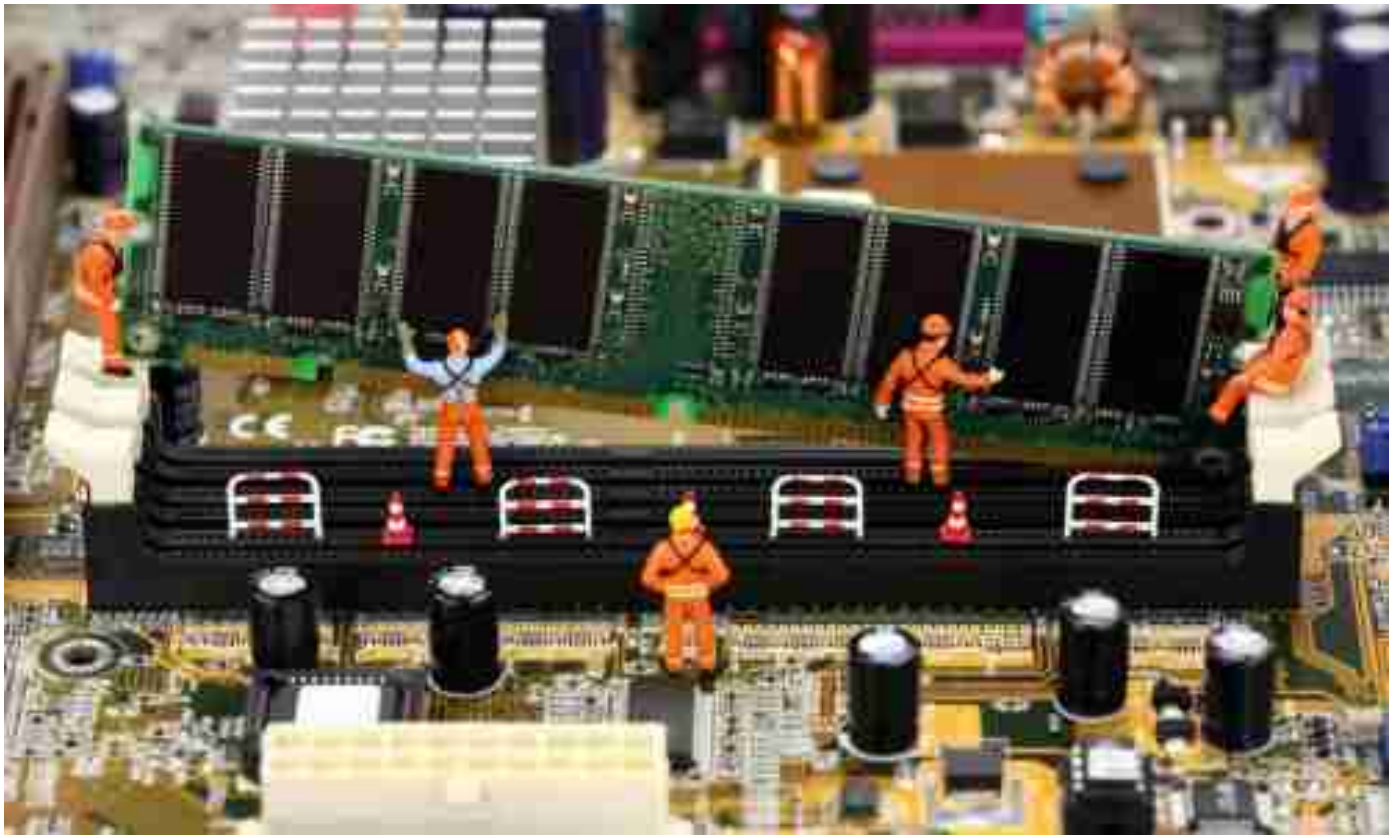
Betriebssysteme

Robert Kaiser

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

9. Speicherverwaltung



https://www.pentagull.co.uk/news/esb-2101-released/

Ausgangspunkt

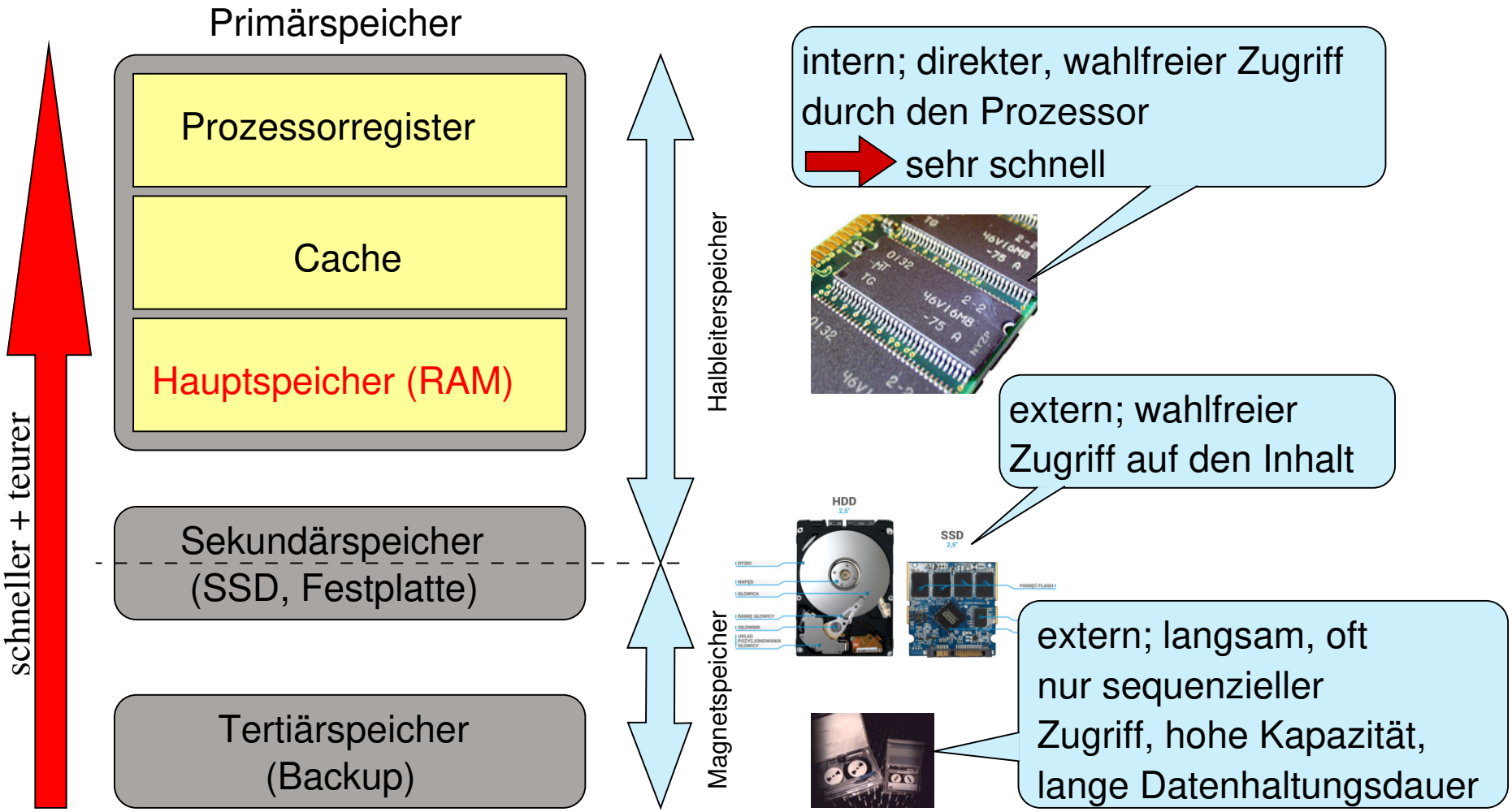
Hochschule RheinMain

- Idealerweise sollte Speicher sein ...
  - groß
  - schnell
  - nicht flüchtig[1]
- In der Realität sind (heute) nicht alle Ziele gleichzeitig zu akzeptablen Preisen mit einem Speichermedium zu erreichen
- Daher: Kombination verschiedener Speicherformen

[1] D.h.: „geht beim Ausschalten nicht verloren“

Die Speicherhierarchie

Hochschule RheinMain



Worum geht es?

Hochschule RheinMain

Definition: Speicherverwalter

Der **Speicherverwalter** ist der Teil des Betriebssystems, der den Arbeitsspeicher verwaltet.

Aufgaben

- Verwaltung von freien und belegten Speicherbereichen.
- Zuweisung von Speicherbereichen an Prozesse, wenn diese sie benötigen (Allokation).
- Freigabe nach Benutzung oder bei Prozessende.
- Durchführung von Auslagerungen von Prozessen (oder Teilen) zwischen Hauptspeicher und Sekundärspeicher bei Speicherengpässen.

