Ankündigungen 9.10.2008

- Aktuell: Kapitel 4, virtueller Speicher
- Zuletzt: Seitenbasierter virtueller Speicher, Seitennachschubverfahren
- Heute: Segmentbasierter virtueller Speicher und Implementierungsaspekte
- Freitagsvorlesung (nicht im Basiskurs): Implementierung von virtuellem Speicher in ULIX
- Freitagsübung: Dynamische Seitenverdrängung

Seitennachschubverfahren

- Wenn man eine freie Hauptspeicherkachel gefunden hat, dann muss man auch eine Seite einlagern
- Demand-Paging: Seiten werden nur bei einem Seitenfehler eingelagert
  - Anwendung beginnt mit einem leeren Adressraum
  - Lokalitätsmenge baut sich langsam auf
  - Anfangs hohe Seitenfehlerrate
- Pre-Paging: Bestimmte Seiten werden im voraus in den Hauptspeicher kopiert (auch ohne Seitenfehler)
  - Im Extremfall wird die gesamte Anwendung eingelagert
  - Einlagerungen über die Lokalitätsmenge hinaus lohnen sich eigentlich nicht
- In der Praxis: Kombination aus beiden Verfahren

Kachelzuteilung

- Welche Kacheln sollen welchem Adressraum zugeteilt werden?
  - Bei einer Anwendung trivial
  - Bei mehreren Anwendungen unterschiedliche Bedürfnisse beachten
- Lokale Kachelzuteilung
  - Jeder Adressraum bekommt mindestens sein Working Set zugeteilt
  - Bei einem Seitenfehler wird eine Seite desselben Adressraums verdrängt
  - Problem: Bestimmung des Working Set
    - Wird durch die gemessene Seitenfehlerrate approximiert

Globale Kachelzuteilung

- Bestimmung der Seitenfehlerrate für jeden Adressraum sehr aufwändig
  - Besser: globale (systemweite) Seitenfehlerrate messen
- Ansatz: Globale Kachelzuteilung
  - Seiten verschiedener Adressräume verdrängen sich gegenseitig
  - Wenn globale Seitenfehlerrate über einen Grenzwert steigt, dann passen alle Lokalitätsmengen zusammen nicht mehr in den Hauptspeicher
- Problem: Anwendungen können sich gegenseitig andauernd Kacheln stehlen
  - Seitenflattern (Thrashing)
    - Kann zum Systemstillstand führen
  - Lösung: Auslagern ganzer Adressräume
    - Adressräume können wieder eingelagert werden, wenn Seitenfehlerrate einen bestimmten Wert unterschreitet

Dämon-Paging

- Schutz vor Thrashing: Dämon-Paging
  - System versucht immer, eine bestimmte Anzahl von freien Kacheln im System vorzuhalten
  - Aus diesem Vorrat können Seitenfehler ohne vorherige Seitenverdrängung bedient werden
  - Ein spezieller Betriebssystemprozess (Dämon) prüft periodisch, ob die Anzahl der freien Kacheln noch groß genug ist
    - Lagert bei Bedarf Seiten aus und schafft wieder Platz
- Trennung von Seitenverdrängung und Seiteneinlagerung
- Beispiel Linux:
  - Typischer Anteil an freien Kacheln im Hauptspeicher: 25%
  - Kernel Swap Dämon kswapd
  - Variante von LRU zur Bestimmung der Auslagerungskandidaten

Swapping

- Wenn Thrashing droht: Ganze Adressräume auslagern!
  - Alle Seiten der Anwendung zusammen mit den Seitentabellen können ausgelagert werden
  - Zurückschreiben der Seiten nur bei gesetztem Dirty-Bit
- Wer sollte ausgeswappt werden?
  - Möglichst ein Prozess in einem I/O-Burst
  - Wie bekommt man das heraus?
    - Siehe Kapitel 5
  - Sinnvoll: Prozesse auslagern, die eh viel Speicher benötigen
    - Schafft den größten Freiraum

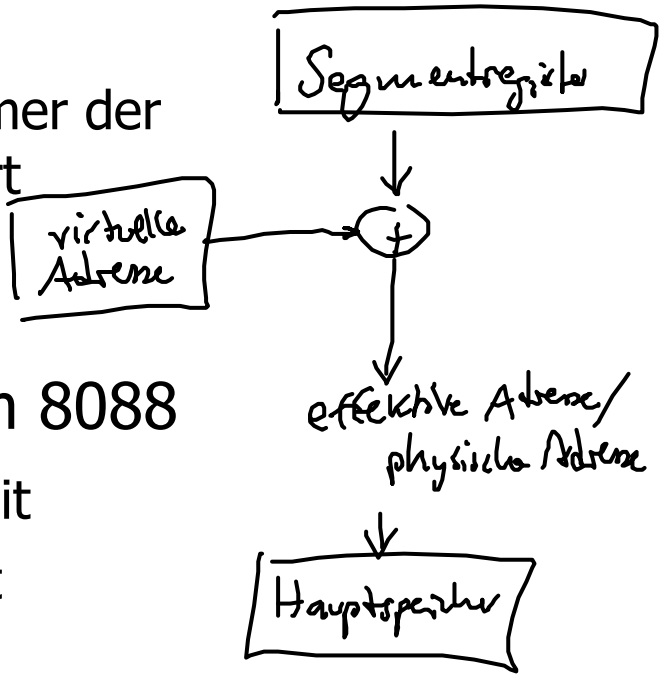
Übersicht

- Adressumsetzung (Einführung)
- Organisation von Adressräumen aus Anwendungssicht
- Seitenbasierter virtueller Adressraum
- Dynamische Seitenersetzung
- Segmentbasierter virtueller Adressraum
  - Einfache Hardwareunterstützung in Form von Segmentregistern ausnutzen
  - Historisch (vor seitenbasiertem virtuellem Speicher erfunden)
- Implementierungsaspekte
- Physischer (nicht-virtueller) Adressraum

Segmentbasierter Virtueller Speicher

Segmentregister

- Segmentregister (auch Basisregister genannt) ermöglichen die Adressierung von Speicherzellen relativ zu einer Basisadresse
  - Bei jedem Speicherzugriff wird implizit immer der aktuelle Wert des Segmentregisters addiert
  - Das Additionsergebnis ist die effektive Speicheradresse



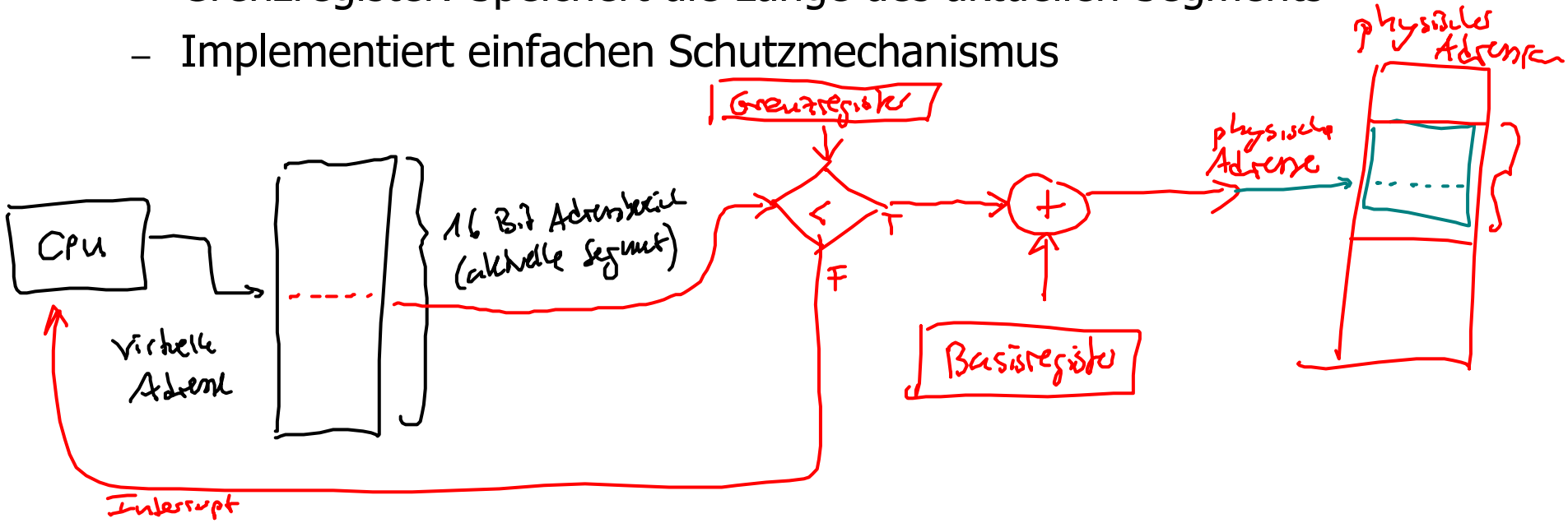
- Bei Intel-Prozessoren beginnt das beim 8088
  - Adressbus ist in Wirklichkeit nur 16 Bit breit
  - Segmentregister sind ebenfalls 16 Bit breit
  - 20-Bit effektive Adresse = Inhalt Segmentregister * 16 + Wert auf dem Adressbus
    - Moderne Intel-Prozessoren besitzen noch einen 16-Bit Kompatibilitätsmodus
  - Dadurch Erweiterung des Adressraums von 64 Kbyte auf 1 Mbyte

