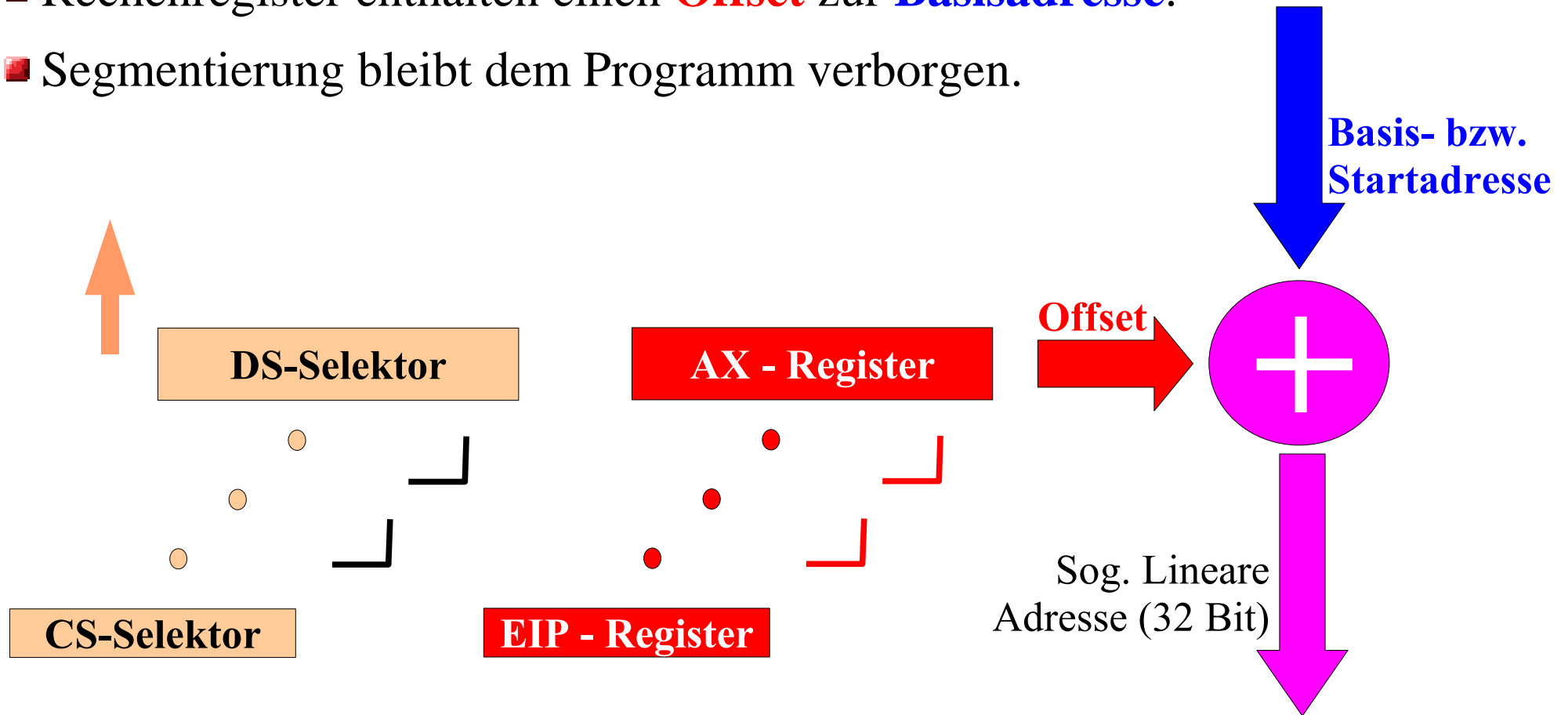
■ URL z.B.:

- ◆ http://www4.informatik.uni-erlangen.de/Lehre/WS06/V_BS/fohlen/07-IA32-2x2.pdf



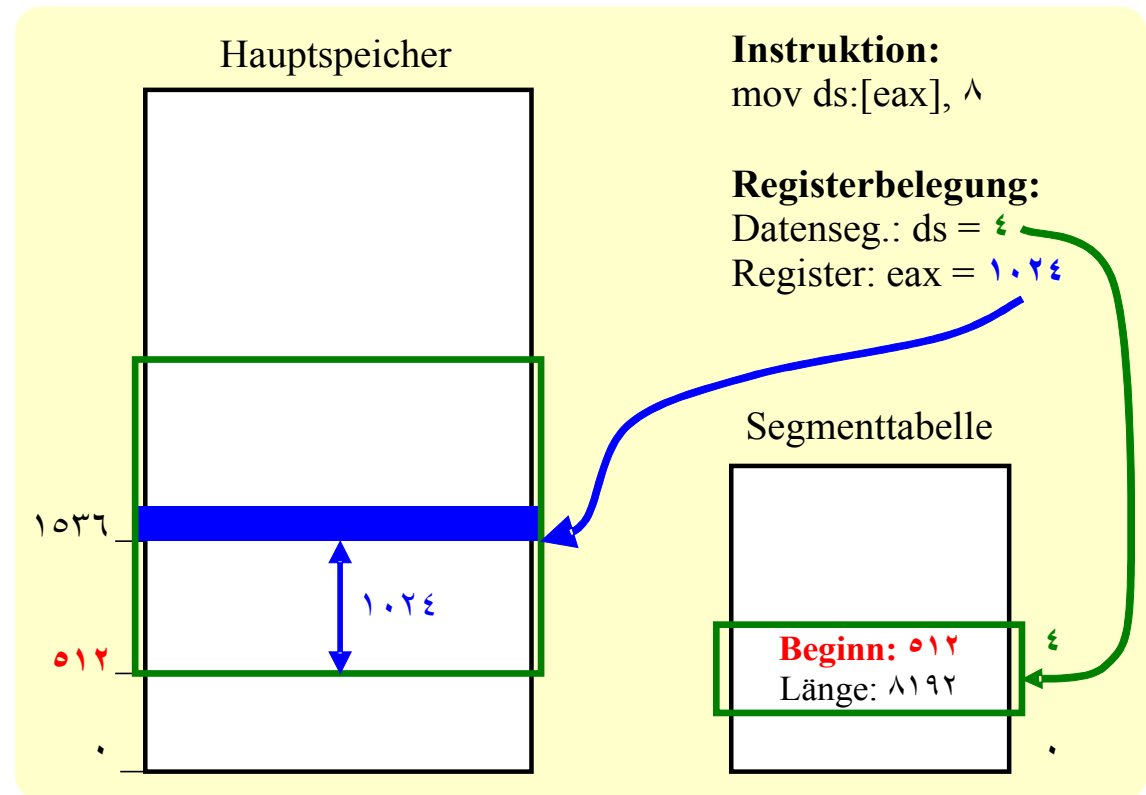
E.3.3 Adressierung mit Basisadresse und Offset:

- Segmentregister werden durch das BS beim Programmwechsel gesetzt.
- Segmentregister enthalten einen Index in die Segmenttabelle.
- Rechenregister enthalten einen **Offset** zur **Basisadresse**.
- Segmentierung bleibt dem Programm verborgen.



E.3.4 Beispiel: Datenadressierung

- Schreibe den Wert ,8' im aktuellen Daten-segment an die Offsetadresse 1024:
 - ◆ auch der Anwender darf ein Segmentregister laden,
 - ◆ evtl. Schutzverletzung.



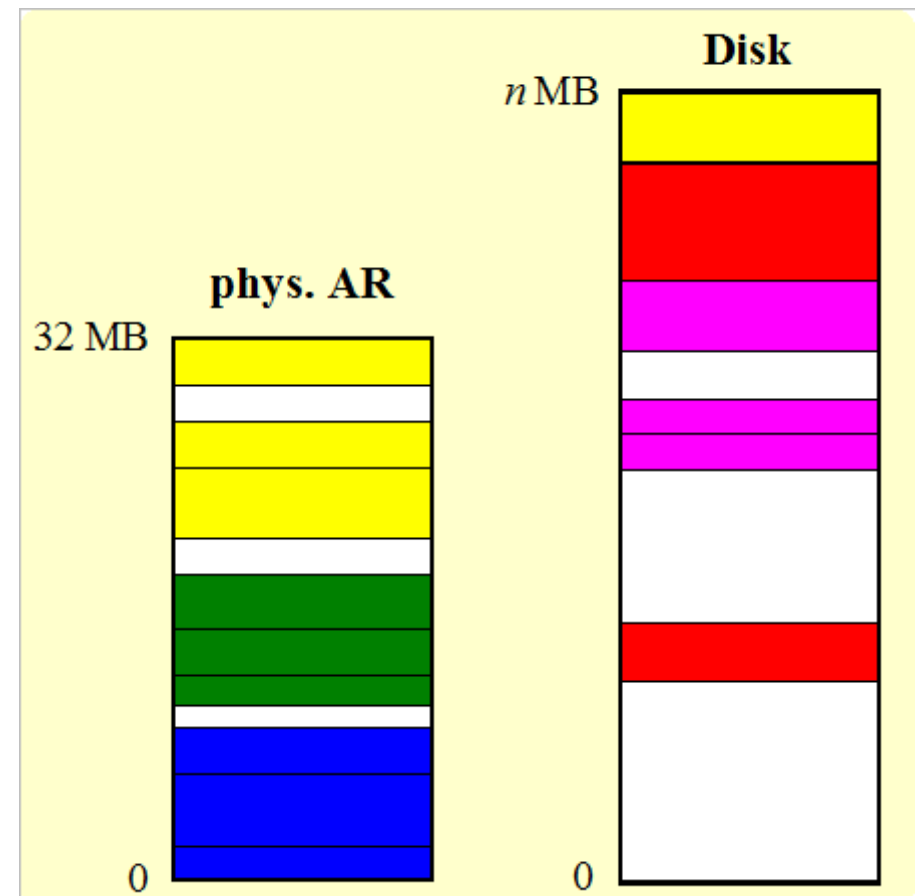
- Compiler kann die Segmentstruktur berücksichtigen:
 - ◆ Sprungtabellen enthalten evt. außer der Offsetadresse, auch die Segment-Nummer.
 - ◆ Beim Prozeduraufruf über Segmentgrenzen hinweg, wird auch das CS Segmentregister geladen und evtl. das Präsenzbit geprüft,
 - ◆ Die Segmentadresse trägt der Lader ein.

E.3.5 Virtualisierung mithilfe von Segmenten:

- Nicht benötigte Segmente können auf Disk ausgelagert werden.
- Präsenz-Bit im Segmentdeskriptor:
 - ◆ ist zurückgesetzt, falls das Segment nicht im Hauptspeicher eingelagert ist,
 - ◆ Beim Zugriff auf ein ausgelagertes Segment tritt ein Segmentfehler (Segment Fault) auf,
 - ◆ Vor der Einlagerung muss evt. ein Segment verdrängt werden, da der HS erschöpft ist,
 - ◆ die Verlagerung von Segmenten ist Aufgabe des Betriebssystems.
- Begrenzung der Programmgröße:
 - ◆ Programme bestehen normalerweise aus vielen Segmenten.
 - ◆ Die maximale Segmentgröße ist beschränkt durch die Größe des installierten Hauptspeichers.
 - ◆ Der verfügbare logische/virtuelle Adressraum bestehend aus Segmentnummer und Offset ist damit größer als der Hauptspeicher.
- Theoretisch möglicher Virtueller AR:
 - ◆ ohne Austausch der lokalen Deskriptortabelle: $2^{13} * 2^{32} = 2^{45} \sim 1,604 * 10^{18}$
 - ◆ mit Austausch der lokalen Tabelle: $> 2^{13} * 2^{12} * 2^{32} = 2^{57} \dots$
- Auslagerungsunterstützung:
 - ◆ IA32 **Accessed-Bit** (gesetzt beim Laden des Selektors),
 - ◆ aber kein **Modified-Bit** wie bei Seiten.