In diesem Kapitel wird eine Folge von einfachen bis hochentwickelten Speicherverwaltungsverfahren betrachtet.

Worum geht es?

Hochschule RheinMain

Definition: Speicherverwalter

Der **Speicherverwalter** ist der Teil des Betriebssystems, der den Arbeitsspeicher verwaltet.

Aufgaben

- Verwaltung von freien und belegten Speicherbereichen.
- Zuweisung von Speicherbereichen an Prozesse, wenn diese sie benötigen (Allokation).
- Freigabe nach Benutzung oder bei Prozessende.
- Durchführung von Auslagerungen von Prozessen (oder Teilen) zwischen Hauptspeicher und Sekundärspeicher bei Speicherengpässen.

In diesem Kapitel wird eine Folge von einfachen bis hochentwickelten Speicherverwaltungsverfahren betrachtet.

Gliederung

Speichermodell

- Ein Programm und seine Daten werden während der Ausführung vollständig im Hauptspeicher gehalten[2]
- Die Programm- und Datenobjekte lassen sich in verschiedene Klassen einteilen
- Der Compiler/Linker ordnet die **statischen** (d.h. zur Compile-Zeit bekannten) Objekte je nach Klasse bestimmten „Sektionen“ zu:

Klasse	Lesen	Schreiben	Ausführen	Initialisiert	Sektion
Programmcode	x	-	x	x	`.text`
initialisierte Daten	x	x	-	x	`.data`
uninitialsierte Daten	x	x	-	-	`.bss`
nur-lese Daten	x	-	-	x	`.rodata`

- Für jede Sektion wird ein entsprechend großes, zusammenhängendes Stück Speicher eingeteilt.

[2] Virtueller Speicher / Swapping / Paging vorerst nicht betrachtet

Dynamische Daten

Der restliche (freie) Speicher wird dynamisch, d.h. zur Laufzeit zugeteilt:

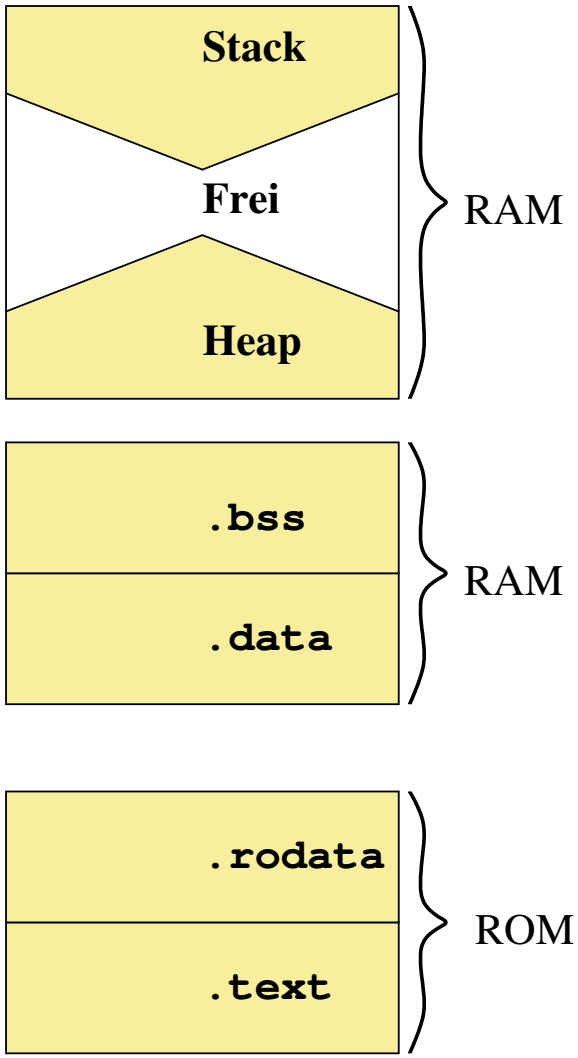
- **Heap**: Dynamischer Speicher
  - Wird in C mit `malloc()` zugeteilt und mit `free()` wieder freigegeben (In C++ mit `new` und `delete`)
  - Wächst nach „oben“, d.h. zu zahlenmäßig höheren Adressen hin
  - Anfang typischerweise direkt im Anschluss an statische Daten (`.bss`)
- **Stack**: Lokale (dynamische) Variablen
  - Sämtliche Automatic-Variablen in C (ausserdem alloca()-Funktion)
  - Auch: Aufrufhierarchie (d.h. Rückkehradressen) und übergebene Parameter)
  - Zuteilung / Freigabe automatisch durch Inkrementieren bzw. Dekrementieren des Stackpointers (z.B. mit `PUSH`- und `POP`-Befehlen)
  - Wächst (i.d.R.) nach „unten“, d.h. zu niedrigeren Adressen hin
  - Anfang typischerweise am oberen Ende des RAM-Speichers

→ **Stack und Heap wachsen einander entgegen. Wenn sie „zusammenstoßen“ geschehen „merkwürdige Dinge“ ...**