Intel-Segmentregister

- Moderne Intelprozessoren haben gleich mehrere Segmentregister
  - Code-Segmentregister (CS): Verwendet beim Zugriff auf Code und PC-relative Daten
  - Daten-Segmentregister (DS): bei normalen Datenzugriffen
  - Stack-Segmentregister (SS): bei Stackoperationen
  - Mehrzwecksegmentregister: ES, FS, GS
- Segmente sind auf 64 Kbyte beschränkt
  - Bei größeren Datenstruktruren müssen die Segmentregister andauernd neu geladen werden
    - Schwierigkeiten bei der Codeerzeugung in Compilern
- Segmentregister erlauben die freie Plazierung von Programmen im physischen Adressraum
  - Primitive Variante virtueller Adressierung mit relativ geringem Hardwareaufwand

Basis- und Grenzregister

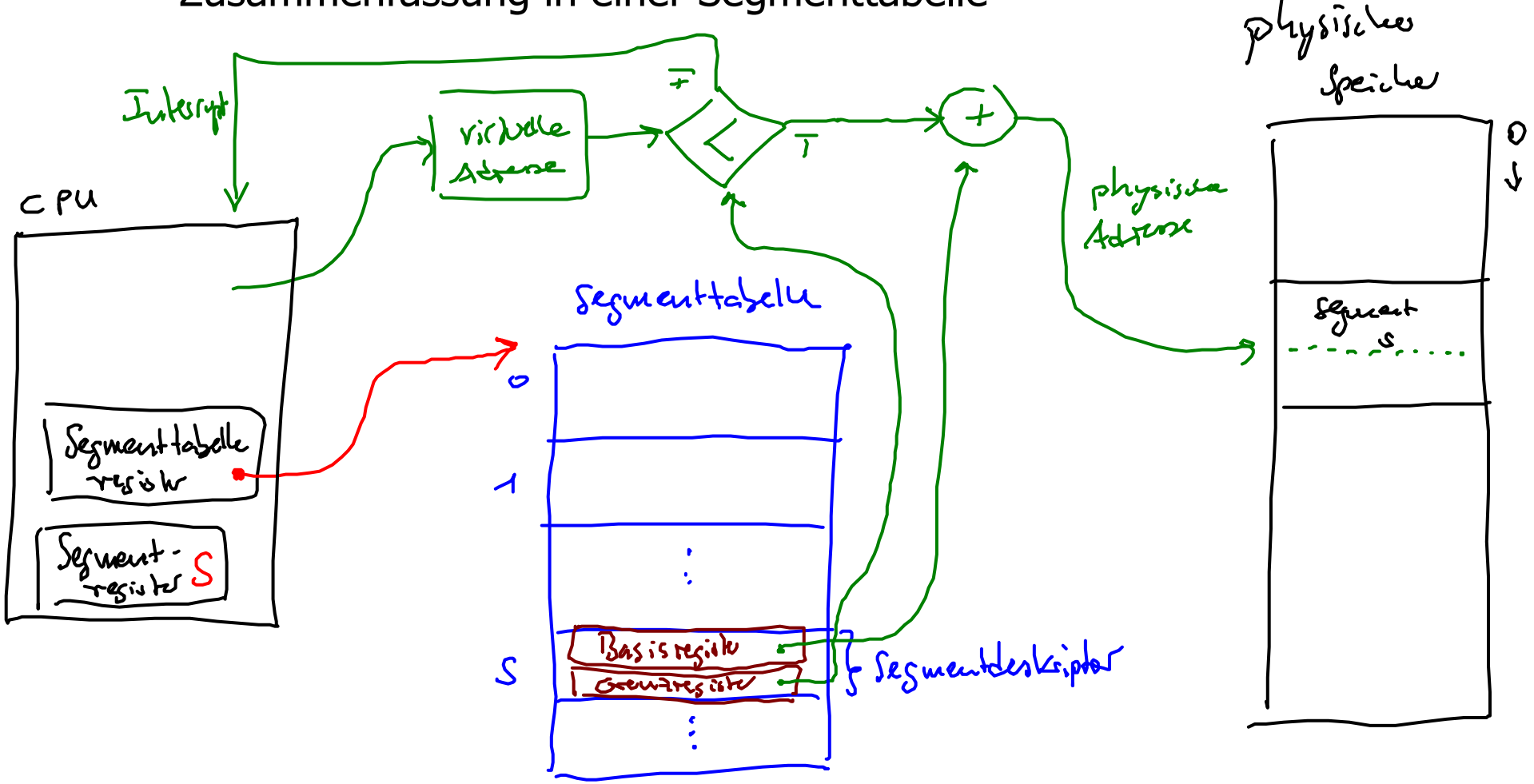
- Bei einfachen Segmentregistern ist jede (z.B. 16 Bit-) Adresse gültig
- Durch Erweiterung auf zwei Register, kann man Segmente verschiedener Länge realisieren
  - Segmentregister (Basisregister)
  - Grenzregister: Speichert die Länge des aktuellen Segments
  - Implementiert einfachen Schutzmechanismus



  - Jedes Segment benötigt weiter einen zusammenhängenden Bereich im Hauptspeicher

Segmenttabelle

- Ein Programm, das aus mehreren Segmenten besteht, benötigt mehrere Paare von Basis- und Grenzregistern
  - Zusammenfassung in einer Segmenttabelle

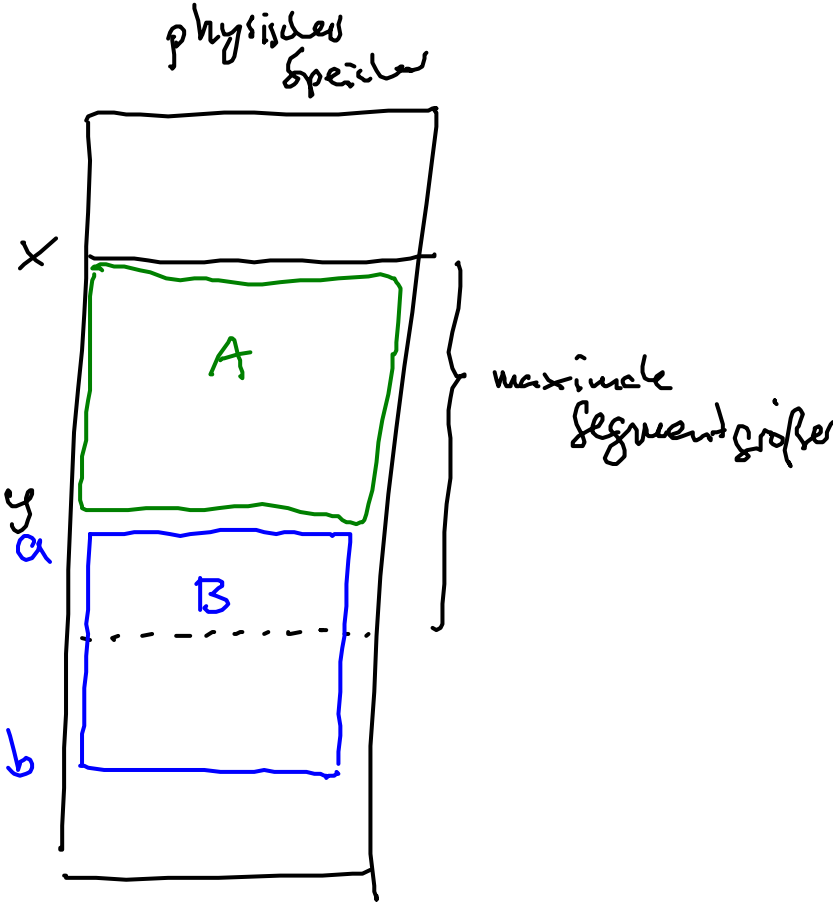


Segmente und Fragmentierung

- Gefahr von externer Fragmentierung!
- Verwendung von Segmenten bedeutet nicht automatisch interne Fragmentierung