E.3.1 Mehrprogrammbetrieb mit segmentiertem virtuellen AR:

- Segmente können an beliebiger Stelle wieder eingelagert werden.
- immer ganze Segmente werden ein- und ausgelagert.
- auch hier externe Fragmentierung.
- Kompaktierung auf Disk ist teuer.

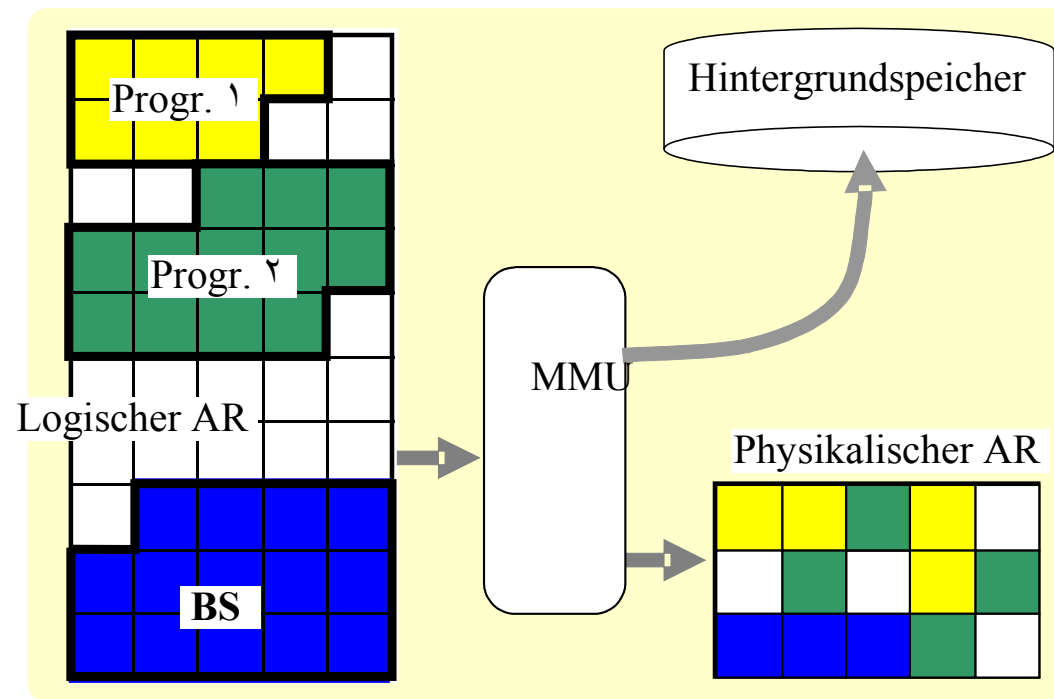


E.4 Paging

- Unterteilt den **logischen Adressraum** in gleich große **Seiten (pages)**:
 - ◆ IA32 unterstützt 4 GB logischen Adressraum, der z.B. in 4 KB Seiten unterteilt wird.
 - ◆ Damit werden Programme in viele kleine auslagerbare Einheiten unterteilt,
 - ◆ **Seitenfehler (Page Fault)** beim Zugriff auf eine ausgelagerte Seite,
 - ◆ mehr Flexibilität als Segmente.
- **Pages und Page-Frames:**
 - ◆ Seitenrahmen sind physisch vorhanden und nehmen jeweils eine logische Seite auf,
 - ◆ typischerweise sind alle Seitenrahmen gleich gross,
 - ◆ also keine externe Fragmentierung.
- Grundlage für Paging ist **Lokalität**:
 - ◆ Programm greift in einem kleinen Zeitraum Δt nur auf einen Teil seines AR zu.
 - ◆ Nach dem Zugriff auf Adresse a ist ein Zugriff in der Nähe von a wahrscheinlich.
 - ◆ Gründe: sequentielle Ausführung, Schleifen, ... => nicht benötigte Teile auslagern.
- **Entwurfsfragen:**
 - ◆ Seitengröße: interner Verschnitt vs. Verwaltung.
 - ◆ Ersetzungsregel: Welche Seite wird ersetzt?
 - ◆ Laderegeln: Wann wird eine Seite geladen?

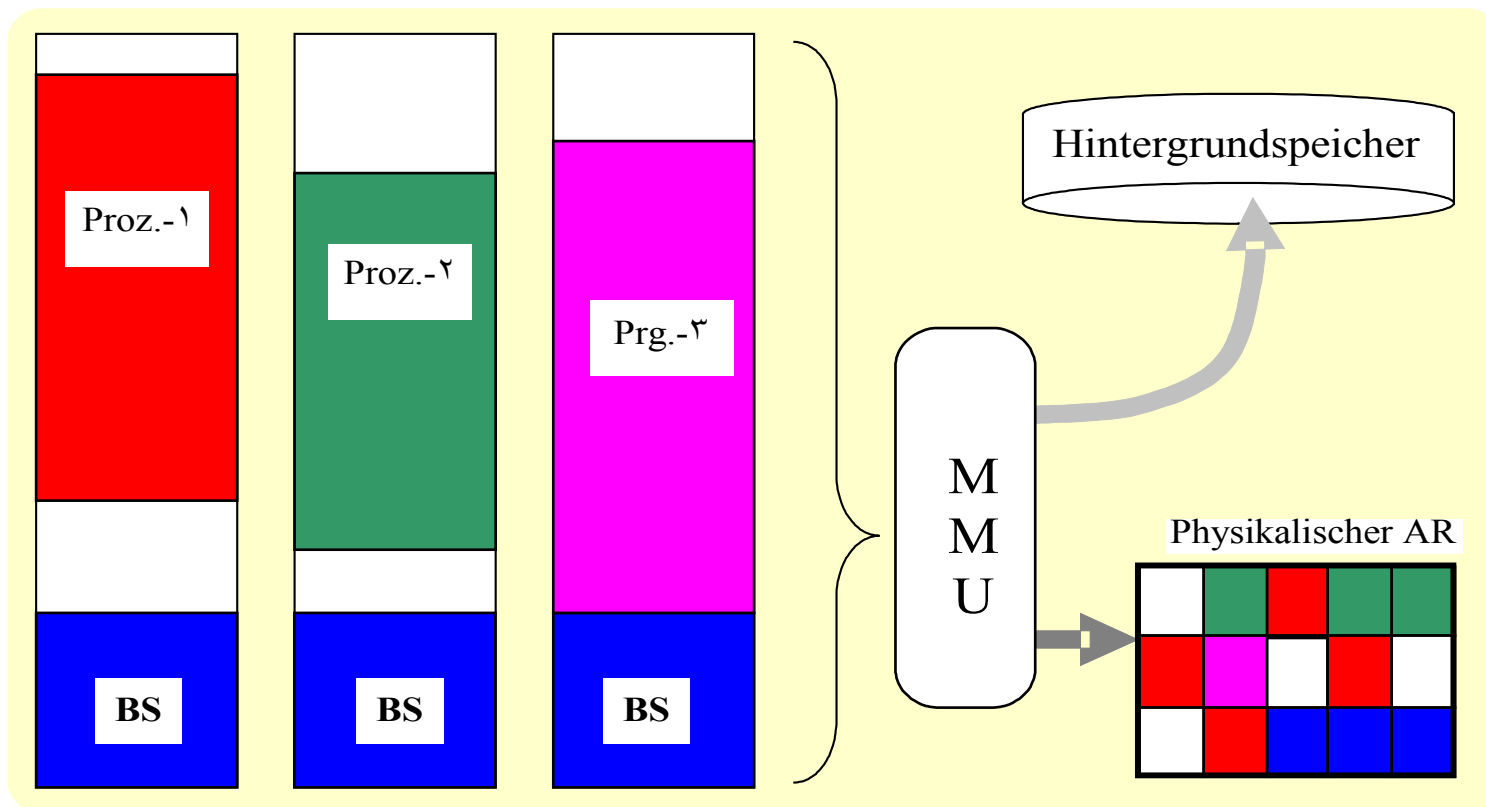
E.4.1 Einfach virtualisierter Speicher:

- Gesamtsumme des Speicherverbrauchs aller Programme ist beschränkt,
- Keine Umschaltung der Adressübersetzung erforderlich (TLB!),
- Kein impliziter Zugriffsschutz durch die Virtualisierung,
- Koexistenz der Programme trotzdem möglich,
- Nur eine Übersetzungstabelle.



E.4.1 Mehrfach virtualisierter Speicher:

- Pro Prozess besteht ein separater virtueller Adressraum.
- Adressübersetzungstabelle beim Prozesswechsel umschalten.
- Teile des Betriebssystemadressraumes auch für die Anwendungsprogramme zugreifbar.



E.4.2 Adressübersetzung

■ Einstufiger Übersetzungsvorgang

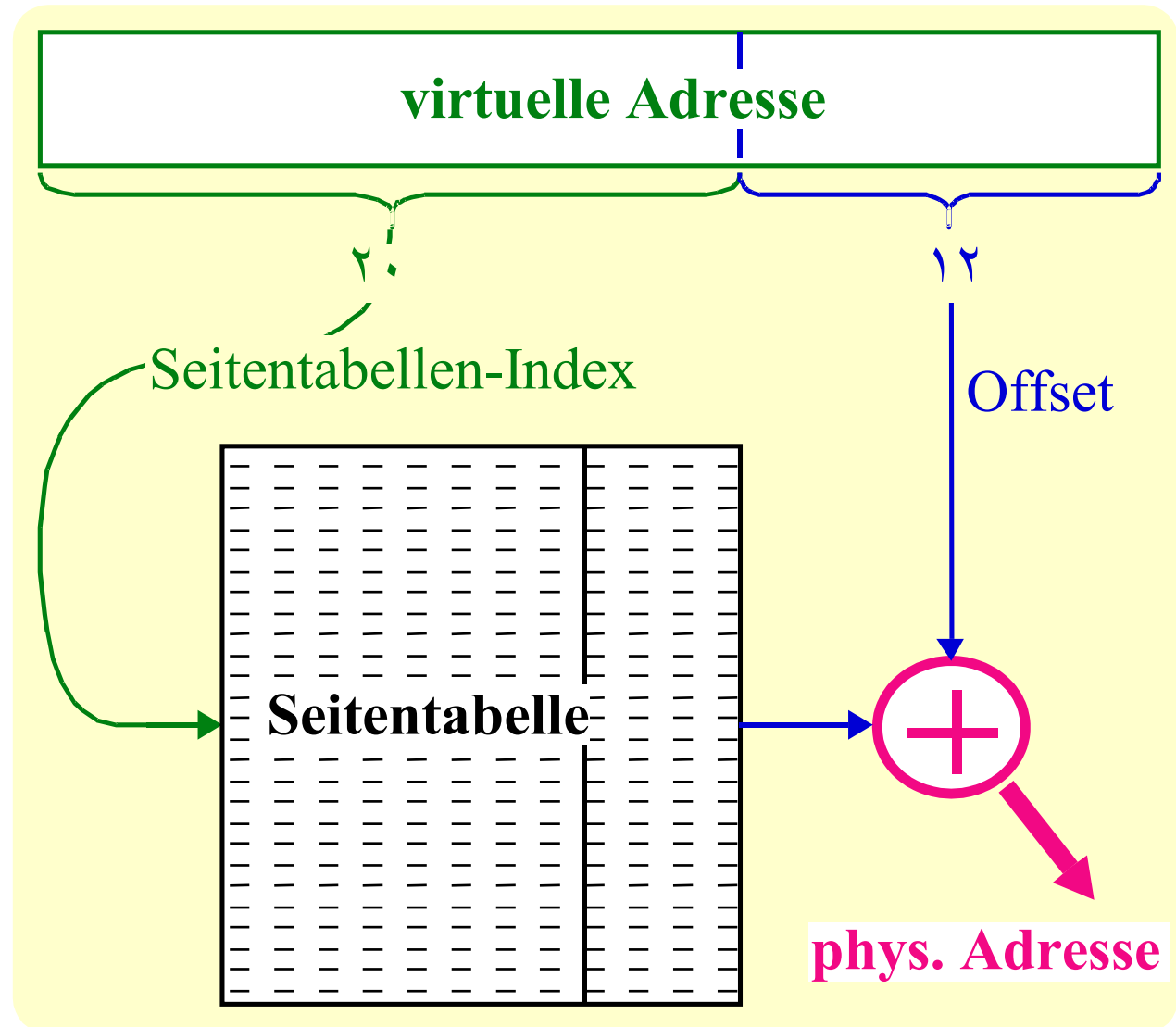
- ◆ z.B. 20 Bit virtuelle Adr.,
- ◆ 4 KByte Seiten.