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

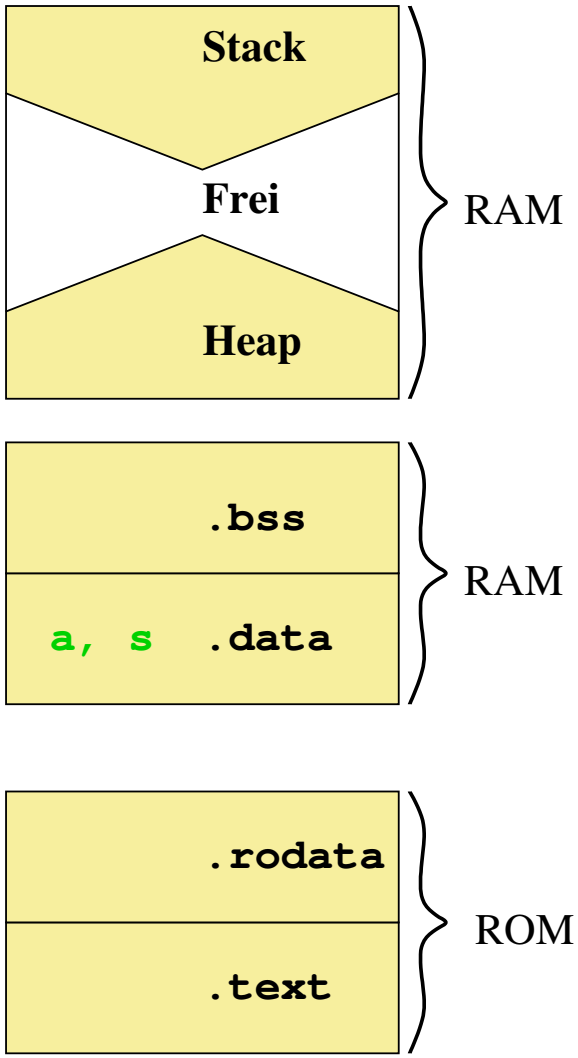


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

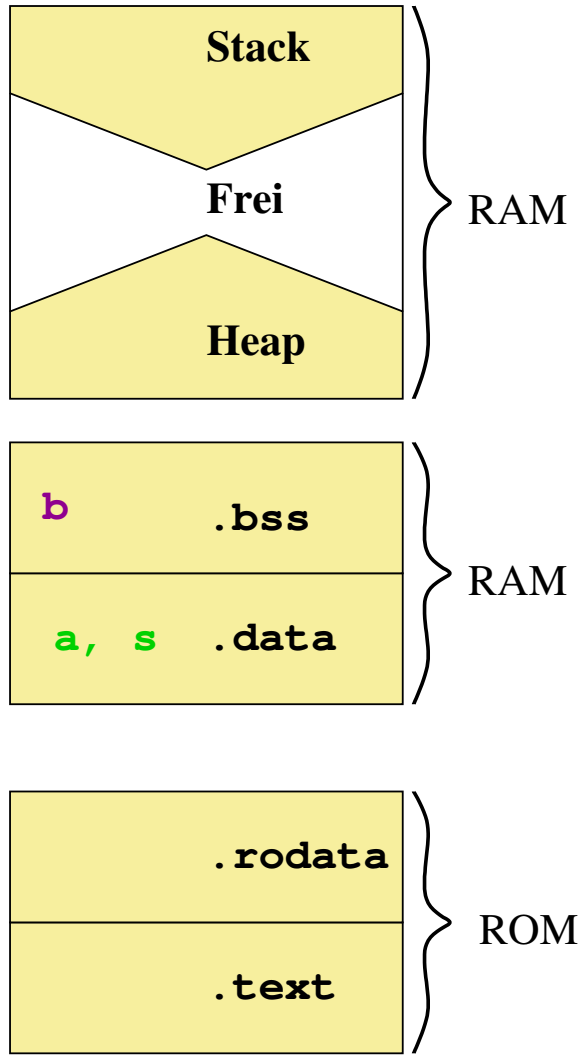


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

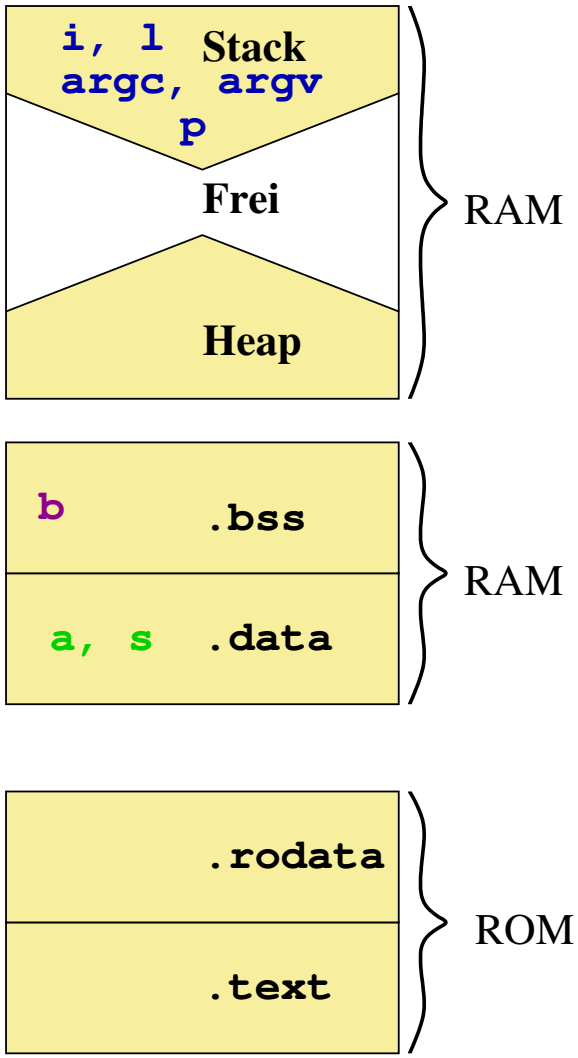


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

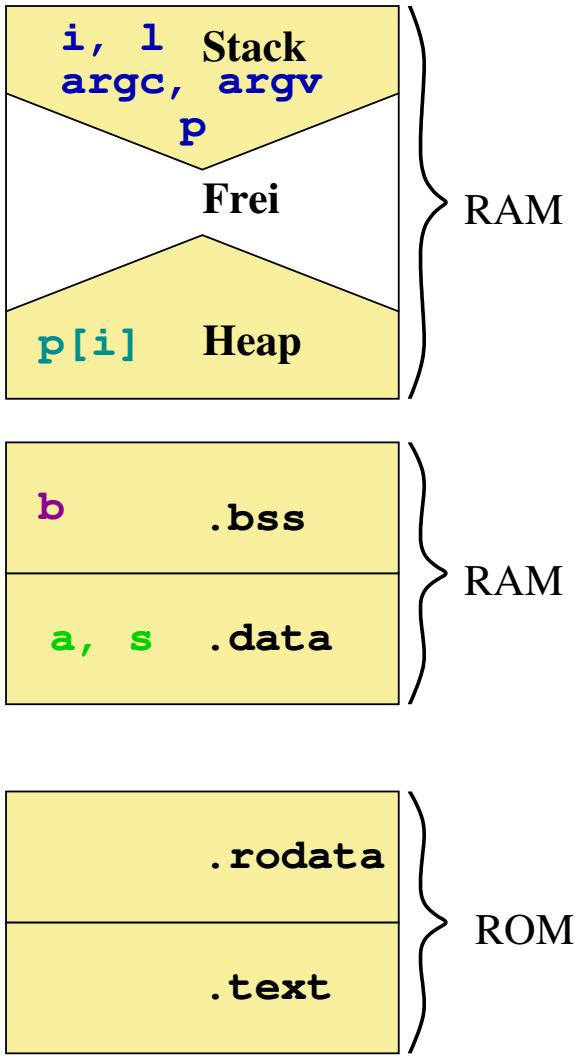


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

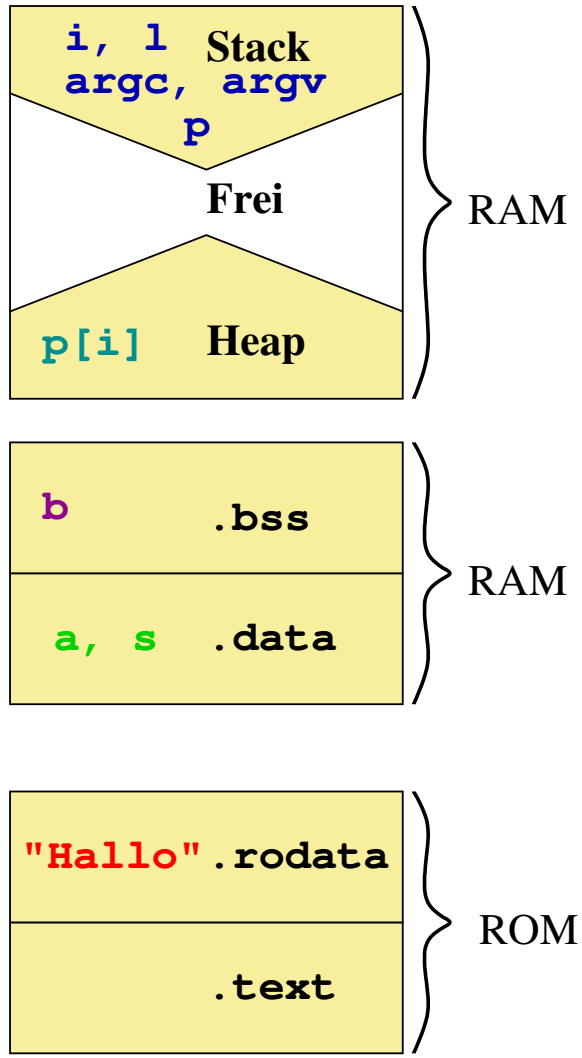


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```

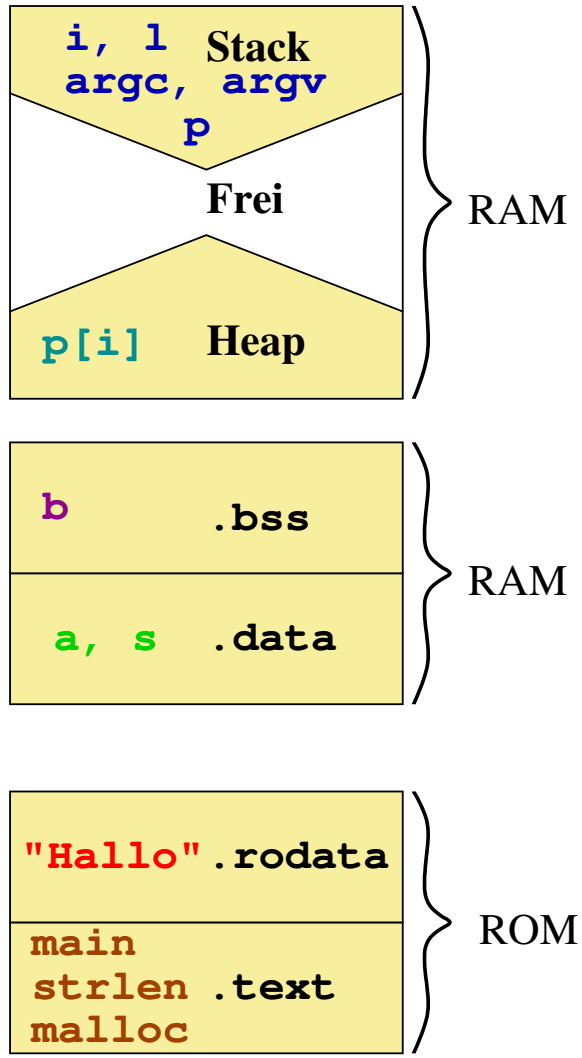


- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Beispiel: Speicherlayout eines C-Programms