Segment A:
Basisregister: x
Grenzregister: y - x

Segment B:
Basisregister: a
Grenzregister: b - a



Diskussion

- Segmenttabelle enthält Segmentdeskriptoren
  - Segmentdeskriptor: Paar von Basis- und Grenzregisterwerten
  - Segmentregister ist keine Adresse mehr sondern ein Index in die Segmenttabelle
  - Spezielles CPU-Register zeigt auf aktuell gültige Segmenttabelle
- Vorteile der Segmenttabelle
  - Das Umschalten zwischen Segmenten wird vereinfacht
    - Man braucht nur noch den Index in die Segmentabelle zu verändern
  - Kaum Effizienzverlust
    - Hauptspeicherzugriff beim Lesen des Segmentdeskriptors nur beim ersten Zugriff durch Verwendung eines prozessorlokalen Caches (analog TLB)

Beispiel: Segmente beim 80386

- Segmentdeskriptor kann neben Basis- und Grenzregisterwerten auch noch weitere Informationen speichern
- Beim Intel 80386 ist jeder Deskriptor 8 Byte groß und enthält:
  - Basisadresse (32 Bit)
  - Segmentlänge (20 Bit)
  - G-Bit (Granularitätsbit)
  - P-Bit (Presence-Bit)
  - A-Bit (Accessed-Bit)
  - Segmenttyp (5 Bit)
  - Privilegierungsstufe (2 Bit) = DPL (Descriptor Privilege Level)

Erläuterungen

- Segmentgröße durch G-Bit erweiterbar auf bis zu 4 Gbyte
  - Bei G-Bit = 1 wird Segmentlänge als Vielfaches von 4096 interpretiert
    - 20 Bit + 12 Bit = 32 Bit Segmentlänge
- P-Bit gibt an, ob sich das Segment im Hauptspeicher des Rechners befindet oder ausgelagert ist
- A-Bit wird bei jedem Zugriff auf das Segment gesetzt
  - Durch periodisches Prüfen und Zurücksetzen kann die Systemsoftware geeignete Auslagerungskandidaten bestimmen
- Segmenttyp bestimmt indirekt die Zugriffsrechte auf das Segment
- DPL legt fest, mit welcher Privilegierungsstufe (0 bis 3) man auf das Segment zugreifen darf
  - DPL = 0 bedeutet: Zugriff nur im Supervisor Mode

Erläuterungen (Forts.)

- Der 80386 besitzt zwei sichtbare Register mit Verweisen auf aktuell gültige Segmenttabellen:
  - GDTR: Global Descriptor Table Register
    - Zeigt auf die Segmenttabelle des Betriebssystems und allgemeiner Funktionsbibliotheken
    - Zugriff auf diese Segmente wird über DPL geregelt
  - LDTR: Local Descriptor Table Register
    - Zeigt auf die Segmenttabelle des aktuell laufenden Anwendungsprogramms
    - Enthält ein oder mehrere Code/Daten/Stack/Heap-Segmente
    - Wird bei Adressraumwechsel neu geladen
- Beide Tabellen sind auf maximal 8192 Einträge beschränkt

Zusammenfassung und Diskussion

- Auch mit segmentbasiertem virtuellen Adressraum kann man die meisten Anforderungen an virtuellen Speicher umsetzen
  - Auch Segmente können (wie Seiten) ein- und ausgelagert werden
- Vergleich mit seitenbasiertem virtuellem Adressraum:
  - Externe Fragmentierung: Jedes Segment benötigt einen zusammenhängenden Teil des phyischen Hauptspeichers
  - Seitenbasierter virtueller Speicher etwas einfacher zu implementieren
  - Segmentbasierter virtueller Adressraum benötigt Hardwareunterstützung im Prozessor (statt externe MMU)
  - Relativ hoher Zeitaufwand bei Segmentwechsel
- Man kann Segmentierung auch mit Paging kombinieren

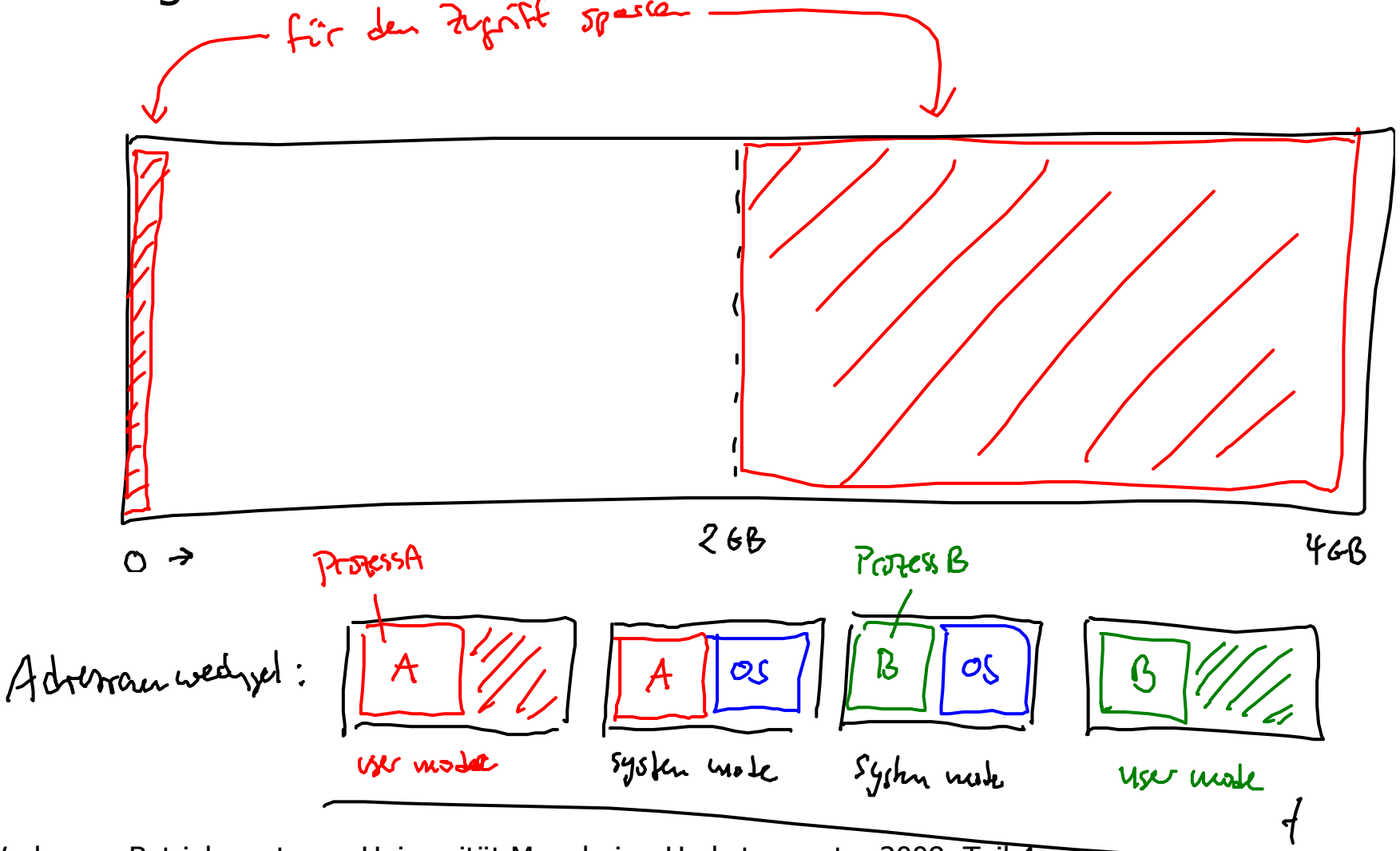
Übersicht

- Adressumsetzung (Einführung)
- Organisation von Adressräumen aus Anwendungssicht
- Seitenbasierter virtueller Adressraum
- Dynamische Seitenersetzung
- Segmentbasierter virtueller Adressraum
- Implementierungsaspekte
- Physischer (nicht-virtueller) Adressraum