■ Vorteil:

- ◆ Aufeinanderfolgende Seiten müssen nicht unbedingt auf fortlaufende Kacheln abgebildet werden.

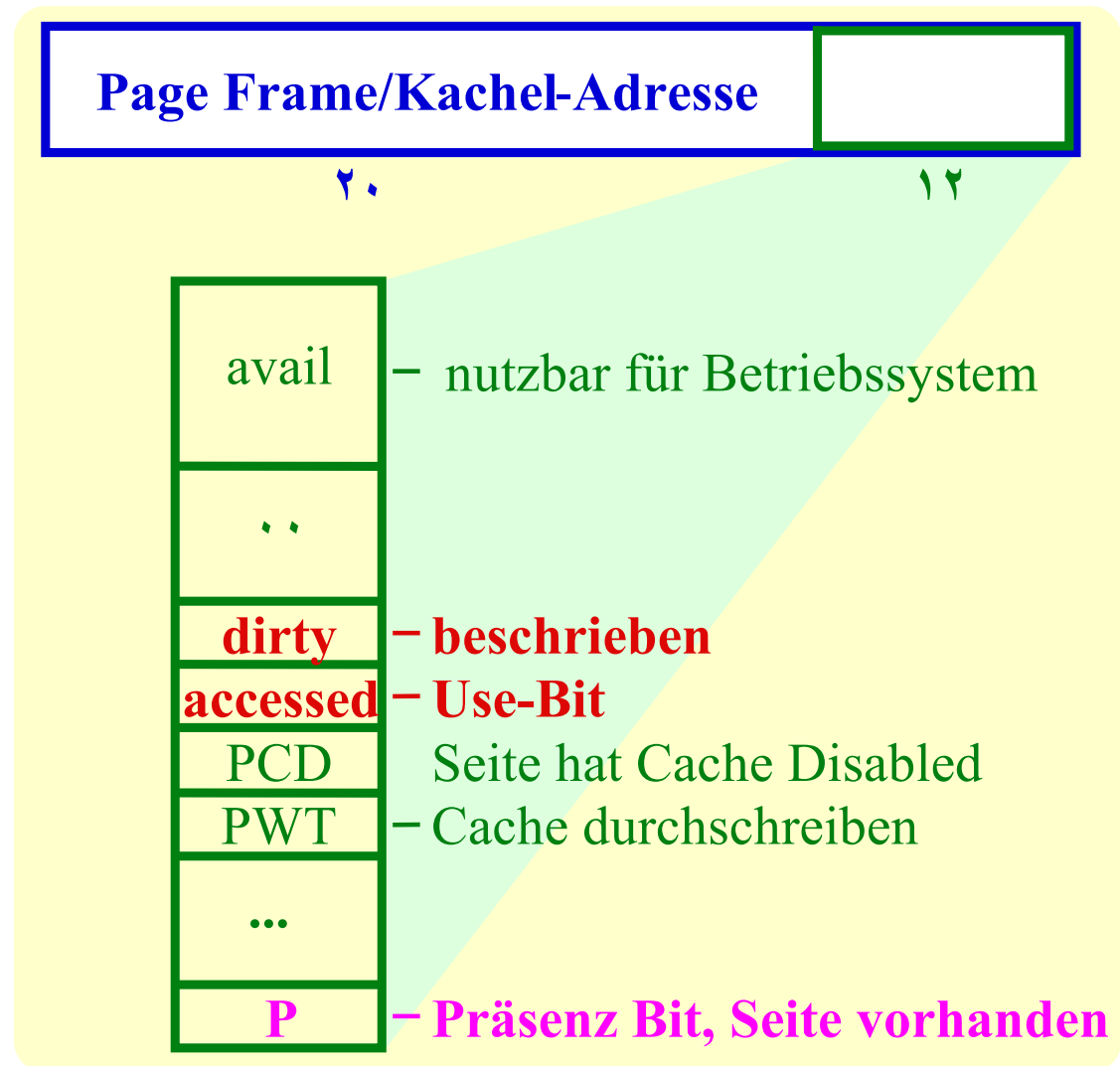
■ Nachteil:

- ◆ Seitentabelle konsumiert 4 MB physischen Speicher und kann nicht ausgelagert werden.



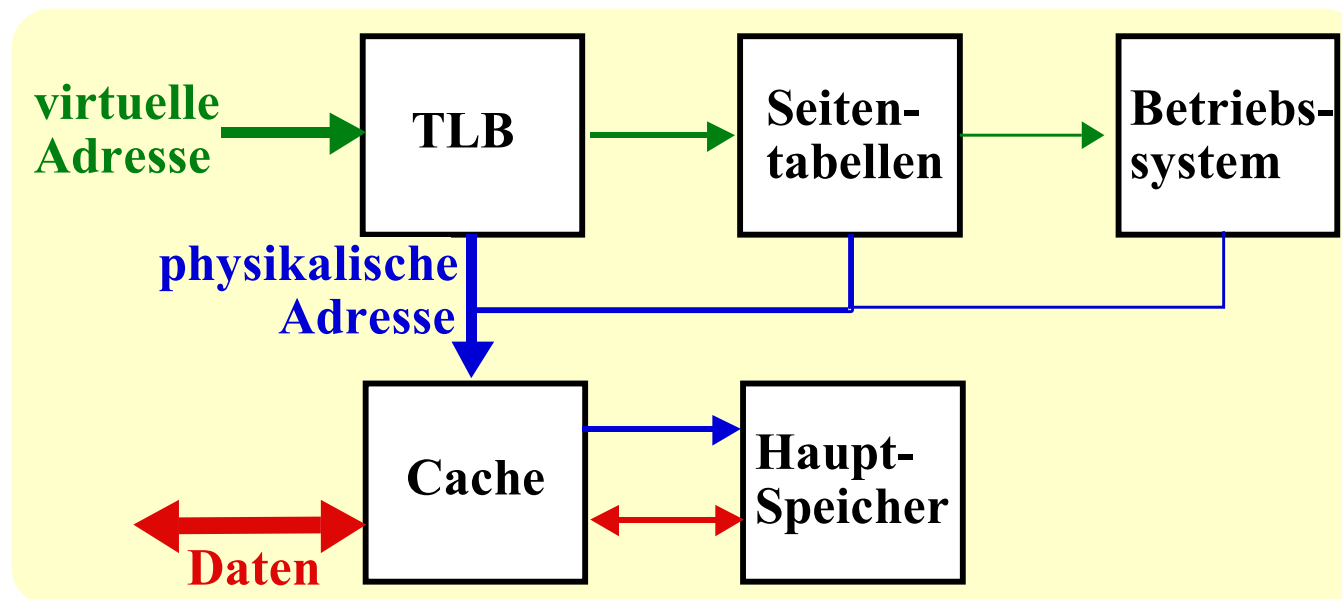
E.4.4 Typisches Format für die Seitentabelle (IA32):

- **Kacheladresse** nur gültig, wenn die Seite im Speicher vorhanden ist.
- Andernfalls können diese 20 Bits zum Auffinden der Seite auf Disk verwendet werden.
- Bits: **dirty & accessed** von MMU gesetzt.
- **Präsenz-Bit** durch BS verwaltet.



E.4.5 Übersetzungspuffer (TLB)

- TLB = "Translation Lookaside Buffer".
- TLB puffert früher übersetzte virtuelle Adressen und deren phys. Adr.
 - ◆ progressiv mehr Einträge: P4 (64), IA64 (128), Core Duo (~500).
 - ◆ Hohe Trefferrate (hit ratio) wichtig.
- Falls kein Treffer im TLB erfolgt, wird hardwaremäßig auf die Seitentabellen im Hauptspeicher oder im Cache zugegriffen (teuer).



■ L2-Cache benützt nur physikalische Adressen:

- ◆ unsichtbar für die Software,
- ◆ physikalischer Adresse ist eindeutig,
- ◆ aber evt. mehrere virtuelle Adressen für eine physikalische Adresse.

■ TLB Programmierung:

- ◆ aufwendiges TLB-Management bei aktuellen Mehrkernprozessoren,
- ◆ zugehörigen Eintrag entfernen beim Auslagern einer Seite,
- ◆ TLBs komplett löschen bei Adressraumwechsel,
- ◆ Eintrag evt. fixieren, z.B. für Kernel.

■ L1-Cache arbeitet evtl. auf Basis der logischen Adressen:

- ◆ besondere HW erkennt mehrwertige Abbildung ...