- ROM = Nur-Lese Speicher
- RAM = Lese-/Schreib-Speicher

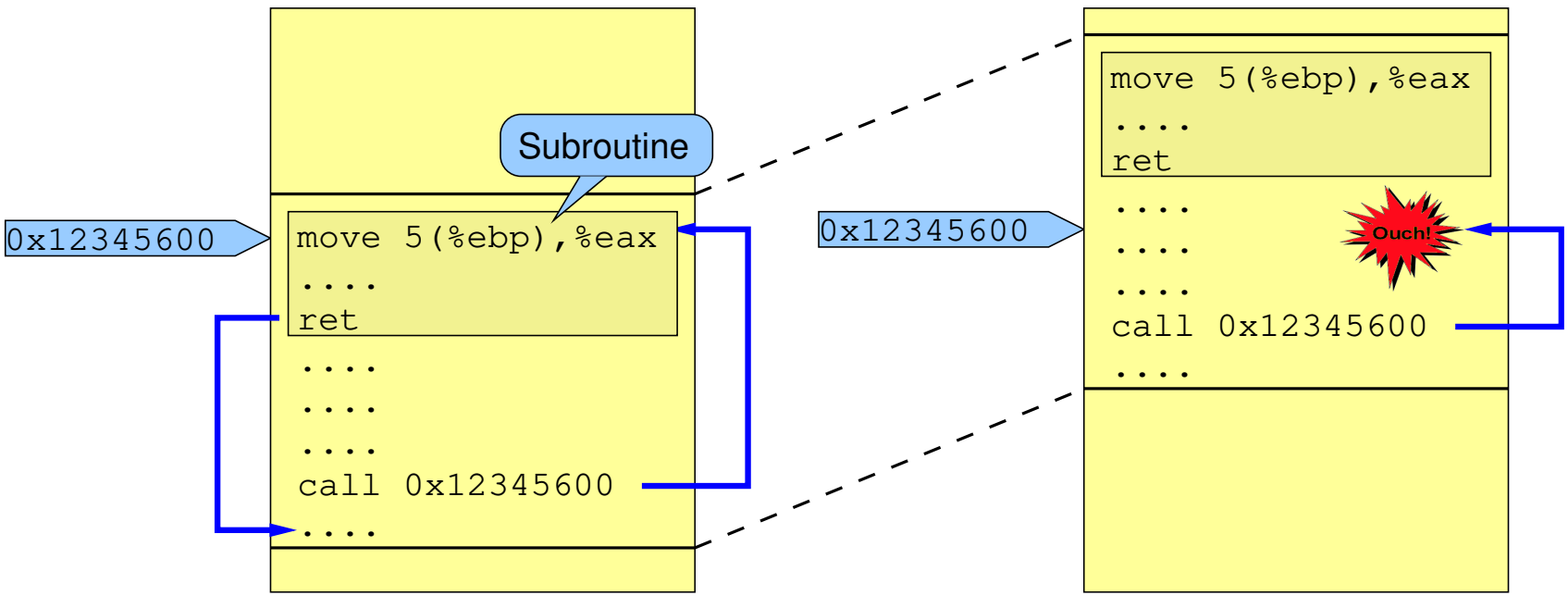
```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i] = s[i];
}
```



- `.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Das Relokationsproblem

- In der Regel sind Programme **nicht** Positionsunabhängig („PIC“=*position independent Code*, „PID“ = *position independent Data*)



- Mögliche Lösung des Relokationsproblems: Loader addiert die Ladeadresse auf alle Adressen gemäß eines vom Binder erstellten Adressverzeichnisses des Programms.

Anforderungen an Speicherverwaltung

Schutz

- Prozesse sollten nicht unerlaubt auf fremde Speicherbereiche zugreifen können

Gemeinsame Nutzung

- Andererseits soll eine kontrollierte gemeinsame Nutzung von Speicherbereichen möglich sein
  (z.B. 10 gleichzeitige bash-Nutzer → Code nicht 10x laden)

Relokation

- Absolute Adressbezüge im Programmcode z.B. beim Laden in einen (anderen) konkreten Speicherbereich anpassen

Speicherorganisation

- Unterstützung von Programm-Modularisierung durch Segmentierung, abgestufter Schutz z.B. für Daten / Code
- Ein-/Auslagern von Speicherbereichen zwischen Hauptspeicher und Sekundärspeicher

Statische Speicherverwaltung

Definition: Statische Speicherverwaltung

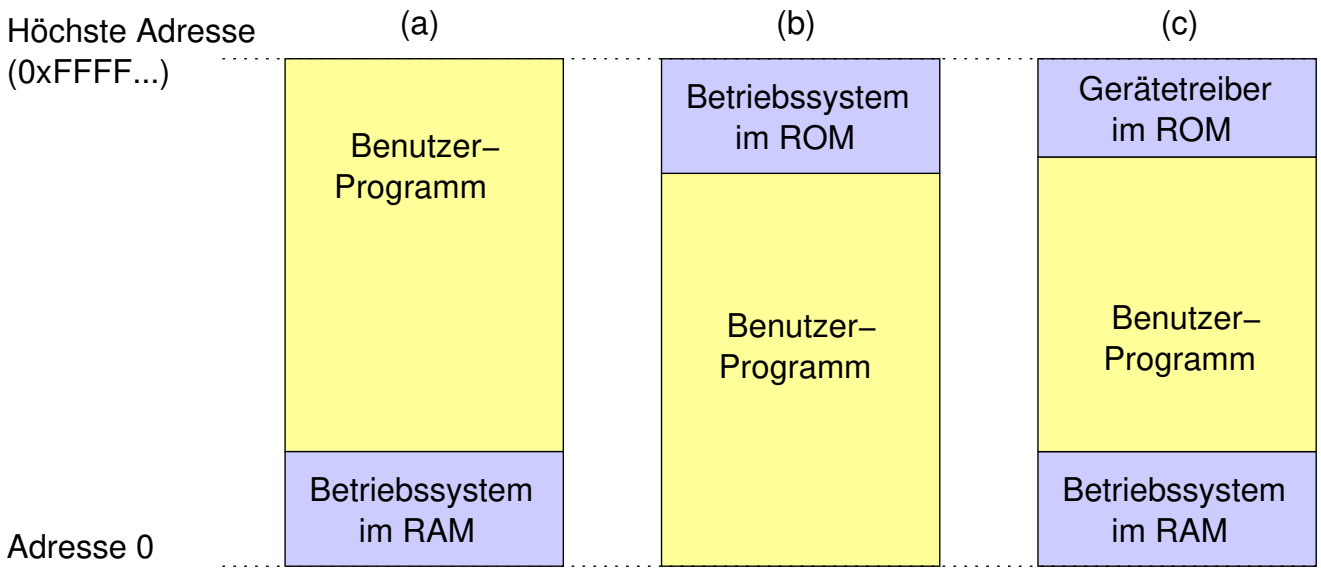
- Verfahren mit fester Zuordnung von Speicher an Prozesse.
- Zuordnung ist keinen Veränderungen während der Laufzeit des Systems oder der Prozesse unterworfen.

- (Nicht-statische Speicherverwaltung heißt auch dynamische Speicherverwaltung (ab Kap. 9.3).)
- Im folgenden für den historischen Einprogramm- und Mehrprogrammbetrieb betrachtet.
- Heute noch in einfachen Echtzeitanwendungen oder bei Mikrocontrollern üblich
  (z.B. Steuerung einfacher Maschinen).

Einprogrammbetrieb

Hochschule RheinMain

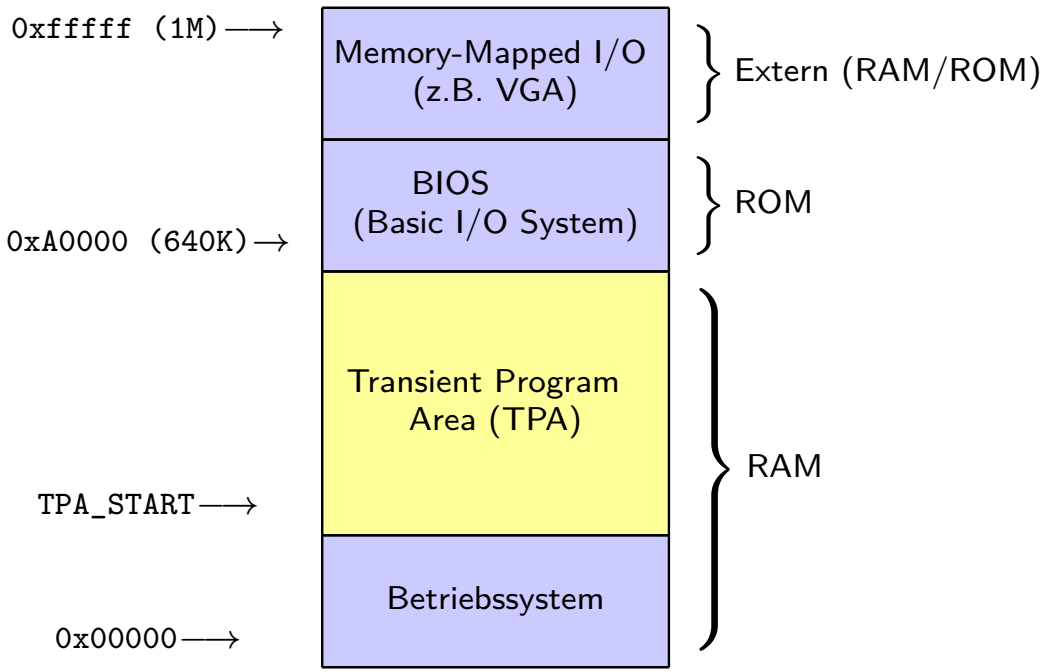
- Nur ein Prozess befindet sich zu einem Zeitpunkt im Speicher. Zuweisung des gesamten Speichers an diesen Prozess.
- Einfache Mikrocomputer: Speicheraufteilung zwischen einem Prozess und dem Betriebssystem.
- Typische Alternativen[3]:



[3] ROM = Read Only Memory, RAM = Random Access Memory (read/write)

Beispiel: IBM PC (i8088)

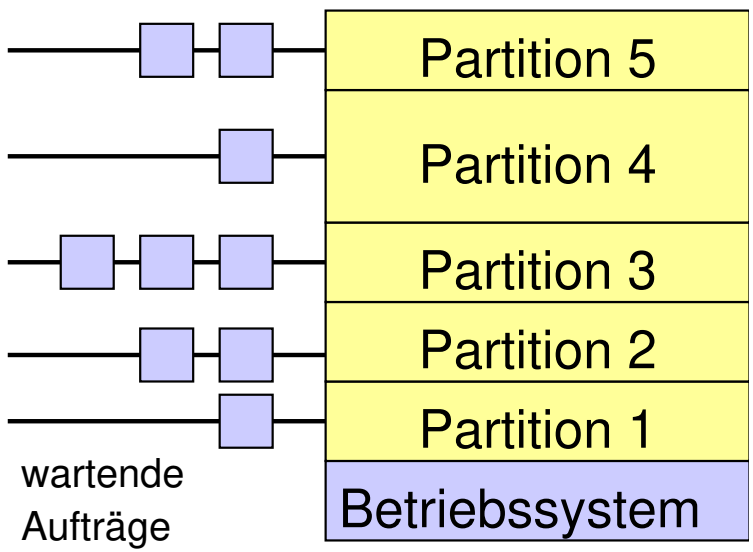
Hochschule RheinMain



Mehrprogrammbetrieb

Hochschule RheinMain

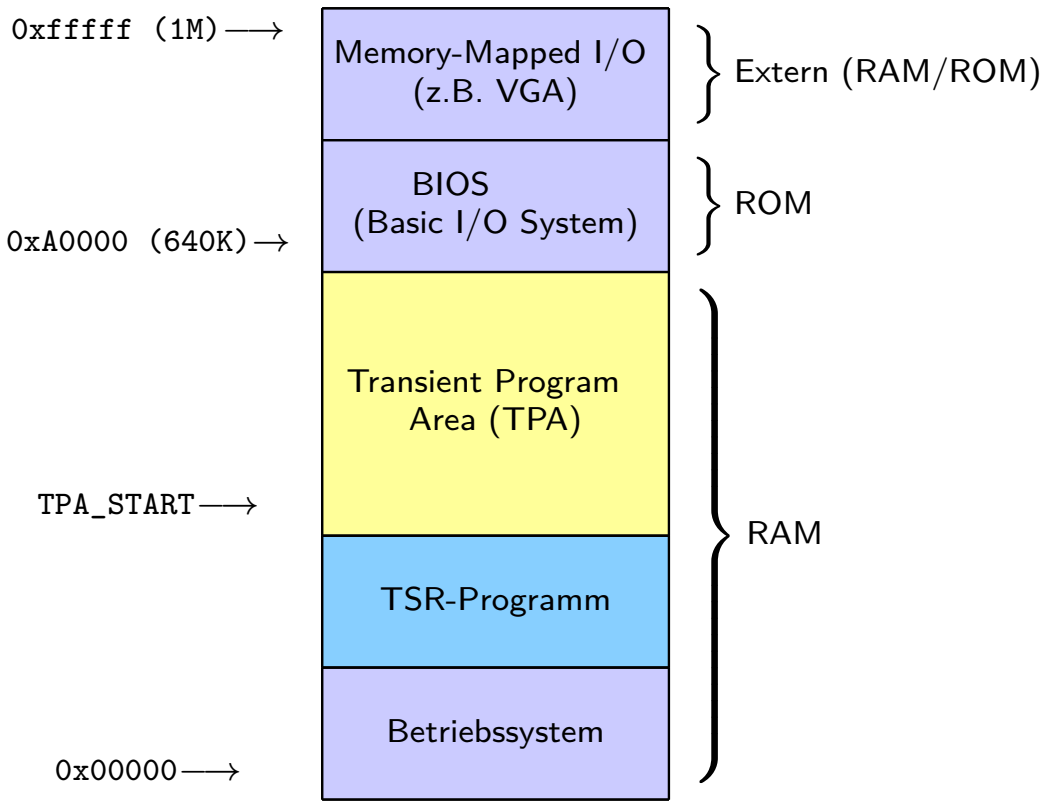
- Ziel: Bessere Ausnutzung der CPU (vgl. Kap. 1).
- Aufteilung des Speichers in feste Partitionen bei Systemstart. In jede Partition kann ein Programm geladen werden.
- Beispiel: IBM Mainframe /360 Betriebssystem OS/MFT (Multiprogramming with a Fixed number of Tasks).
- Programme können für eine Partition gebunden sein und sind dann nur in dieser lauffähig.



„Krudes“ Beispiel: MS-DOS

Hochschule RheinMain

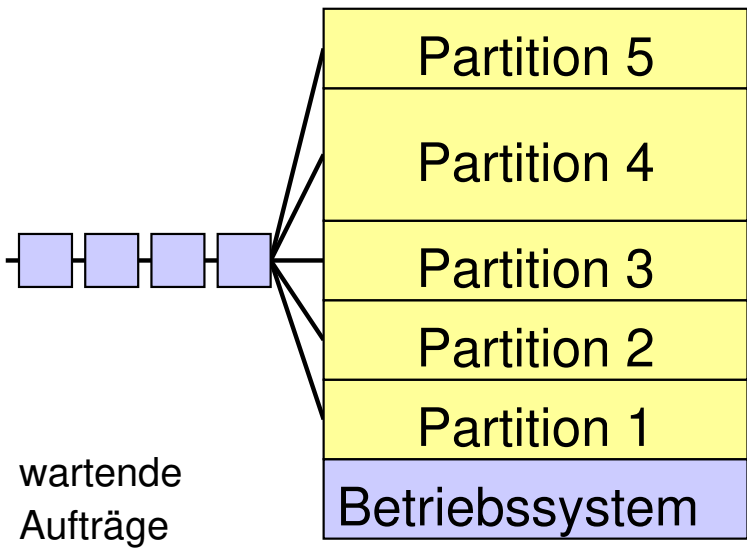
„Mehrprogrammbetrieb“ mit TSR[4]-Programm



[4] terminate, stay resident

Mehrprogrammbetrieb (2)

- Freie Auswahl einer Partition verlangt Lösung des Relokationsproblems (Verschiebbarkeit, s.o.), d.h. Anpassung aller (absoluten) Adressen des Programms in Abhängigkeit von der Ladeadresse.