Implementierungsaspekte

Organisation des Adressraums

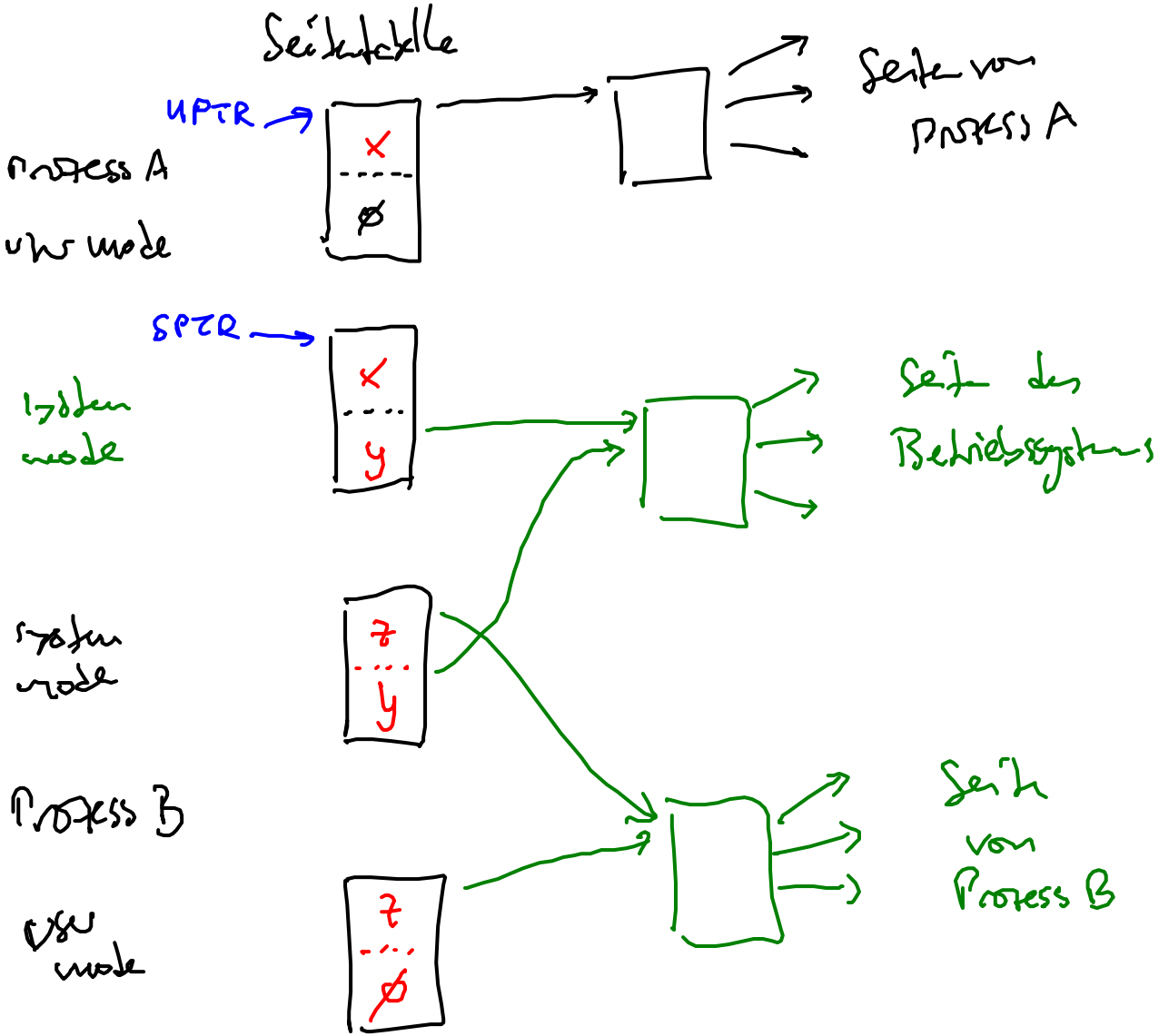
- Wie wird der "flache" virtuelle 32-Bit-Adressraum organisiert?



Erläuterungen

- Übliche Techniken:
  - Adresse 0 für Zugriff sperren
    - Dereferenzieren eines Nullzeigers (häufiger Programmierfehler) wird dadurch einfach vom System abgefangen
  - Für Programme sind häufig nur die unteren 2 GB vorgesehen
    - In oberen 2 GB wird das Betriebssystem eingeblendet
    - Im User Mode vor Zugriff geschützt
    - Erleichtert das Umschalten in den Supervisor Mode
      - Speicherbereiche sind bereits "vorgeladen"
      - TLB muss nicht immer gelöscht werden
  - In Windows werden die unteren 4 MByte für 16 Bit-Anwendungen freigehalten
    - Alte Anwendungen verwenden gemeinsamen physischen Adressraum