E.4.7 Beispiel: Seitentabelle im IA64 - Itanium

- **Parallele Suche in größerem TLB (z.B. 128 Einträge bei IA64).**
 - ◆ TLB wird im Gegensatz zu IA32 und AMD64 in Software verwaltet
 - ◆ Also kein autom. Verdrängen und Befüllen durch MMU,

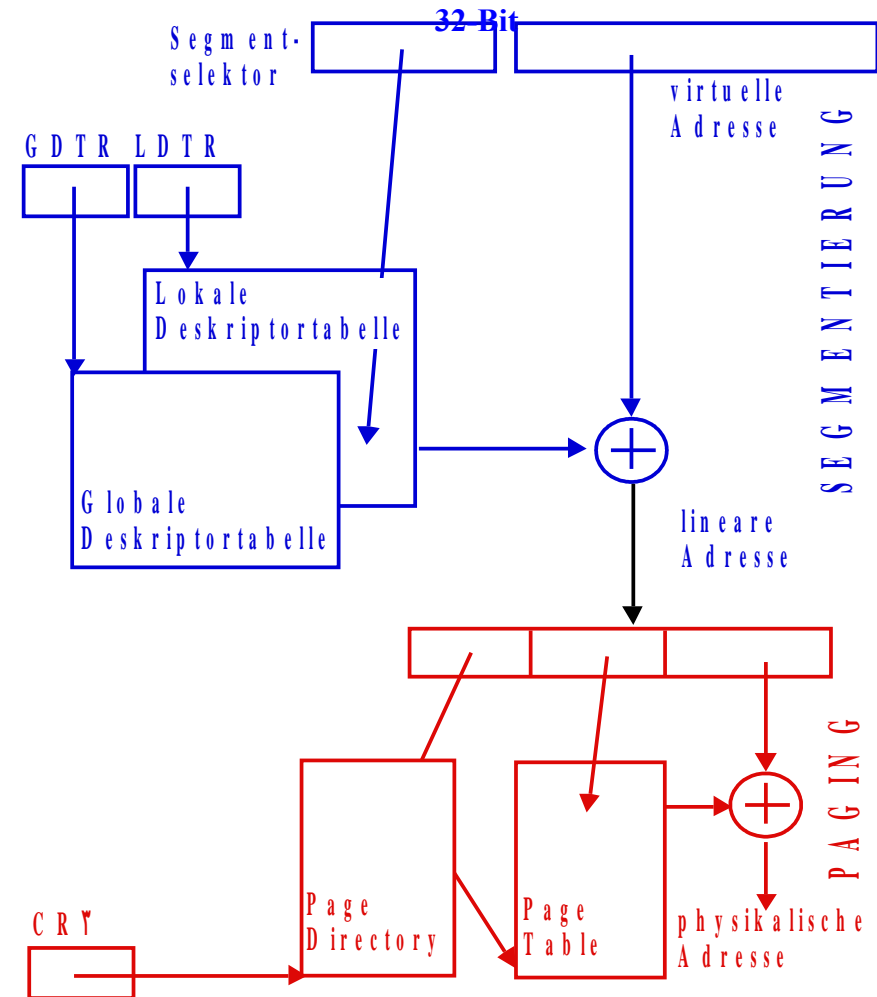
- **Bei Fehlanzeige im TLB:**
 - ◆ OS wird per Interrupt gerufen (TLB Miss Fault),
 - ◆ Über Hashtabelle (IA64) zum Seitentabellen-Eintrag,
 - ◆ Hashtabelle (vorgegebene Struktur) liegt im virtuellen Adressraum,
 - ◆ evt. (Translation) Fault beim Durchsuchen der Hash-Tabelle.

- **Bei Fehlanzeige in der Hashtabelle:**
 - ◆ Betriebssystem konsultiert invertierte Seitentabelle (lineare Suche),
 - ◆ spezialisierte Maschineninstruktionen vorhanden.

- **Bei Fehlanzeige in der invertierten Seitentabelle:**
 - ◆ Seite von Festplatte einlagern

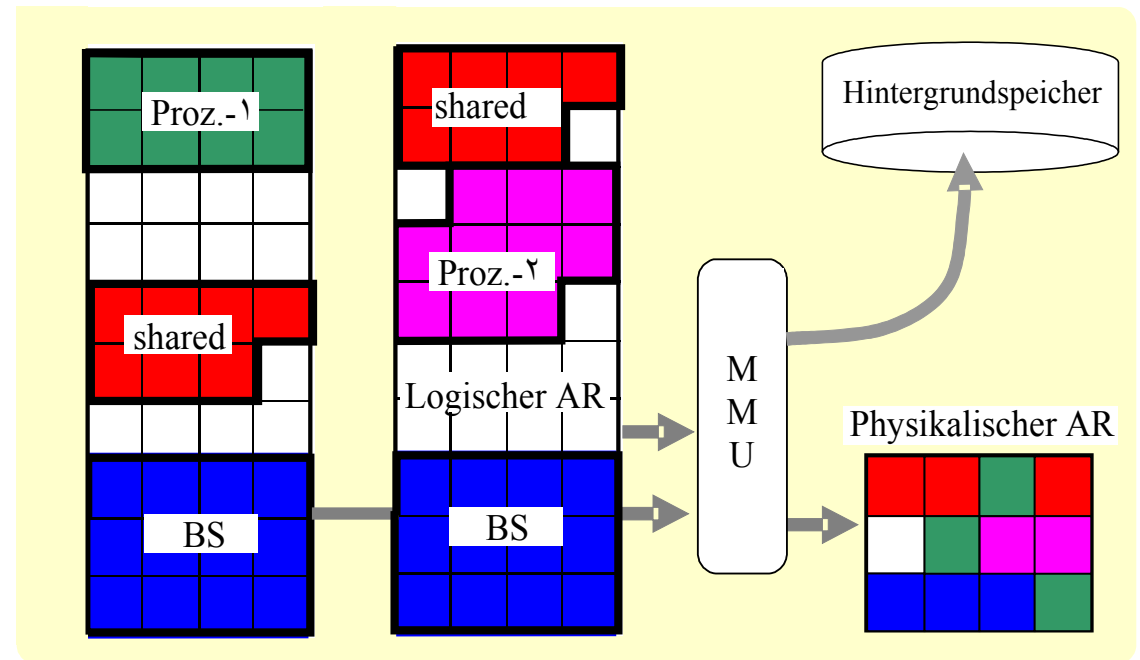
E.4.8 Kombination von Segmentierung und Paging

- Soll den Nachteil des internen Verschnitts (Paging) und der externen Speicherfragmentierung (Segmentierung) mildern.
 - ◆ log. Adr.raum wird in Segmente geteilt.
 - ◆ Segmente werden in Seiten zerlegt.
 - ◆ Segmente können sich eine Kachel teilen.
 - ◆ Seitenersetzung aber schwieriger
- Speicherschutz:
 - ◆ i.d.R. nur auf Segmentebene,
 - ◆ zusätzlich auf Seitenebene möglich.
- Ausgelagert werden nur Seiten.
- Beispiel: Intel IA32.
 - ◆ Segmentierung ergibt lineare Adresse,
 - ◆ Paging ergibt physikalische Adresse (PageDirectory & PageTables umfassen jeweils 4KB).



E.4.9 Shared Memory

- Für DLLs & zum Datenaustausch bei mehrfach virtualisiertem Speicher
- Ein Speicherbereich kann an verschiedene log. Adressen eingeblendet werden
- Sharing erschwert die Auslagerungsentscheidung.



- Copy-On-Write
 - ◆ Schreibt ein Prozess, so wird die zugehörige Seite kopiert,
 - ◆ Verwendet für Unix fork, globale Variablen in shared libraries/DLLs, ...,
 - ◆ Ziel: keine Replizierung in mehrfachen Adressräumen von Read-Only Speicherinhalten.

E.5 Ersetzungsalgorithmen

E.5.1 Einlagerungsstrategie

- Demand Paging:
 - ◆ Nur die benötigten Seiten eines Programms werden eingelagert.
 - ◆ Einlagerung erfolgt erst bei Bedarf, also bei einem Seitenfehler.
- Pre-Paging:
 - ◆ Versuch, hohe Seitenfehlerrate beim Lauf eines Programms zu vermeiden.
 - ◆ Kosten/Nutzen-Verhältnis von Pre-Paging hängt davon ab, ob die in der Zukunft benötigten Seiten eingelagert werden können.
 - ◆ Evt. bei einem Seitenfehler benachbarte Seiten auch einlagern
=> Lokalität berücksichtigen.
- Kombination von Pre- und Demand-Paging:
 - ◆ Pre-Paging zu Beginn der Programmausführung vermeidet anfängliche hohe Seitenfehlerrate. für den Anfang des Programmcodes, für statische Daten, Teile des Heaps und Stacks.
- weitere Einlagerungen erfolgen durch Demand Paging.

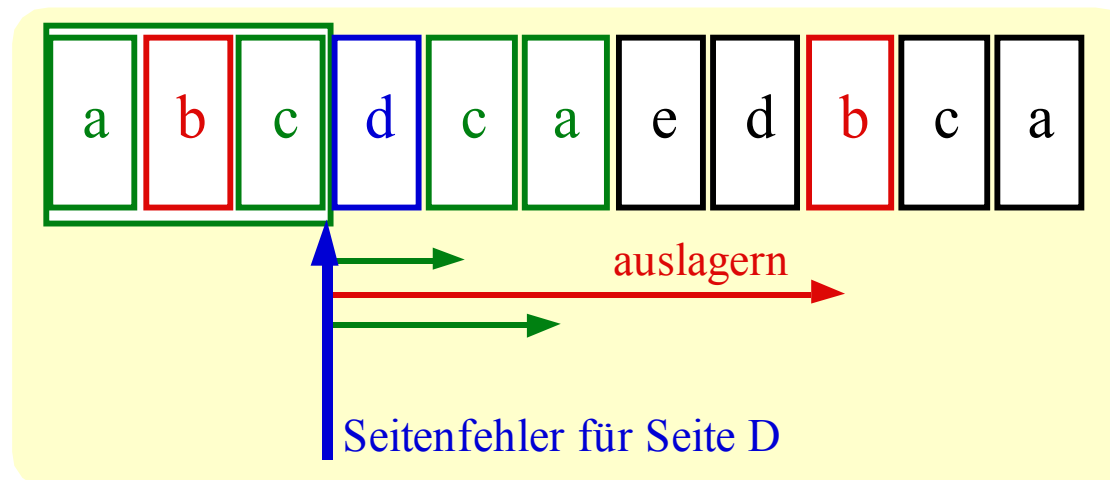
E.5.2 Auslagerungsstrategie

■ Prinzip der Seitenersetzung:

- ◆ Ist eine Seite nicht im HS (page fault), so wird sie mit Seiteneretzungsverfahren eingelagert.
- ◆ Ist der HS erschöpft muss zuvor eine Seite auf Disk ausgelagert werden, damit Platz frei wird.
- ◆ Ein-/Auslagern teuer, deshalb soll der Ersetzungsalgorithmus Seitenfehler minimieren.

■ Optimale Seitenersetzung:

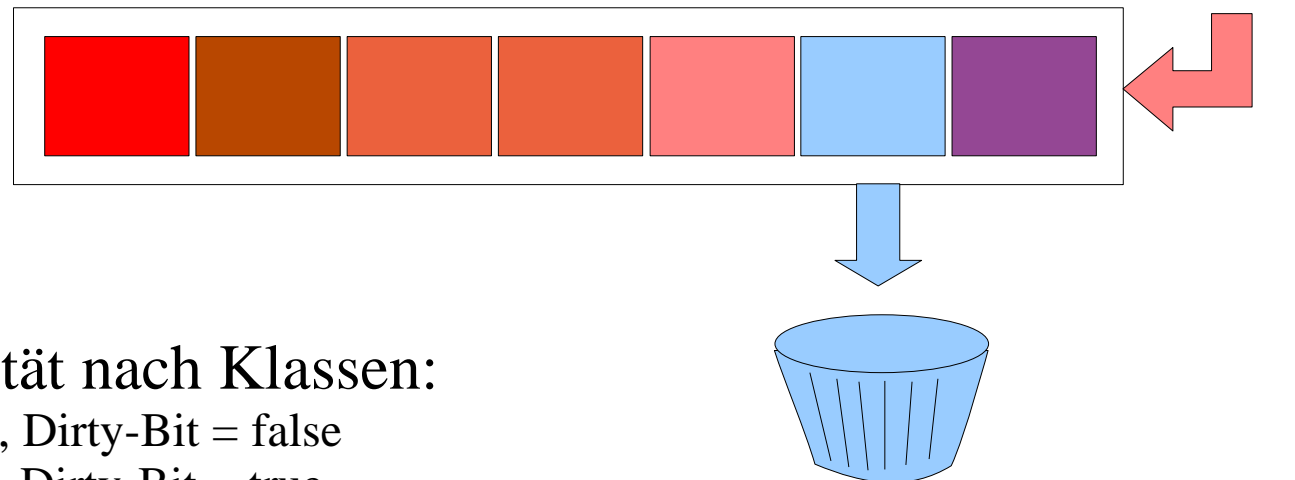
- ◆ Ersetze die Seite, die zukünftig am längsten nicht mehr benötigt wird.
- ◆ Die Zukunft ist aber unbekannt => theoretische untere Grenze
- ◆ Beispiel für einen HS mit 3 Kacheln:



E.5.3 Not recently used (NRU)

Flags in der Seitentabelle:

- ◆ Accessed-Bit falls die Seite referenziert wurde (periodisch zurücksetzen)
- ◆ Dirty-Bit falls Seite verändert wurde (nicht periodisch zurücksetzen, zeigt an, ob Seite beim Auslagern auf Disk geschrieben werden muss).