Mehrprogrammbetrieb (3)

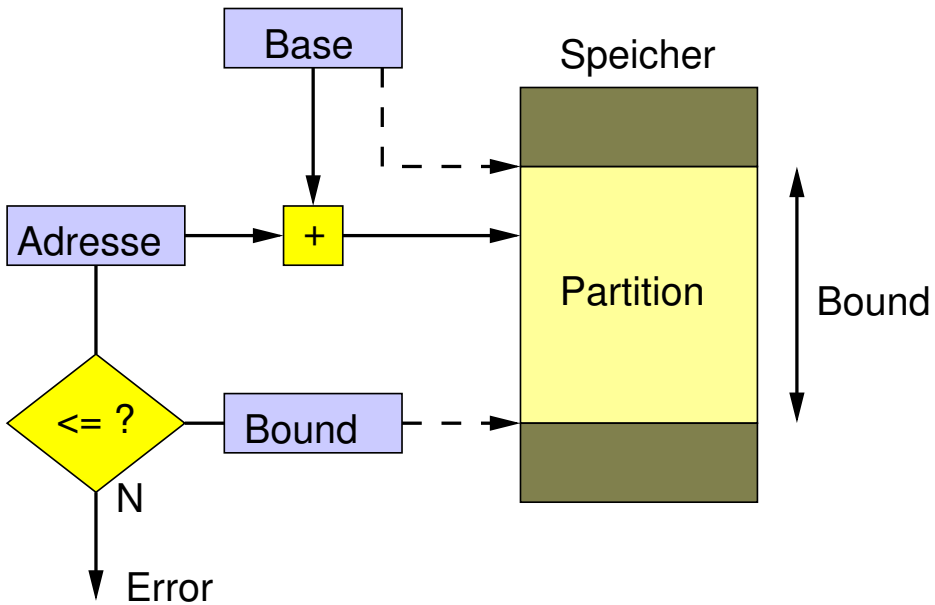
Hochschule RheinMain

- Mehrprogrammbetrieb führt auch zum **Schutzproblem**:
  Programme können aufgrund der absoluten Adressierung Speicherbereiche anderer Benutzer lesen und schreiben (unerwünscht !).
- Lösung des Schutzproblems z.B. in IBM /360:
  - Jedem 2kB-Block des Arbeitsspeichers wird ein 4-Bit Schutzcode (*protection key*) als Schloss zugeordnet.
  - Jeder Prozess besitzt einen ihm im Programmstatuswort (PSW) zugeordneten Schlüssel, der beim Zugriff auf einen Block passen muss, ansonsten: → Abbruch (SIGSEGV).
  - Nur das Betriebssystem kann Schutzcodes von Speicherblöcken und Schlüssel von Prozessen ändern.

⇒ Fazit: Ein Benutzerprozess kann weder andere Benutzerprozesse noch das Betriebssystem stören.

Mehrprogrammbetrieb (4)

- Alternative Vorgehensweise zur Lösung des **Schutzproblems und des Relokationsproblems**:
  Ausstattung der CPU mit zwei zusätzlichen Registern, die **Basisregister** und **Grenzregister** genannt werden (*base and bound register*).



- Weiterer Vorteil eines Basisregisters zur Relokation: Verschieblichkeit des Programms **nach Programmstart** wird möglich.

Swapping

Einführung

- Bei klassischen Timesharing-Systemen gibt es normalerweise nicht genügend Hauptspeicher, um Prozesse aller Benutzer aufzunehmen.
- **Swapping** beinhaltet das Verschieben von Prozessen vom Hauptspeicher auf Sekundärspeicher (Auslagern) und umgekehrt (Einlagern).
- Im Hauptspeicher zu jedem Zeitpunkt nur eine Teilmenge der Prozesse, diese sind aber **vollständig repräsentiert** (im Gegensatz zu virtuellem Speicher, vgl. 9.5).
- Die restlichen (möglicherweise auch rechenwilligen) Prozesse werden in einem Bereich im Hintergrundspeicher abgelegt. Dieser Bereich heißt **Swap-Bereich** (*swap area*).
- Beispiel: Im Unix vor Einführung der virtuellen Speicherverwaltung genutzt.

Variable Partitionen

Definition: variable Partitionen

Mehrprogrammbetrieb kann in **variablen Partitionen** erfolgen, d.h.: Anzahl, Anfangsadresse und Länge der Partitionen und damit der eingelagerten Prozesse ändern sich dynamisch.

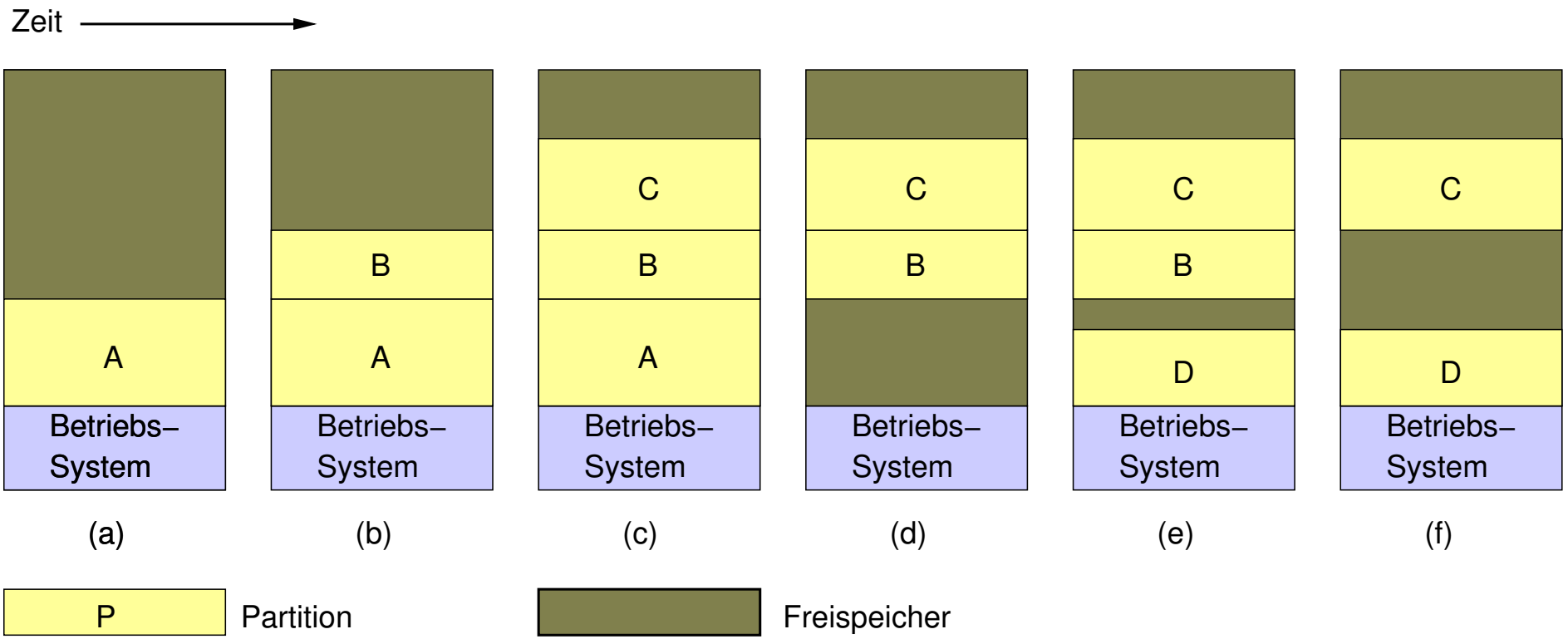
Definition: Segment

Ein zusammenhängender Speicherbereich variabler Länge heißt auch **Speichersegment** oder einfach **Segment**.

- Ziele variabler Partitionen:
  - Anpassung an die tatsächlichen Speicheranforderungen
  - Vermeidung von Verschwendung.

Beispiel

Hochschule RheinMain



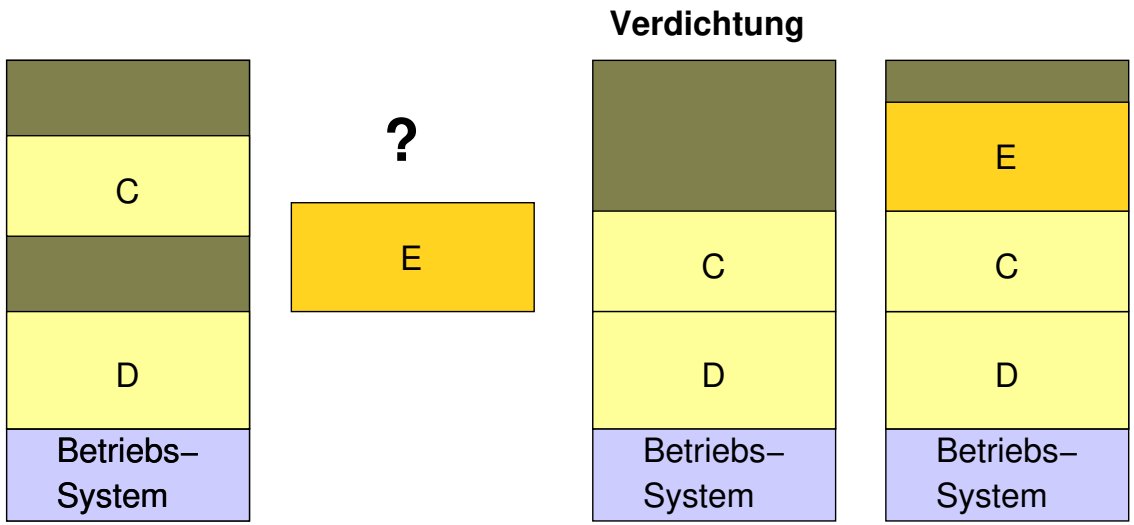
Speicherverdichtung

Hochschule RheinMain

Definition: Externe Fragmentierung

Zerstückelung von Freispeicher zwischen den belegten Segmenten wird **externe Fragmentierung** genannt.