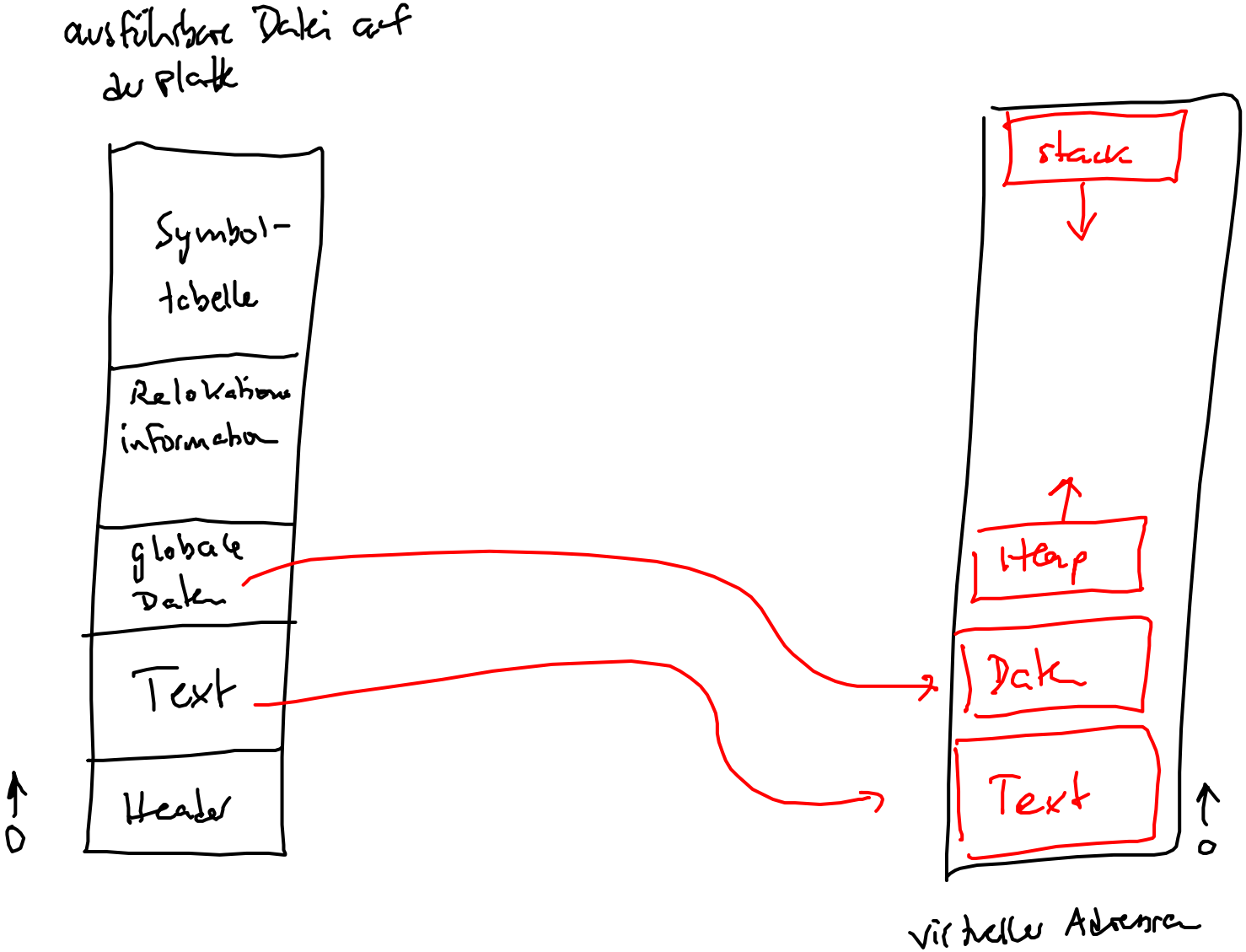
"Einblenden" der oberen 2GB



Adressraumerzeugung

- Vorgehen bei der Erzeugung eines neuen Adressraums:
  - Speicherplatz (Hauptspeicherkacheln) reservieren
    - für die Seitentabellen und
    - für eine Teilmenge der virtuellen Seiten
  - Initialisierung der Seitentabellen
  - Initialisierung der virtuellen Seiten des Programms
- Initialisierung der virtuellen Seiten:
  - In der Regel wird eine ausführbare Datei angegeben
  - Ausführbare Datei enthält mehrere Teile, die in einem Dateiheader beschrieben sind
  - Maschinenprogramm ist bereits "seitenweise" organisiert
    - erleichtert das Einblenden in den virtuellen Adressraum
  - Relokationsinformationen erlauben das Verschieben von Daten im virtuellen Adressraum
  - Symboltabelle für Debugger und zum Binden an Bibliotheken

Initialisierung mit Executable

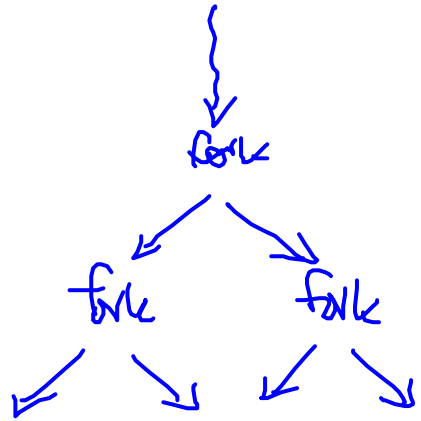


Beispiele

- Adressraumerzeugung und -initialisierung werden in vielen Betriebssystemen zusammengefasst
  - Beispiel Windows 9x und NT: **`Create(Dateiname, ...)`**
- UNIX trennt Erzeugung und Initialisierung:
  - Erzeugung: **`fork()`**
    - Erzeugt eine identische Kopie des ursprünglichen Adressraums
    - Verwendet Copy-on-Write (siehe gleich)
  - Initialisierung: **`exec()`**
    - Überschreibt den Inhalt eines Adressraums mit dem Inhalt einer ausführbaren Datei

Beispiel: `fork()`

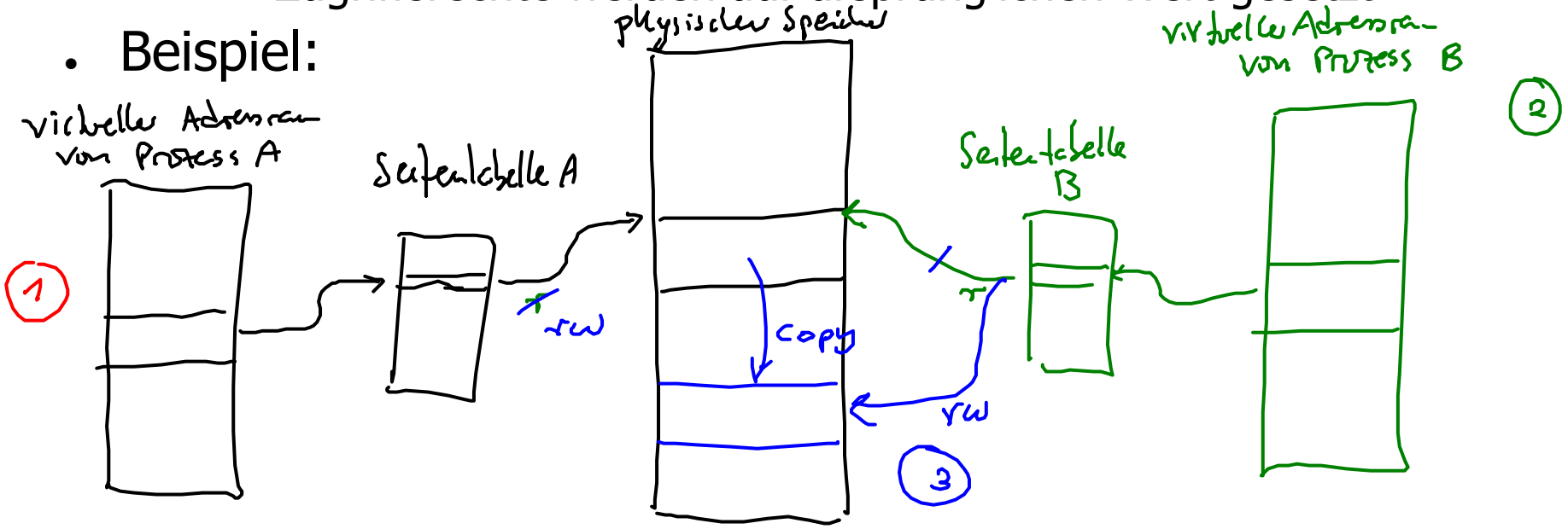
```
int main(...) {
   int r = fork();
   if (r==0) {
      // Code fuer Kindprozess
   } else {
      // Code fuer Elternprozess
   }
}
```




```
main (...) {
    fork();
    fork();
    bla ...
}
```

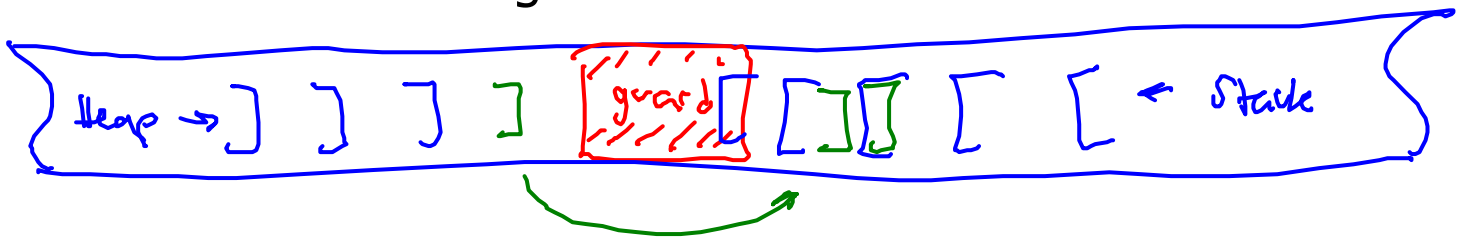
Copy-on-Write

- Verfahren für extrem schnelle Adressraumgenerierung
  - Kopiere anfangs ausschliesslich die Seitentabellen
  - Jede Seite wird vor dem schreibenden Zugriff geschützt
    - Spezielles Bit zeigt Copy-on-Write-Status an
  - Adressraum ist bei nicht-schreibendem Zugriff sofort benutzbar
  - Bei schreibendem Zugriff:
    - Kopie der Seite wird angelegt
    - Zugriffsrechte werden auf ursprünglichen Wert gesetzt
- Beispiel:



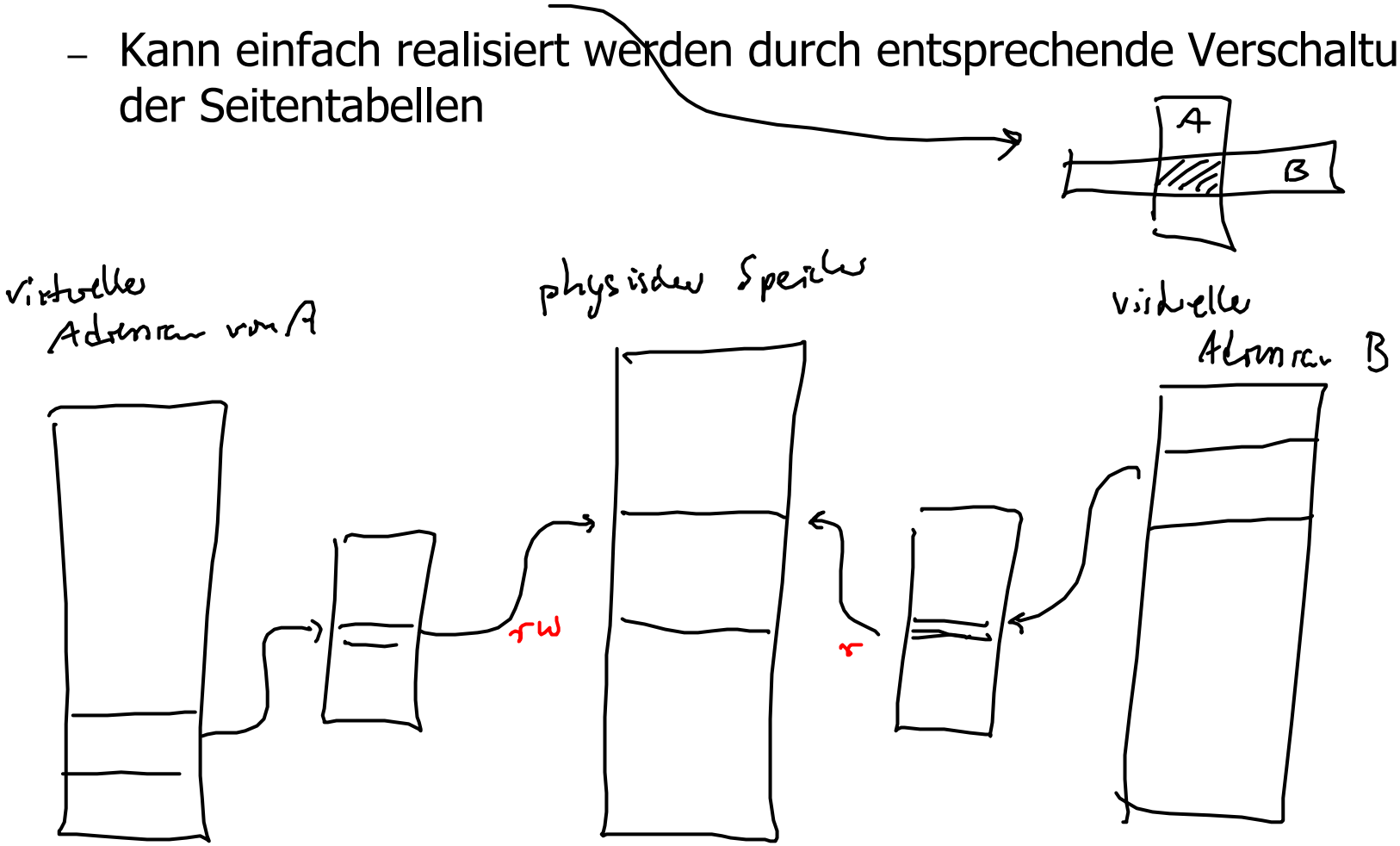
Überschneidung Heap/Stack

- Ohne Segmente müssen veränderliche Datenbereiche mit Implementierungstricks vor Überschneidungen geschützt werden
  - Vor und hinter Heap/Stack werden Speicherbereiche für den Zugriff gesperrt (Guard)
  - Bei Über-/Unterlauf löst die MMU einen Interrupt aus
  - Nach Analyse durch das Betriebssystem kann das Programm abgebrochen bzw. der entsprechende Speicherbereich erweitert werden
  - Problem: Es wird angenommen, dass der "überschreibende" Zugriff in den schreibgeschützten Bereich hineintrifft
    - Durch Guards wird nur die Wahrscheinlichkeit für den Überschneidungsfehler reduziert



Überlappende Adressräume

- Laufzeitmodell D aus Kapitel 3: gemeinsamer Speicher
  - Kann einfach realisiert werden durch entsprechende Verschaltung der Seitentabellen



Gemeinsame Bibliotheken

- Funktionsbibliotheken können als Spezialfall gemeinsam benutzter Adressraumbereiche angesehen werden
  - Bibliothek belegt nur ein Mal physische Ressourcen
    - Shared Libraries (Unix)
    - Dynamic Link Libraries (Windows)
  - Funktioniert nur bei positionsunabhängigen Code
    - Ansonsten Zwang zum Einblenden des Codes an einer festen Stelle im virtuellen Adressraum
  - Sollte sinnvollerweise vor schreibendem Zugriff geschützt werden
    - Notfalls Copy-on-Write verwenden
  - Referenzen auf Funktionen in der Bibliothek können erst zum Startzeitpunkt des Programms aufgelöst werden
    - Programme, die Bibliotheken verwenden, und Bibliotheken müssen speziell übersetzt werden
    - Querreferenzen werden mit Symboltabelle aufgelöst

Verankerung im Speicher

- Bestimmte Seiten müssen gelegentlich vor dem Auslagern geschützt werden
  - Spezielles Bit im Seitendeskriptor: locked
    - Zugeordnete Kacheln werden vom Verdrängungsverfahren ausgeklammert
- Beispiel 1: I/O-Locking
  - Ein-/Ausgabe auf eine Hauptspeicherkachel wird vom Betriebssystem angestossen
    - E/A wird asynchron durch Controller/DMA durchgeführt
  - Kachel darf erst *nach* Beendigung der E/A ausgelagert werden
- Beispiel 2: Teile des Betriebsystemcodes
  - (Teile der) Betriebssystem-Seitentabellen müssen immer vorhanden sein, genauso wie (Teile des) Betriebssystemcodes
    - Beispiel: Interrupthandler

Echtzeitaspekte

- Wenn ein Programm kritische Echtzeitschranken einhalten muss, sollten unerwartete Seitenfehler vermieden werden
  - Anwendung kann bestimmte Teile ihres virtuellen Speichers für die Auslagerung sperren
  - Setzt normalerweise bestimmte Privilegierungsstufe voraus
- POSIX-Funktionen:
  - Mit **`mlockall`** können alle aktuellen und ggf. auch alle zukünftig eingelagerten Kacheln vor dem Auslagern geschützt werden
  - Verfeinerte Kontrolle mit **`mlock`** und **`munlock`**

Übersicht

- Adressumsetzung (Einführung)
- Organisation von Adressräumen aus Anwendungssicht
- Seitenbasierter virtueller Adressraum
- Dynamische Seitenersetzung
- Segmentbasierter virtueller Adressraum
- Implementierungsaspekte
- Physischer (nicht-virtueller) Adressraum
  - Speicherorganisation ohne Hardwareunterstützung
  - für kleine/einfache/eingebettete Systeme

Physischer Adressraum

(Optional, nicht prüfungsrelevant)

Probleme

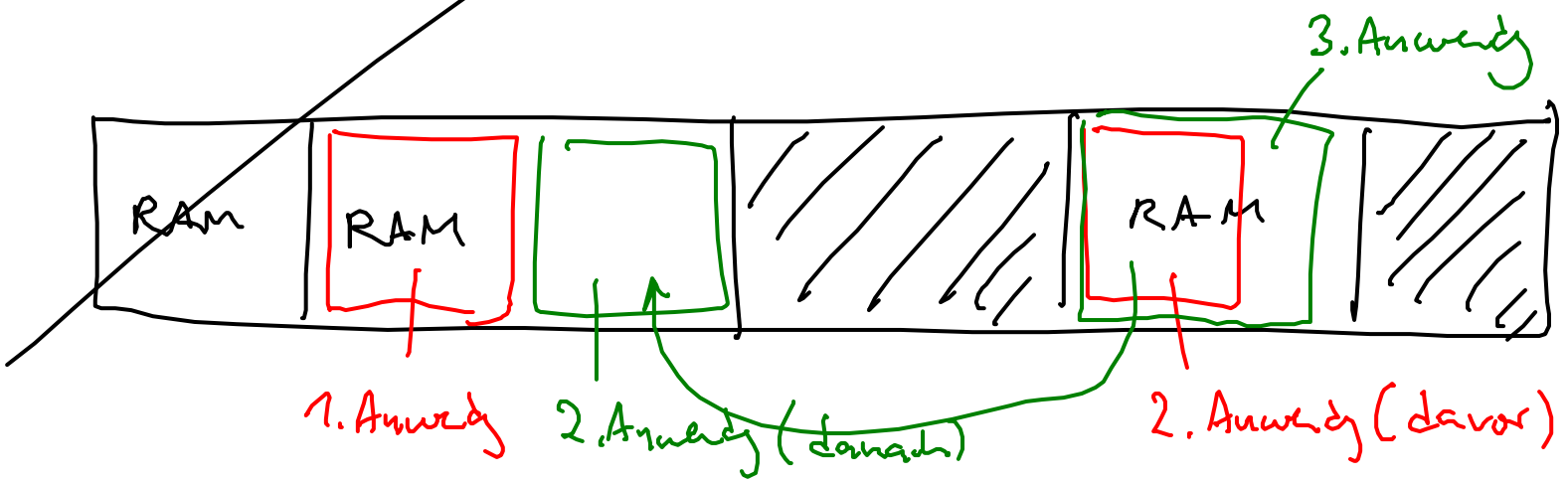
- Vergessen wir die zusätzliche Abbildungstabelle und kehren wir zurück zum reinen physischen Adressraum
  - Was ist da alles machbar?
- Problemfelder:
  - Der physische Adressraum ist selten homogen
  - Zugriffe auf Text und Daten sind nicht zu unterscheiden und können praktisch nicht erkannt werden
  - Überschneidung zwischen Stack und Heap sind ebenfalls nur sehr aufwändig zu erkennen
- Warum sollte man trotzdem ohne virtuellen Speicher arbeiten wollen?
  - Speicherabbildungstabelle kostet RAM
  - Zusätzlicher Aufwand ist für einfache Industriesteuerungen oder eingebettete Systeme zu groß