Auslagerungspriorität nach Klassen:

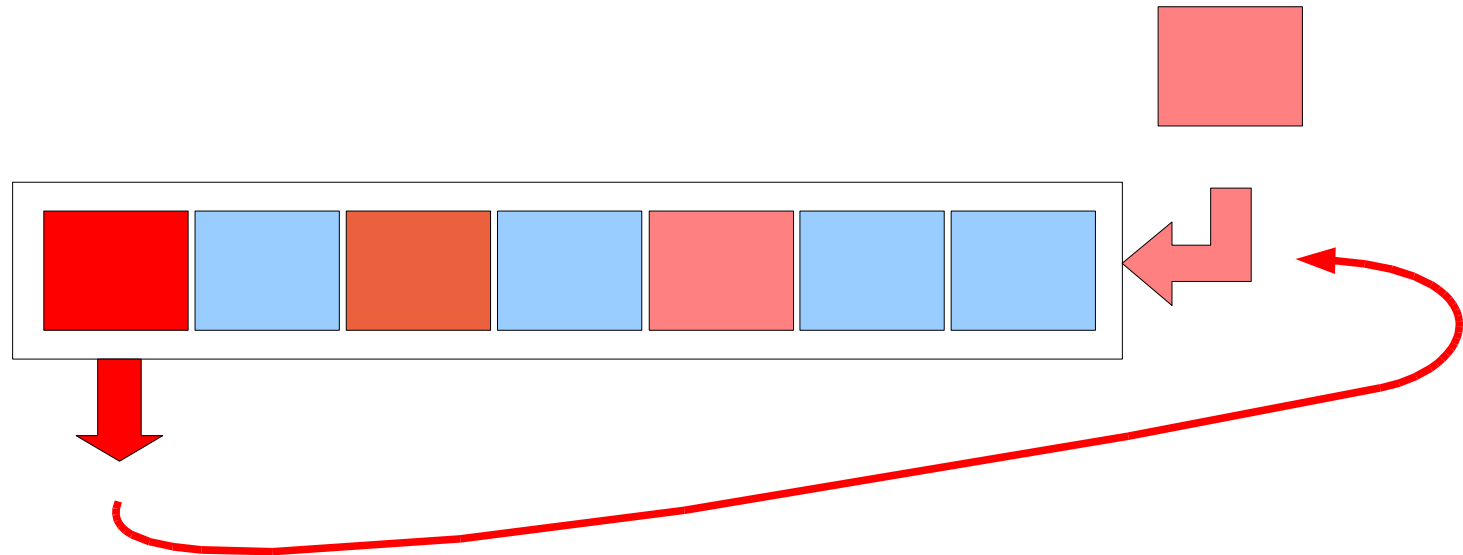
- ◆ A: Accessed-Bit = false, Dirty-Bit = false
- ◆ B: Accessed-Bit = false, Dirty-Bit = true
- ◆ C: Accessed-Bit = true, Dirty-Bit = false
- ◆ D: Accessed-Bit = true, Dirty-Bit = true

z.B.:

- ◆ Priorität B beschreibt eine Seite, die in einem früheren Intervall verändert wurde und noch zurückgeschrieben werden muss.

E.5.4 First-in, First-out (FiFo)

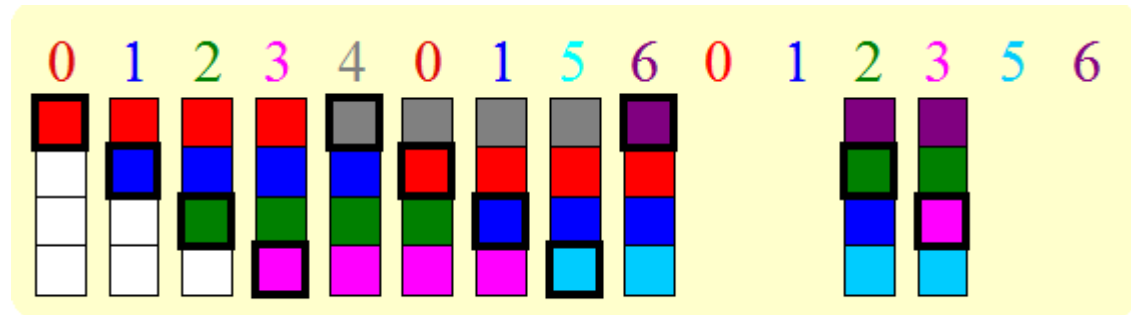
- Einfache Liste der eingelagerten Seiten.
- Die am längsten residente Seite wird ersetzt.
- Nachteil:
 - ◆ Auch häufig genutzte Seiten (**hot pages**) werden entfernt.
 - ◆ ungünstig bei zyklischen Zugriffsmustern.



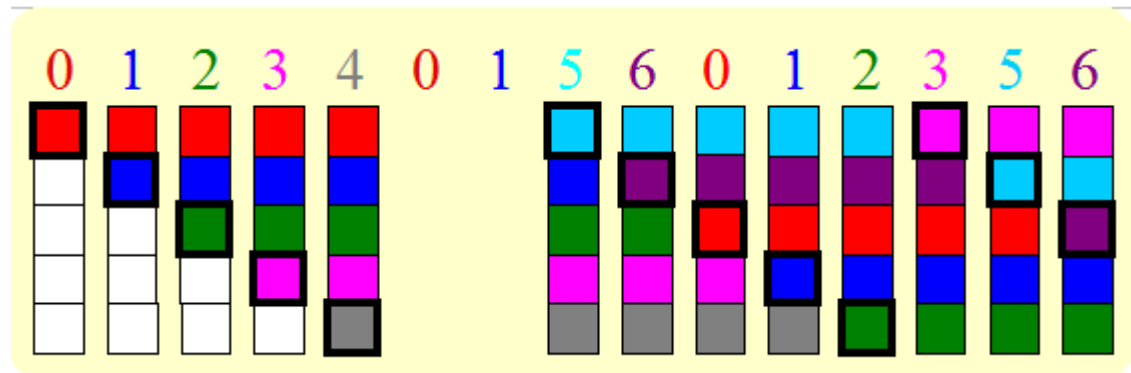
E.5.5 Belady's Anomalie

■ Mehr Seitenfehler trotz mehr Kacheln: tritt u.U. bei FIFO-Strategie auf.

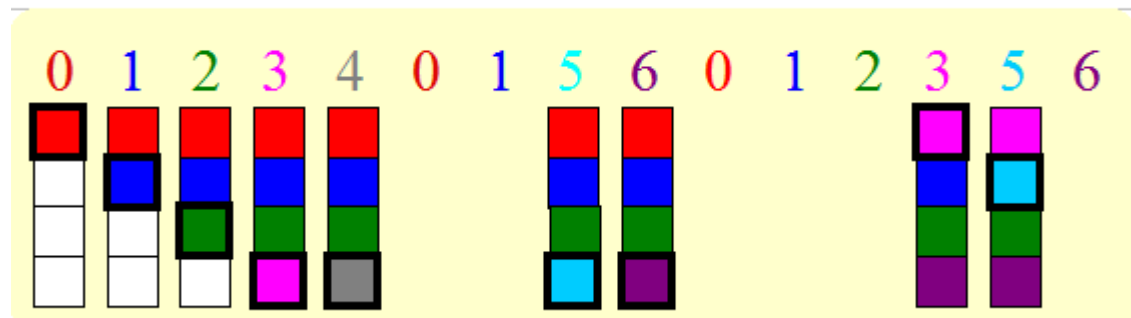
■ 4 Kacheln
→ 11 Seitenfehler:



■ 5 Kacheln
→ 13 Seitenfehler:



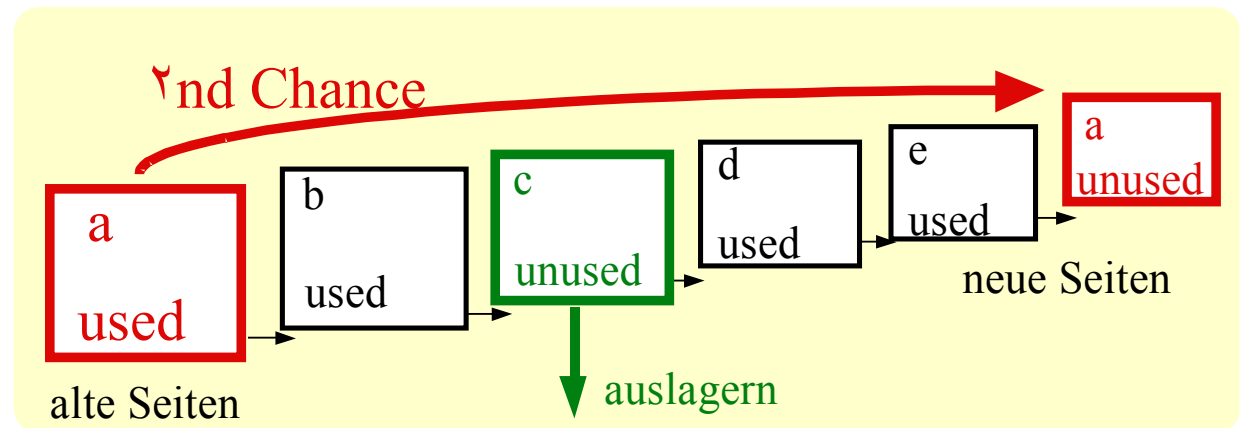
■ Optimal: 4 Kacheln
→ 9 Seitenfehler:



E.5.6 Verfahren des zweiten Versuches

- "Second chance page replacement algorithm".
- Falls USE-Bit (=Accessed-Bit) gelöscht, dann Seite auslagern.
- Falls USE-Bit gesetzt: → zweite Chance
 - ◆ USE-Bit zurücksetzen,
 - ◆ Seite hinten erneut einordnen.