- Abhilfe: Alle unbelegten Speicherbereiche können durch Verschiebung der belegten Segmente zu einem einzigen Bereich zusammengefasst werden. ($\Rightarrow$ **Speicherverdichtung** oder **Kompaktifizierung**).
- Beispiel:

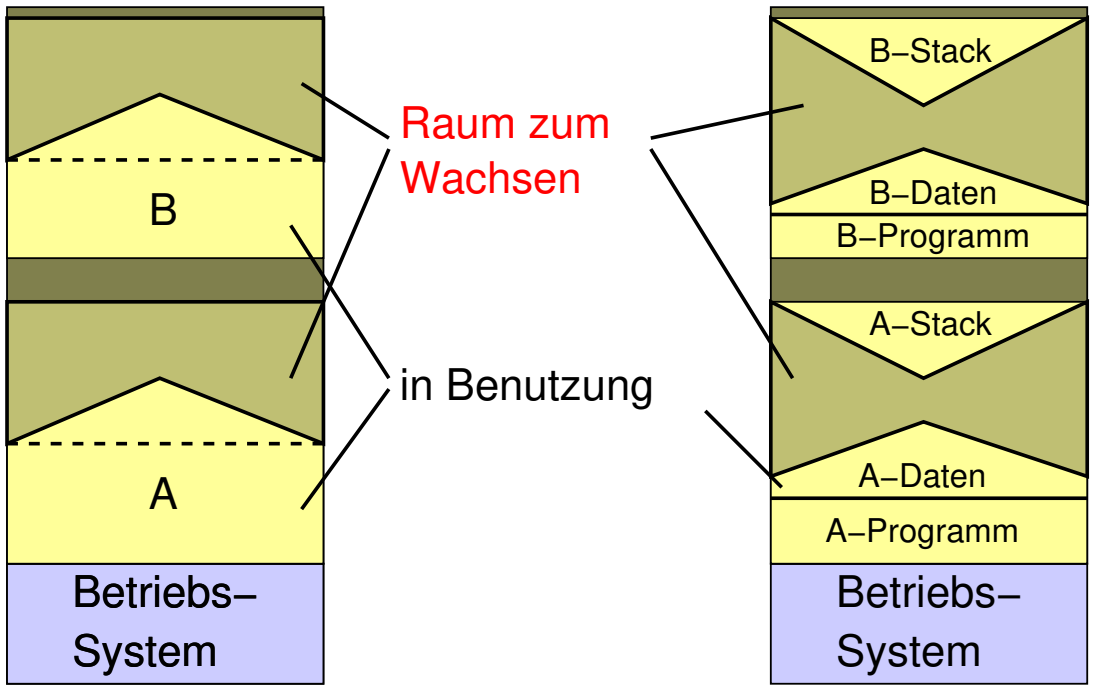


Dynamische Speicheranforderungen (1)

- Speicherbedarf eines Prozesses kann sich im Verlaufe ändern (i.d.R. wachsen)
- Prozess stellt **dynamische Speicheranforderungen**, z.B. zur Aufnahme von neu erzeugten Objekten auf der Halde (dem Heap).
- Behandlung dynamischer Speicheranforderungen:
  - Belegen angrenzender „Löcher“, falls vorhanden.
  - Verschiebung von anderen Prozessen, falls möglich.
  - Auslagern von anderen Prozessen.
  - „Raum zum Wachsen“: Bei der ersten Speicherbelegung vorab zusätzlichen Speicherplatz allokieren.

Dynamische Speicheranforderungen (2)

Beispiele für „Raum zum Wachsen“:



(a) Wachsender Datenbereich eines Prozesses

(b) Wachsender Datenbereich und wachsender Stack

Zusammenfassung: Swapping

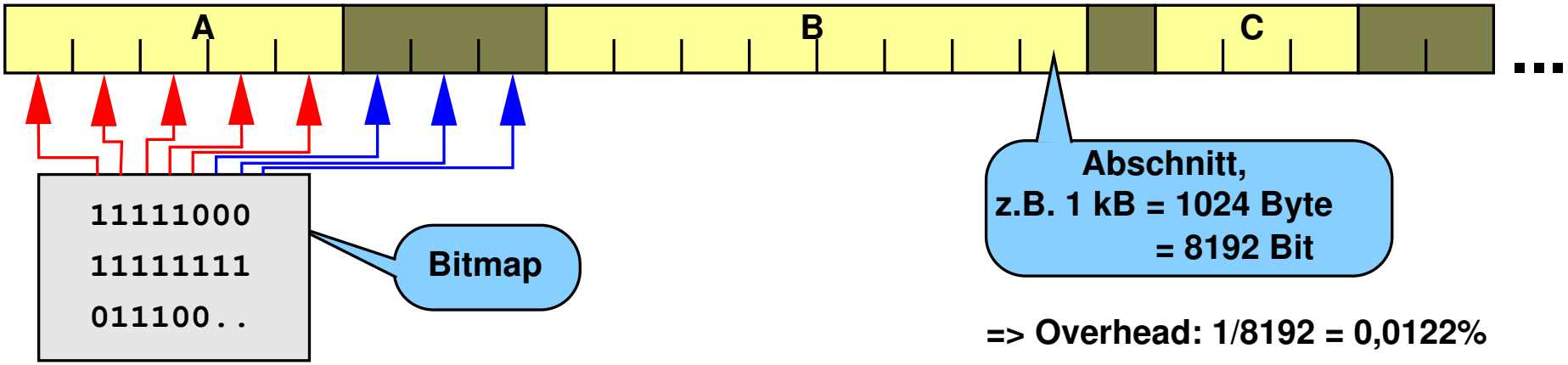
- Im Hauptspeicher wird zu jedem Zeitpunkt nur eine Teilmenge der Prozesse gehalten. Die restlichen Prozesse sind in einen Swap-Bereich auf Platte ausgelagert.
- Alle Prozesse im Hauptspeicher sind vollständig repräsentiert.

Algorithmen der Speicherverwaltung

Verfahren zur Freispeicherverwaltung

- Speicherverwaltung mit Bitmaps (9.4.1)
- Speicherverwaltung mit verketteten Listen (9.4.2)
- Speicherverwaltung mit dem Buddy-System (9.4.3)

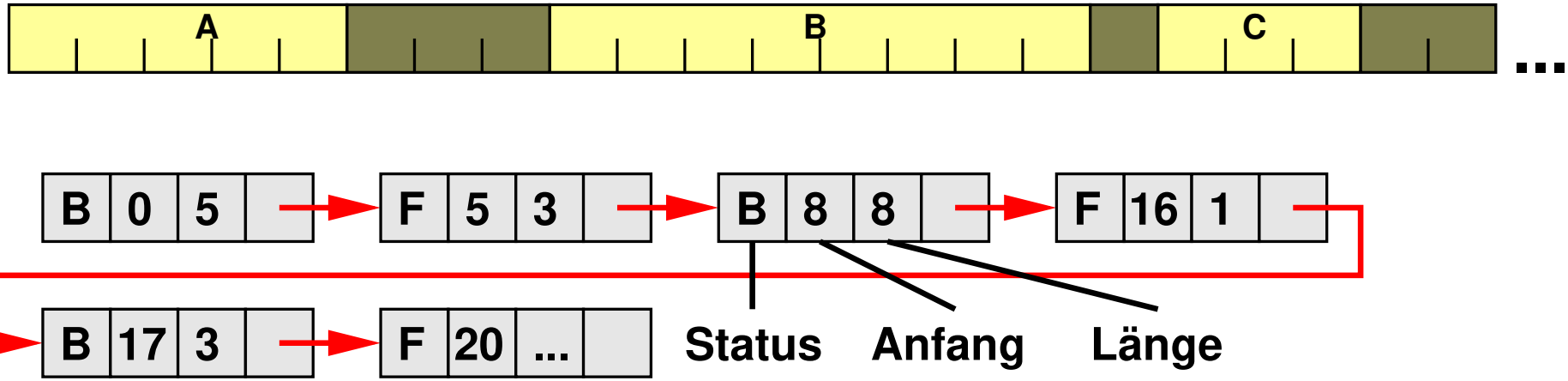
Speicherverwaltung mit Bitmaps



Arbeitsweise

- Der zur Verfügung stehende Speicher wird in Abschnitte fester Länge unterteilt (Granularität einige Worte bis einige KB).
- Jedem Abschnitt wird ein Bit in einer Bitmap zugeordnet:
  0 → Einheit ist frei, 1 → Einheit ist belegt.
- Je kleiner die Einheit desto größer die Bitmap.
- Je größer die Einheit desto mehr Speicher wird verschwendet, da die letzte „angebrochene“ Einheit voll allokiert werden muss.
- Hauptproblem: Belegen eines Speicherbereichs von k Einheiten erfordert Durchsuchen der Bitmap nach einer Folge von k Null-Bits → Aufwändig!
- Für Hauptspeicherverwaltung daher selten eingesetzt.