Schutz der Systemsoftware

- Wie kann der Adressraum der Systemsoftware vor unberechtigtem Zugriff aus einer Anwendung geschützt werden?
  - Einfachste Möglichkeit: Unveränderliche Teile der Systemsoftware im ROM Speichern
    - Wurde früher im Heimcomputerbereich gemacht (z.B. C64)
    - Upgrade der Systemsoftware sehr aufwändig
  - Andere Möglichkeit: Systemsoftware in einem Adressraum, der nur im privilegierten Modus zugreifbar ist
    - Prozessoren kommunizieren den Modus über Statusleitungen auf das Mainboard
    - Status kann in die Adressdecodierlogik eingebunden werden

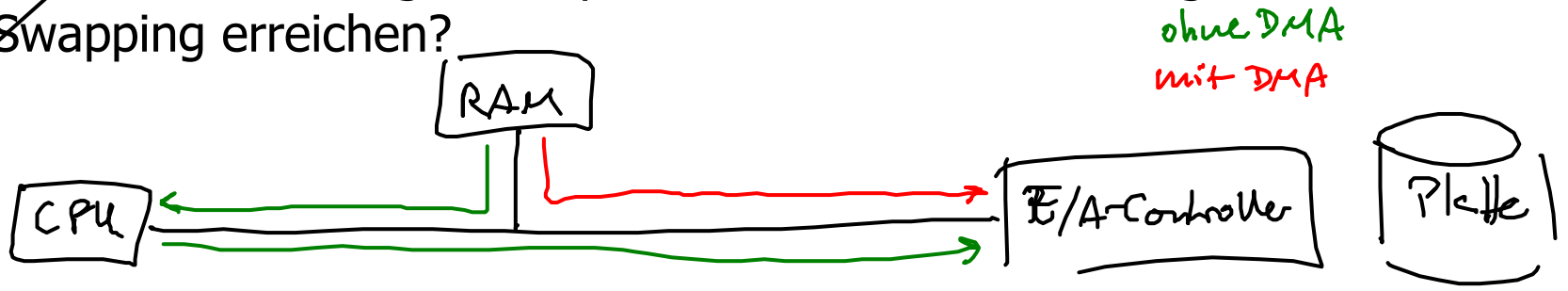
Mehrere Anwendungen

- Mehrere Anwendungen können sich gleichzeitig den physischen Adressraum teilen
  - Nutzbare Adressraum wird für jede Anwendung kleiner
  - Risiko der Überschneidung von Heap und Stack wächst
  - Schutz vor fehlerhaften Programmen nicht gegeben
- Beim Start von neuen Anwendungen müssen teilweise alte Adressräume im physischen Adressraum verschoben werden
  - Positionsunabhängiger Code hilt (keine absoluten Sprünge etc.)



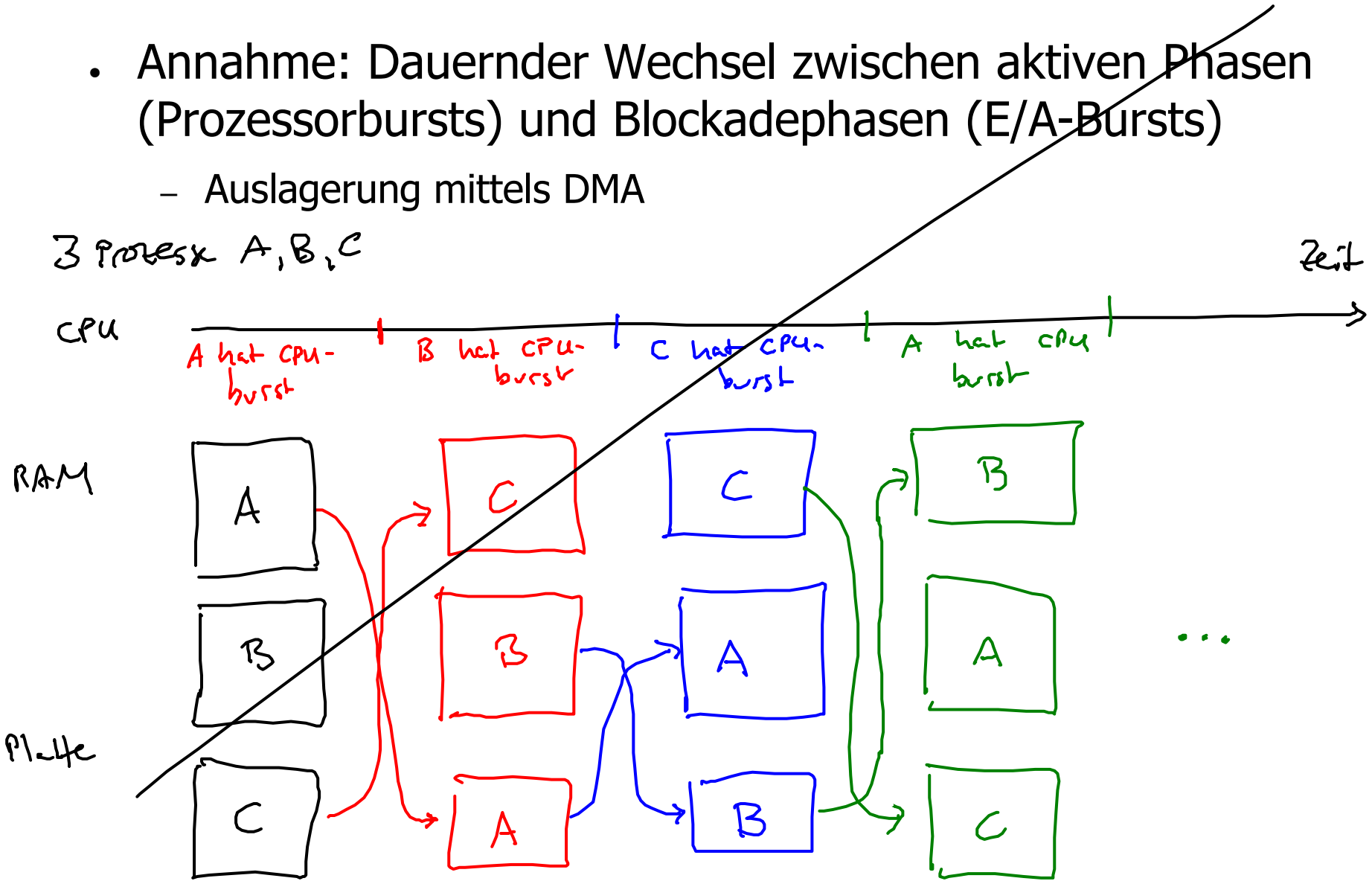
Swapping

- Implementierung eines Zeitmultiplexers, der den vorhandenen physischen Speicher reihum den Anwendungsadressräumen zuordnet
  - Wenn Speicherplatzanforderungen den verfügbaren physischen Speicherplatz übersteigen:
    - Auslagern eines Anwendungsadressraums auf Hintergrundspeicher
    - Späteres Einlagern des Adressraums, wenn wieder Speicher frei ist
- Optimierungskriterium: Prozessorauslastung
  - Wie kann man möglichst optimale Prozessorauslastung beim Swapping erreichen?