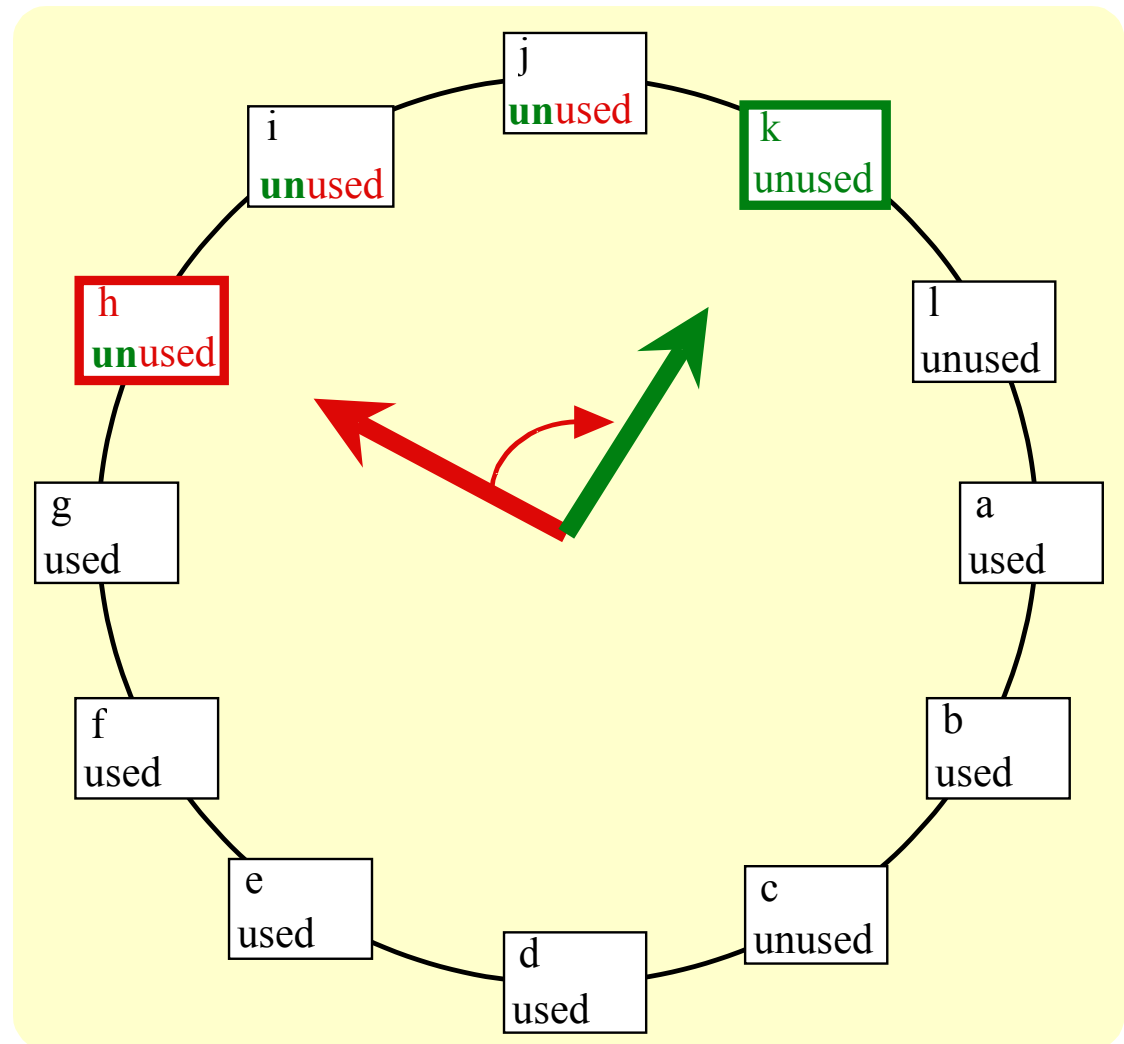
- Mehr Verwaltungsaufwand für die Liste als bei einfachem Fifo.



- Problem: evt. lange Suche nach Auslagerungs-Kandidat
- Bemerkung:
 - ◆ Zurücksetzen der Used-Bits impliziert auch, dass betroffene Seiten aus TLB gelöscht werden.
 - ◆ Andernfalls wird das Used-Bit beim nächsten Zugriff nicht gesetzt.

E.5.7 Uhrzeigerverfahren (engl. clock algorithm)

- Implementierungsvariante des Verfahrens des zweiten Versuchs.
- **Einen** Zeiger im Ring umlaufen lassen.
- Ringliste der Seiten absuchen.
- Falls Use-Bit gelöscht, dann Seite auslagern.
- Falls Use-Bit gesetzt, zurücksetzen & Zeiger inkrementieren.
- Unterschied zu „second chance“ Verfahren:
 - ◆ kein Umhängen von Einträgen.



E.5.8 Least recently used (LRU)

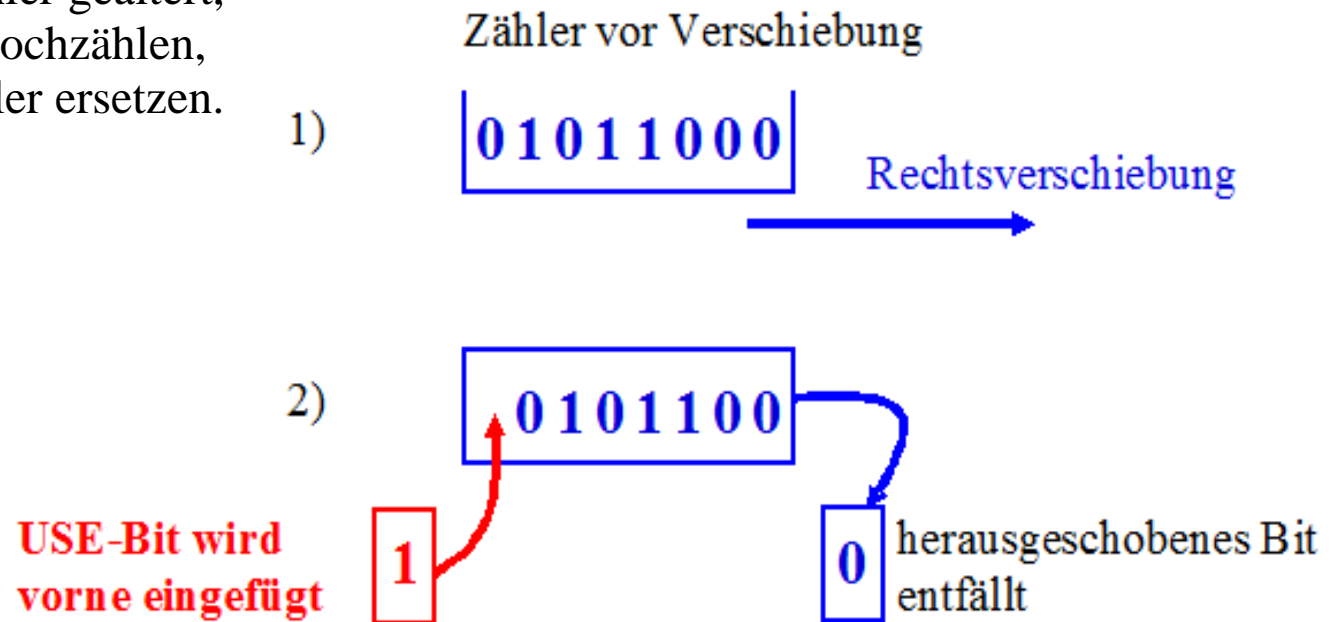
■ Am längsten unbenutzte Seite auslagern.

■ Theoretischer Ansatz:

- ◆ Zeitpunkt des letzten Zugriffs eintragen,
- ◆ 64 Bit Zeitstempel in Kacheltabelle halten,
- ◆ bei jedem Zugriff aktualisieren wäre zu teuer.

■ LRU-Annäherung in Software: →

- ◆ pro verwendeter Seite wird ein Zähler benötigt,
- ◆ periodisch werden die Zähler gealtert,
- ◆ Accessed-Bit dient dem Hochzählen,
- ◆ Seite mit niedrigstem Zähler ersetzen.
- ◆ Verwendet in Linux

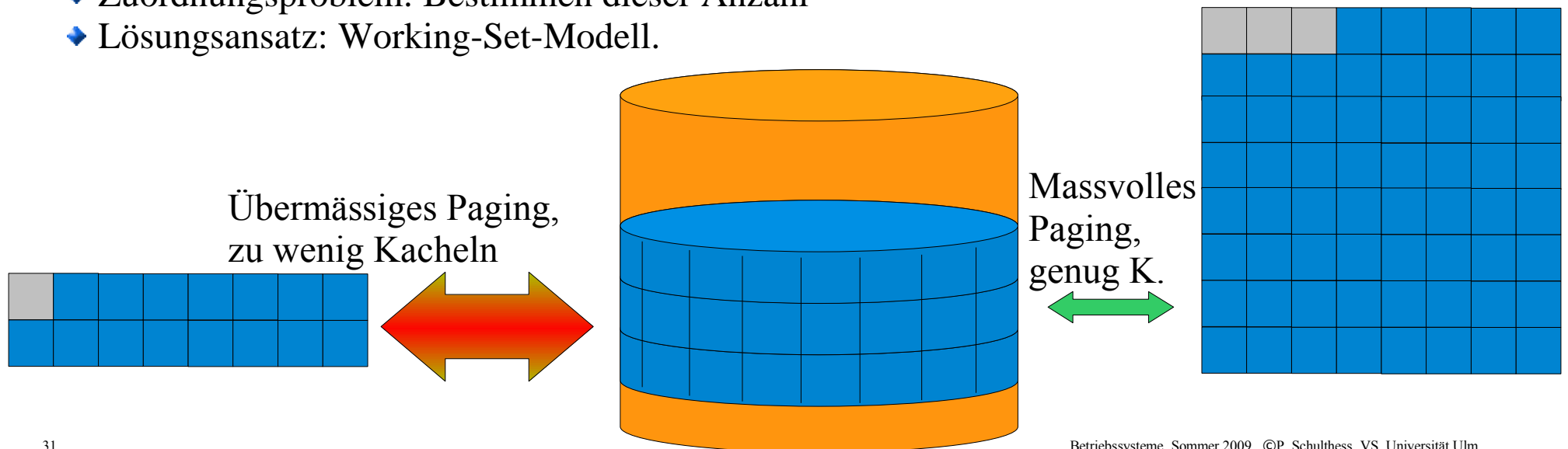


E.5.9 Allokation von Kacheln bzw. Seitenrahmen

- Jedes Programm benötigt eine Mindestanzahl an Kacheln.
- Allokation: proportional, gleichverteilt, prioritätenabhängig, ...
- **Lokale Strategie:**
 - ◆ Bei einem Seitenfehler werden nur Seiten des betroffenen Programms ausgelagert.
 - ◆ Programme haben eine feste Anzahl Kacheln zugeordnet.
 - ◆ Vorteil: andere Programme werden nicht beeinträchtigt.
 - ◆ Nachteil: Kachelbedarf schwer schätzbar.
- **Globale Seitenersetzungsverfahren:**
 - ◆ Tritt ein Seitenfehler auf, so stehen die Kacheln aller Programme zur Disposition.
 - ◆ Programme haben einen physikalischen Speicherbereich variabler Größe.
 - ◆ Vorteil: mehr Flexibilität => Optimierung des Gesamtsystems.
 - ◆ Nachteil: gegenseitige Beeinflussung des Paging-Verhaltens.

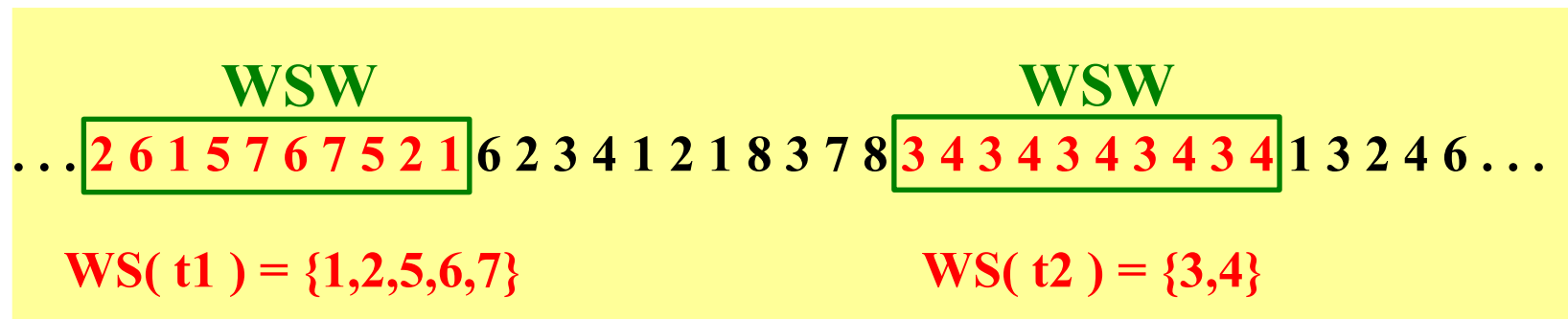
E.5.10 Thrashing (Seitenflattern)

- Falls ein Programm weniger Kacheln zur Verfügung hat, als der WS verlangt, so ergeben sich zahlreiche Seitenfehler (Thrashing).
- Mögliche Ursachen:
 - ◆ lokale Verfahren: Zahl der Kacheln zu gering.
 - ◆ globale Verfahren: ein Programm braucht zu einem Zeitpunkt sehr viele Kacheln, wodurch der Kachelvorrat aller anderen zu klein wird.
- Abhilfe:
 - ◆ eventuell Programme mit niedriger Priorität ganz auslagern => viele Kacheln werden frei,
 - ◆ falls möglich ausreichend grossen Kachelvorrat zuordnen,
 - ◆ Zuordnungsproblem: Bestimmen dieser Anzahl
 - ◆ Lösungsansatz: Working-Set-Modell.



E.5.11 Working-Set-Modell

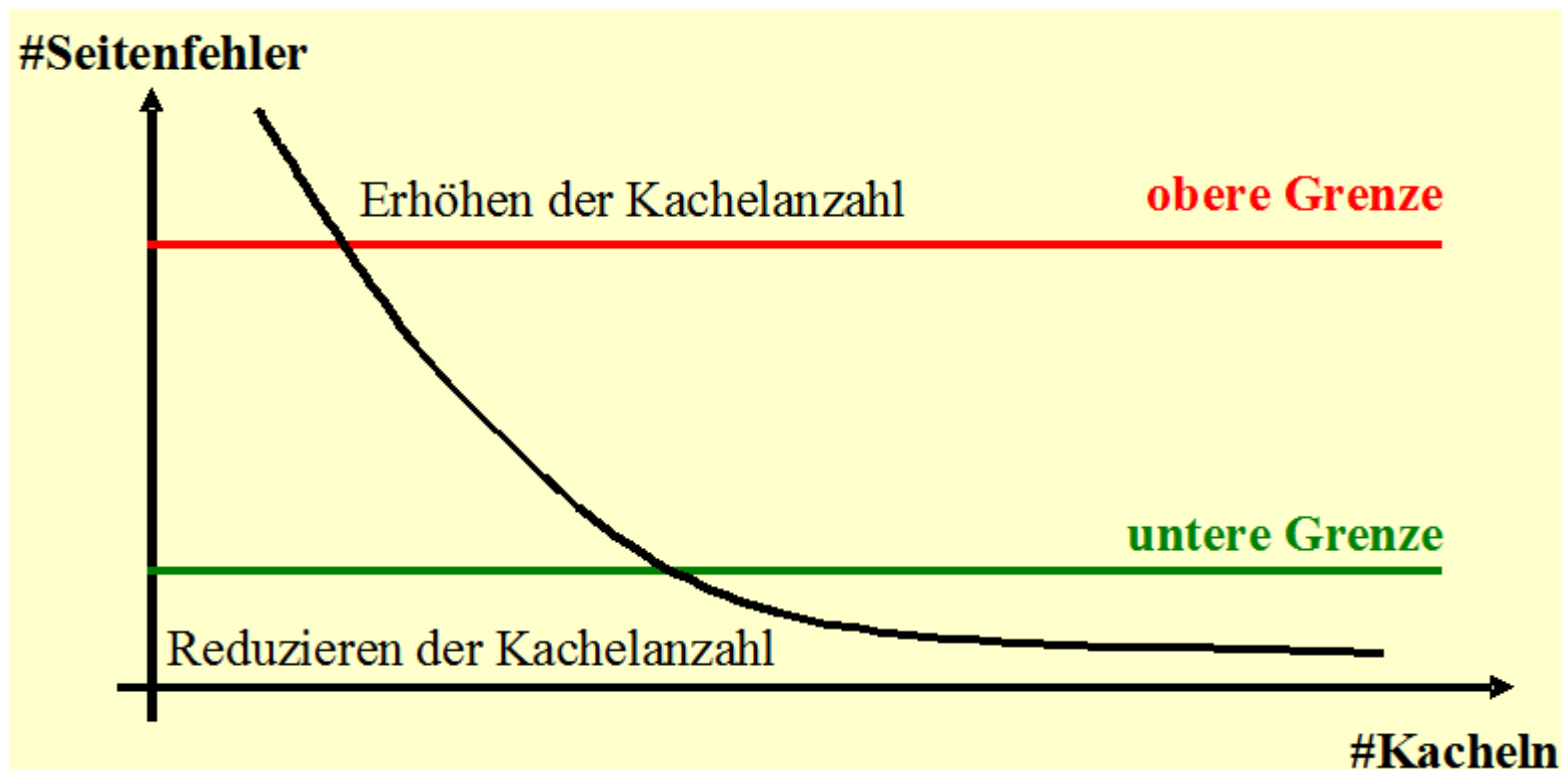
- = Arbeitsmengen-Modell - approximiert Lokalitätverhalten des Progr.
- **Working Set Window (WSW)** = Beobachtungsfenster der Länge T.
- **Working Set (WS)** = Seitenmenge, die im WSW referenziert wird.
- Beispiel: Seitenzugriffssequenz, WSW & Working-Set



- Wahl des WSW (Beobachtungsfenster):
 - ◆ zu kurz: WSW umfasst nicht das gesamte Working Set. WS erscheint zu klein.
 - ◆ zu lang: WSW umfasst mehrere Lokalitäten. WS erscheint zu gross.
- WS ändert sich im Laufe der Zeit.

E.5.1 Lösung: Page Fault Frequency Strategie

- falls Seitenfehlerrate $>$ oberer Schwellwert/Grenze:
 - ◆ Allokation zusätzlicher Kacheln, wegnehmen von anderen Programmen
 - ◆ wenn nicht mögl. Programm verdrängen.
- falls Seitenfehlerrate $<$ unterer Schwellwert/Grenze:
 - ◆ Freigabe von Speicherkacheln, für andere Programme bereistellen.
 - ◆ Bem.: verwendet in Windows NT/XP.



E.6 Beispiel: Linux

E.6.1 Physikalische Speicherverwaltung

- Zonen des physischen Adressraums:
 - ◆ ZONE_DMA: für ISA Geräte.
 - ◆ ZONE_NORMAL: Standardzone für Kern- & User-Mode (direkt für Kern zugreifbar).
 - ◆ ZONE_HIGHMEM: für Rechner mit > 896 MB RAM (nur indirekt für den Kern zugänglich).
 - ◆ Zonen-Information siehe auch Befehl dmesg.

- pro Zone definierte Schwellwerte (Watermarks):
 - ◆ Balance/Ausgleich zwischen Zonen
 - ◆ Auslagern, falls zu wenig freie Kacheln.

- Verwaltung der einzelnen Zonen in Buddy-Technik:
 - ◆ Fragmentierung innerhalb der 2^n -Byte großen Container; $n=\{12, \dots, 23\}$,
 - ◆ alloziert Kacheln fortlaufend (wenn möglich),
 - ◆ leere Blöcke schnell aggregierbar,
 - ◆ freie Blöcke schnell auffindbar.

E.6.1 Slab Allocator:

■ Zielvorstellung:

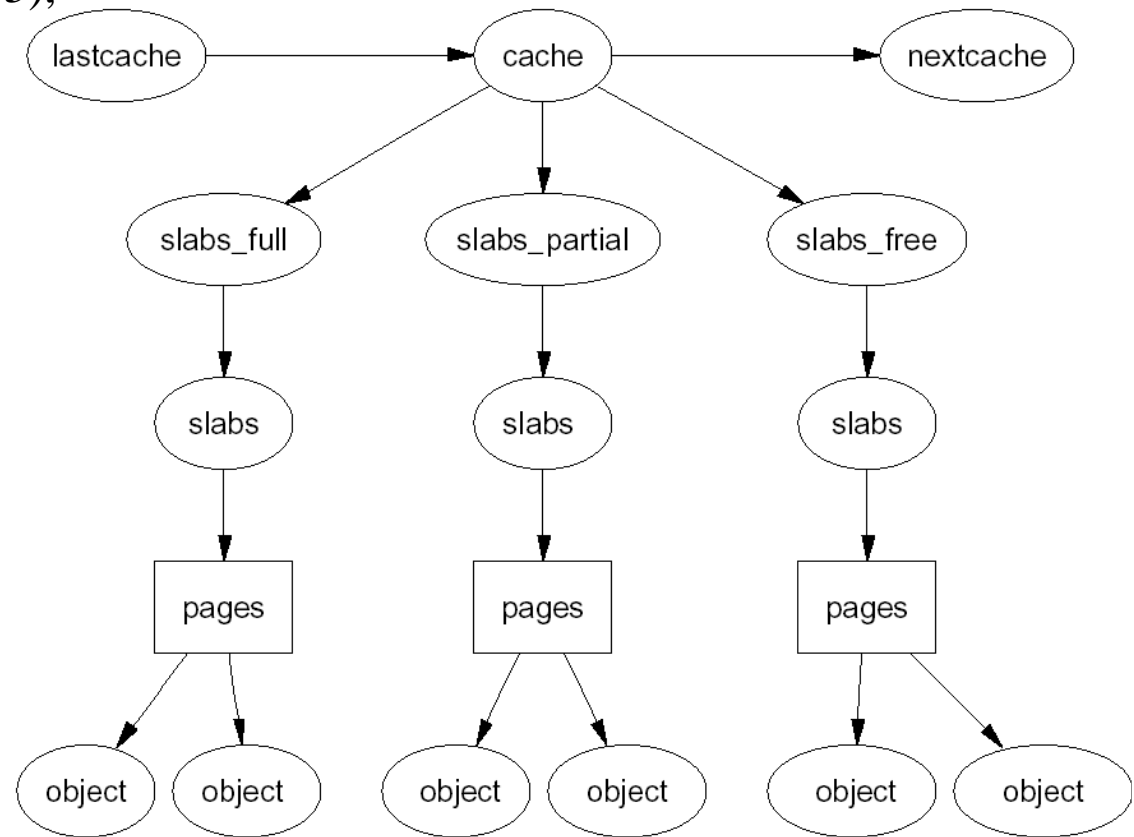
- ◆ gruppiert vorinitialisierte Objekte (z.B. I-Nodes) gleichen Typs (gleicher Größe),
- ◆ pro ‚slab‘ eine oder mehrere Speicher-Kacheln vorsehen,
- ◆ schnelles Auffinden von Blöcken mit passender Größe,
- ◆ Vorteilhaft zur Allokation von kleinen Objekten,
- ◆ Speicherallokation in den Größen $2x$ ($x > 5$),
- ◆ leere Slabs auf Anfrage freigeben,
- ◆ mildert interne Fragmentierung,
- ◆ geerbt von Solaris (1994).