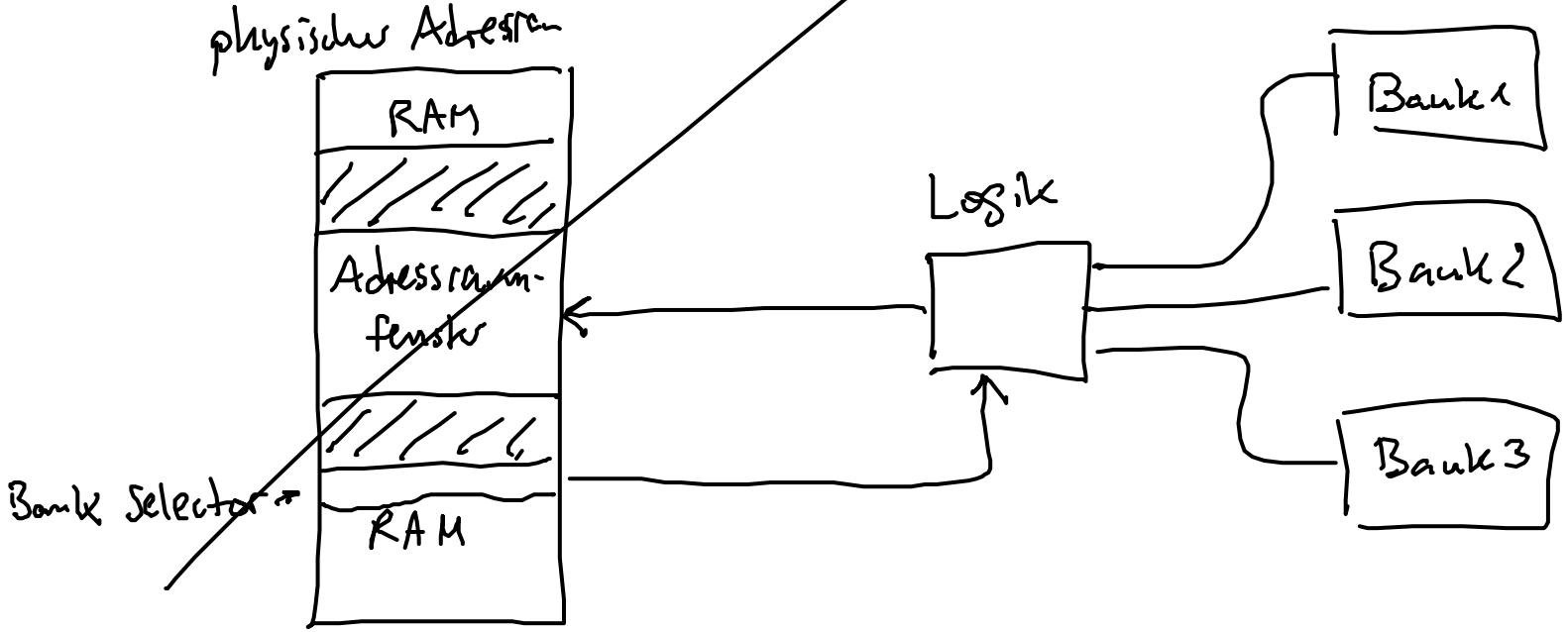
Optimales Swapping mit DMA

- Annahme: Dauernder Wechsel zwischen aktiven Phasen (Prozessorbursts) und Blockadephasen (E/A-Bursts)
  - Auslagerung mittels DMA



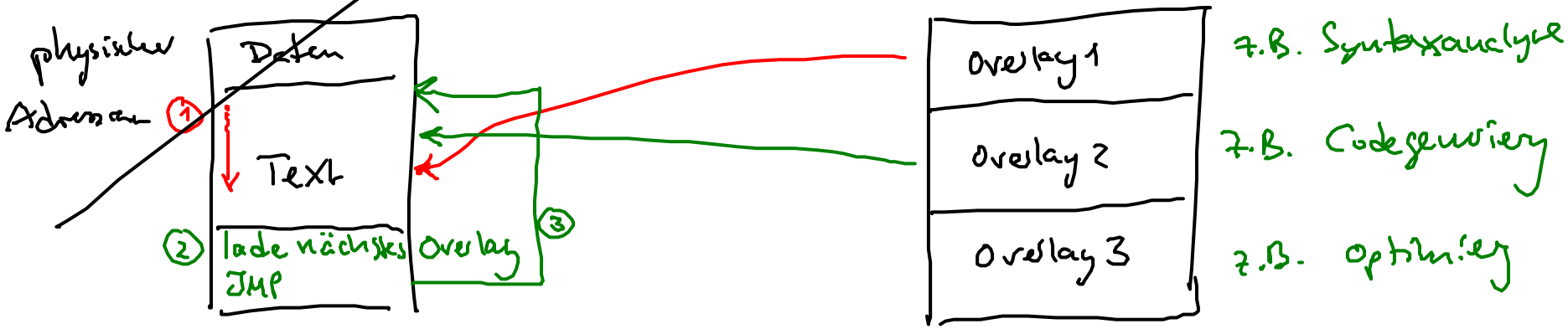
Bank-Switching

- Wie kann man Programme ausführen, die nicht vollständig in den Hauptspeicher passen?
- Erste Idee: Bank-Switching
  - Verschiedene gleichgroße Speicherbänke können abhängig von einem Speichereintrag (Bankselektor) in ein Adressfenster des physischen Speichers eingeblendet werden



Overlay-Technik

- Zweite Idee: Softwaremäßige Realisierung des Bank-Switching
  - Der Entwickler teilt die Anwendung in mehrere funktionell unabhängige Einheiten auf (Overlays)
  - Jeweils nur ein Overlay befindet sich im Speicher
  - Beim Übergang zum nächsten Overlay wird Code nachgeladen und überschreibt altes Overlay
  - Übergänge zwischen den Overlays werden am einfachsten durch den Compiler erzeugt
- Beispiel: der Compiler selbst (zwei Übersetzungsphasen)



Beispiel: MS-DOS

- Erste Version von MS-DOS von 1981
  - Geschrieben für den Ur-PC mit 8088-Mikroprozessor
  - Adressbus: 20 Bit, kann maximal 1 Mbyte adressieren
- Speicheraufteilung:
  - Die ersten 640 Kbyte sind für Betriebssystem und Anwendungsprogramme
  - Die verbleibenden 384 Kbyte sind für ROM-Speicher und speicherbasierte E/A-Geräte
- Mit besseren Prozessoren und höheren Benutzeransprüchen wurde die 640 Kbyte-Grenze schnell erreicht
  - Virtueller Speicher wurde (wohl aus Kostengründen) nicht eingebaut
  - Abwärtskompatibilität wichtig
  - Stattdessen: viele exotische Techniken, um trotzdem größere Programme verwenden zu können (z.B. HIMEM)
  - Erst mit Windows95 kommt ein 32-Bit virtueller Adressraum

Zusammenfassung

- Adressumsetzung (Einführung)
- Organisation von Adressräumen aus Anwendungssicht
- Seitenbasierter virtueller Adressraum
- Dynamische Seitenersetzung
- Segmentbasierter virtueller Adressraum
- Implementierungsaspekte
- Physischer (nicht-virtueller) Adressraum

Ausblick

- Wir wissen nun, wie man für einen einzelnen Prozess den Adressraum verwaltet
- In Zukunft: jeder Prozess/jedes Team lebt in einem eigenen virtuellen Adressraum
  - Eigene Seitentabelle
  - Schutz vor Zugriff aus anderen Adressräumen
  - Dynamische Seitenersetzung
- Betriebssystem lebt in eigenem virtuellen Adressraum
  - Adressraum aller Prozesse ist aus dem Betriebssystem heraus zugreifbar
- Wie werden einzelne Prozesse im Kontext des Betriebssystems verwaltet?
  - Kapitel 5: Threads