■ "Caches" im Sinne v. Behälter:

- ◆ für einen Objekttyp/-größe,
- ◆ mehrere ‚slabs‘ pro Cache,
- ◆ Slabs: voll, partiell gefüllt, leer,

■ Info:

- ◆ auf Linux-Konsole: `cat /proc/slabinfo`
- ◆ <http://rtg.informatik.tu-chemnitz.de/mitarb/robge/talks/slaballocator.pdf>



E.6.2 Virtuelle Speicherverwaltung in Linux

■ Unterteilung des virtuellen Adressraums:

- ◆ User Mode: 0-3GB (privat pro Prozess), Kernel Mode: 3-4 GB (shared).
- ◆ Schutz für Kern und Anwendungen durch Segmentdeskriptoren,
- ◆ keine Segmentierung im User-Mode (flat memory model),
- ◆ nur Demand Paging (kein Pre-Paging).

■ Insgesamt komplizierte Speicherverwaltung:

- ◆ dreistufige Seitentabellenstruktur (entwickelt für 64-Bit Alpha).
- ◆ IA 32 verwendet nur eine zweistufige Tabelle.
(Eintrag im Page-Directory wird als mittlere Seitentabelle mit nur einem Eintrag behandelt).

■ Seitenersetzung nach globaler LRU-Strategie:

- ◆ modifizierter Clock Algorithmus, Use-Bit erweitert \wedge 8-Bit Altersvariable,
- ◆ beim ersten Zugriff nach einem Taskwechsel wird das Alter erhöht,
- ◆ Daemon kswapd verringert Alter periodisch (10s) oder bei Bedarf (Watermark),
- ◆ Code-Kacheln des Kern-Images werden nicht ausgelagert,
- ◆ Slabs werden gesondert behandelt (siehe Literatur).

■ Weitergehende Informationen unter:

- ◆ Understanding the Linux Memory Manager, 2004 (800 Seiten), http://www.phptr.com/content/images/0131453483/downloads/gorman_book.pdf,
- ◆ Linux Memory Management, <http://linux-mm.org>.

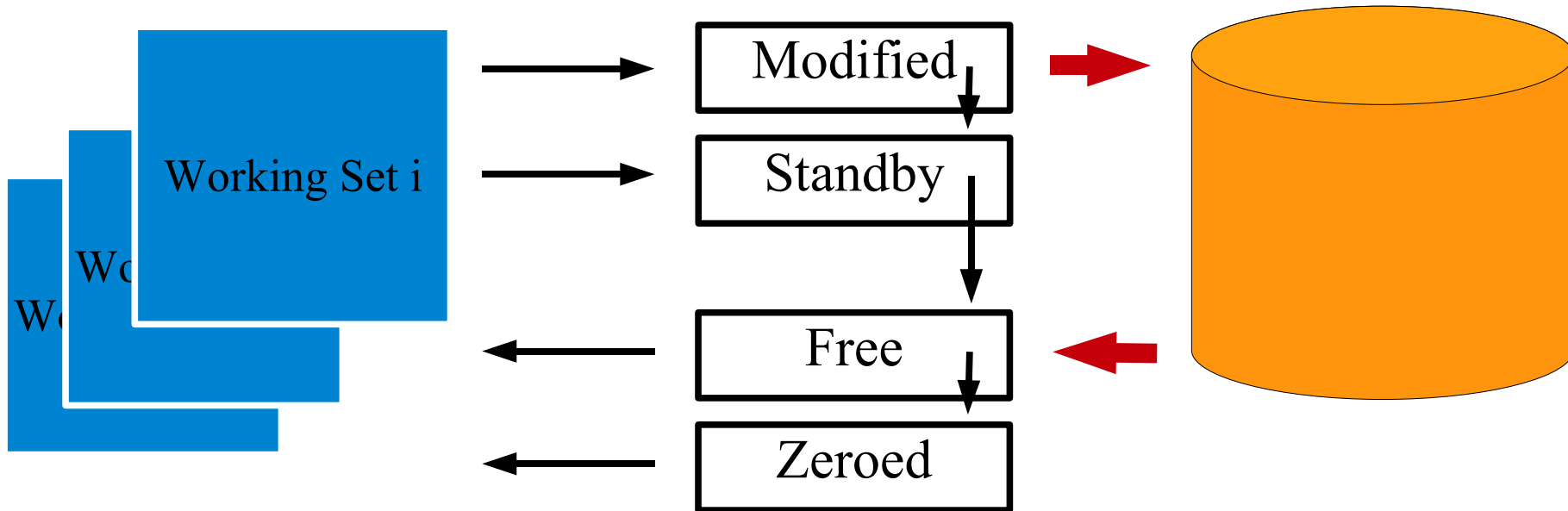
E.7 Beispiel: Windows NT etc.

E.7.1 Listen der Physischen Speicherverwaltung

- Jede Kachel ist in einem oder mehreren Working-Sets oder in einer der vier folgenden Speicherlisten.
- *standby list*:
 - ◆ unveränderte Seiten,
 - ◆ kürzlich aus Working-Set entfernt.
 - ◆ gehören derzeit noch zu einem Prozess.
 - ◆ Identische Kopie noch auf Festplatte; Umwidmung der Kachel ohne weiteres möglich.
- *modified list*:
 - ◆ enthält veränderte Seiten, sonst wie Standby-Liste, keine Kopie auf Disk!
- *free list*:
 - ◆ unveränderte Seiten, die zu keinem Prozess gehören.
- *zeroed pages list*:
 - ◆ wie free list, Seiten jedoch gelöscht mit Nullen.
- Separate Liste für Kacheln mit einem physikalischen Defekt.

E.7.2 Verschiebung von Kacheln zwischen Listen

- Kacheln wandern zwischen Working-Sets und verschiedenen Listen.
- Wird eine Kachel aus dem Working Set verdrängt, so kommt sie in die Modified/Standby–Liste.
- Falls eine Seite ausgelagert wird, Kachel in die Free–Liste verschieben.
- BS verschiebt Kacheln (spezielle Kernel-Threads) zwischen den Listen:
 - ◆ von Modified- auf Standby-List,
 - ◆ von Free- auf Zeroed-List.



E.7.3 Virtuelle Speicherverwaltung

■ Unterteilung des Adressraums:

- ◆ durch Segmente für Kern und Anwendungen (jeweils 2 Stück: 1 Daten und 1 Code).
- ◆ User Mode: 0-2GB (privat pro Prozess) und Kernel Mode: 2-4 GB (shared).
- ◆ Demand- und Pre-Paging (flat memory model).

■ Lokale Seitenersetzungsstrategie:

- ◆ Clock bei Einzel-CPU's => zurücksetzen des Used-Bits => TLB-Eintrag löschen.
- ◆ FIFO bei Multiprozessor-Systemen => Clock-Algorithmus zu teuer, da mehrere TLBs
- ◆ Kern verwendet zwei Speicherpools, einen mit und einen ohne Seitenersetzung.

■ Clustering (Pre-Paging):

- ◆ Beim Einlagern einer Seite werden Nachbarseiten mit eingelagert
- ◆ Data: 0-3 Seiten; Code: 1-7 Seiten

■ Kachelzuordnung beobachten(≠ Working-Set):

- ◆ es wird periodisch gezählt, wieviele Kacheln ein Prozess besitzt
- ◆ Grenzwerte für den WS-Manager festlegen: MIN: 20-50, MAX: 45-345 Kacheln,
- ◆ initiale Werte sind für alle Prozesse gleich, aber konfigurierbar mit speziellen OS-Aufrufen.

■ Anzahl Kacheln anpassen, je nach beobachteter Kachelzuordnung:

- ◆ $WS < MIN$: Kacheln für andere Prozesse bereitstellen (in Standby-Liste z.B.),
- ◆ $WS > MAX$: Kacheln zusätzlich beanspruchen (aus Free-Liste z.B.).

E.7.4 „Managers-Galore“

■ Balance-Set-Manager:

- ◆ Hintergrundprozess, der ein Mal pro Sek. prüft, ob genügend freie Kacheln vorhanden sind.
- ◆ Wenn nicht, WorkingSet-Manager starten.

■ Working-Set Manager:

- ◆ <http://www.cs.bgu.ac.il/~os032/assignments/Ex3/Working%20Set%20Management.htm>,
- ◆ progressiv aggressivere Durchläufe, falls hohe Paging-Rate beobachtet wird,
- ◆ passt WS-Größe an und entzieht Kacheln (=> Standby ...)
- ◆ bei Bedarf inaktive Proz. mit $WS > MAX$ auslagern.
- ◆ Prozesse mit $WS < MIN$ nur im Notfall auslagern.

■ Systemaufrufe zur Speicherverwaltung:

- ◆ seitenweise Allokation/Freigabe: `VirtualAlloc/VirtualFree` ,
- ◆ Unterscheidung zwischen Reservieren von logischem und physikalischem Speicher (`malloc()`)
- ◆ sperren von Speicher gegen Auslagerung (mit Einschränkung): `VirtualLock/VirtualUnlock`

■ Stacks:

- ◆ i.d.R. 1 MB logisch alloziert, aber physisch zunächst nur zwei Kacheln,
- ◆ eine Kachel als „guard trap page“ zum Abfangen von Stacküberlauf,
- ◆ eine Kachel wird für den ersten Stackframe alloziert,
- ◆ Stack kann dynamisch bei Bedarf wachsen.

E.8 Zusammenfassung

■ Segmentierung:

- ◆ variable Größe (vermeidet internen Verschnitt), aber beschränkte Anzahl,
- ◆ Adressübersetzung: Segmentstart + 32-Bit Offset (zB),
- ◆ variable Größe vermeidet internen Verschnitt,
- ◆ Schutz: Länge, Zugriffsart, Privilegstufe.

■ Paging:

- ◆ größerer logischer Adressraum unterteilt in Seiten fester Größe (z.B. 4 KB),
- ◆ Adressübersetzung: mehrstufig über Seitentabellen (sind auch auslagerbar),
- ◆ Ein virtueller Speicherblock muss nicht auf fortlaufende Kacheln abgebildet werden,
- ◆ Dies vermeidet externe Fragmentierung (im Vergleich zu reiner Segmentierung)

■ IA32: Segmentierung & Paging (abschaltbar) ist kombinierbar.

■ TLB puffert bereits übersetzte virtuelle Adressen.

■ Ersetzungsstrategien (für Paging & Segmentierung):

- ◆ verwenden accessed & dirty Bits (gesetzt durch MMU), Präsens-Bit verwaltet durch BS,
- ◆ dienen der Analyse des Verhalten in der Vergangenheit, um damit die Zukunft abzuschätzen.

■ Thrashing: Seitenflattern → ständig zu wenig Kacheln → WorkingSet näherungsweise bestimmen mit einer Page Fault Frequency Strategie.