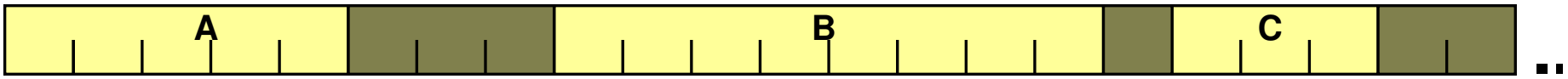
Speicherverwaltung mit verketten Listen (1)

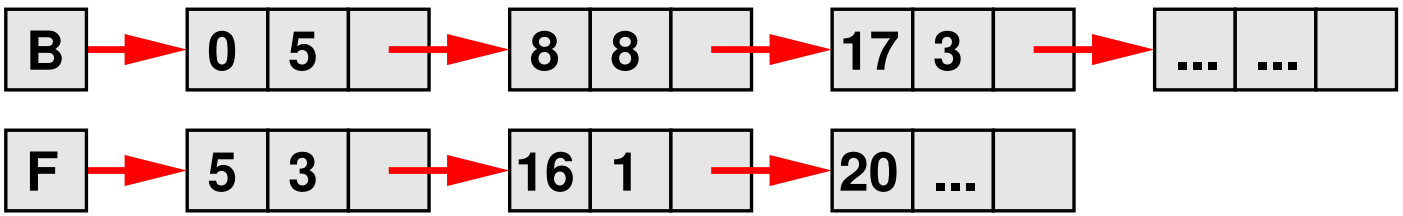


Arbeitsweise

- Jedem belegten und jedem freien Speicherbereich wird ein Listenelement zugeordnet.
- Segmente dürfen variabel lang sein.
- Jedes Listenelement enthält Startadresse und Länge des Segments, sowie den Status (*B*=belegt, *F*=frei).
- Gefundenes freies Segment wird (falls zu groß) aufgespalten.
- Freigegebenes Segment wird ggf. mit ebenfalls freien Nachbarsegmenten „verschmolzen“.

Speicherverwaltung mit verketten Listen (2)





- Die freien Segmente können auch in separater Liste geführt werden („Freiliste“). Die Freiliste kann „in sich selbst“ gehalten werden, d.h. in den verwalteten freien Bereichen. (→ kein zusätzlicher Speicher notwendig).
- Die Segmentliste kann nach Anfangsadressen geordnet sein.
  Vorteil: freiwerdendes Segment kann mit benachbartem freien Bereich zu *einem* freien Segment „verschmolzen“ werden.
- Freiliste kann alternativ nach der Größe des freien Bereichs geordnet sein.
  Vorteil: Vereinfachung beim Suchen nach einem freien Bereich bestimmter Länge.

Strategien zur Speicherallokation

- Gegeben: Speicheranforderung gegebener Größe (z.B. `malloc(groesse)`)
- Aufgabe: Finde ein passendes Segment in der Freiliste.
- Voraussetzung: Liste der Segmente nach Anfangsadressen geordnet.

Definitionen

- **First Fit**: Durchsuche Freiliste, bis ein freies Segment hinreichender Größe gefunden ist. Zerlege den freien Bereich in ein Segment der geforderten Größe und ein neues freies Segment für den übrig bleibenden Rest. Kurze Suchzeit.
- **Rotating First Fit** oder **Next Fit**: Variante von First Fit. Anstelle von vorn zu suchen, beginne Suche die an der nachfolgenden Stelle, an der beim letzten Durchlauf ein Segment belegt wurde.
- **Best Fit**: Durchsuche die gesamte Liste, wähle das kleinste für die Anforderung gerade ausreichende freie Segment und spalte dieses. Langsam. Neigt zur ungewollten Erzeugung vieler kleiner Freisegmente (*Fragmentierung*).
- **Worst Fit**: Wähle das größte freie Segment und spalte dieses. Vermeidet Entstehung vieler kleiner Freisegmente, aber keine gute Idee.

Analyse (1)

Hochschule RheinMain

- Speicherverwaltung mit Bitmaps oder verketteten Listen führt zu **externer Fragmentierung**.

50%-Regel

50%-Regel (Knuth): Sind im Mittel n Prozesse im Speicher, so gibt es im Mittel $\frac{n}{2}$ freie Bereiche.

- Ursache ist, dass zwei benachbarte Freibereiche zu einem zusammengefasst werden, belegte Bereiche hingegen nicht.

→ Im Mittel gibt es doppelt so viele belegte Bereiche, wie Freibereiche.

Analyse (2)

Hochschule RheinMain

Damit...

Ungenutzte-Speicher-Regel

Ungenutzte-Speicher-Regel: Sei s die mittlere Größe eines Prozesses, und $k \cdot s$ sei die mittlere Größe eines Freibereichs für einen Faktor k, der abhängig vom Allokationsalgorithmus ist.
Dann gilt für den ungenutzten Anteil f des Speichers:

$$f = \frac{ungenutzt}{gesamt\,belegt}$$

$$f = \frac{\frac{n}{2} \cdot k \cdot s}{\frac{n}{2} \cdot k \cdot s + n \cdot s}$$

$$f = \frac{k}{k+2}$$

Beispiel: Sei $k = 0,5$. Dann werden 20% des Speicherplatzes für Freibereiche verbraucht.

Speicherverwaltung mit dem Buddy-System

Hochschule RheinMain

Arbeitsweise:

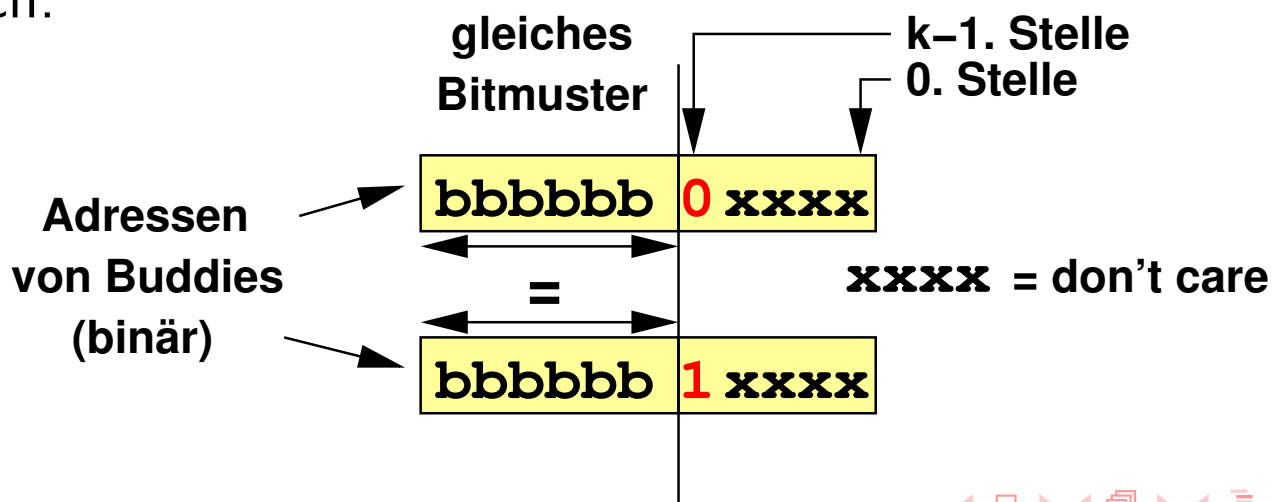
- Freie und belegte Speicherbereiche haben ausschließlich Längen, die 2er Potenzen sind.
- Da nur bestimmte Längen auftreten, wird von **Speicherblöcken** (anstelle von Segmenten) gesprochen.
- Die maximale Blocklänge L_{max} entspricht der größten 2er Potenz kleiner oder gleich dem verfügbaren Speicher, z.B. $L_{max} = 2^{22}$ bei 4 MB oder 6 MB[5]
- Es kann eine minimale Blocklänge größer $2^0 = 1$ Byte geben, z.B. $L_{min} = 2^6 = 64$ Bytes.
- Eine Speicheranforderung wird auf die nächst mögliche 2er Potenz aufgerundet, z.B.: angefordert 70 KB, belegt 128 KB
- Für jede der verwalteten Blocklängen zwischen L_{min} und L_{max} wird eine separate Freiliste gehalten.

[5] 4 MB$= 4096 \cdot 1024 = 2^{12} \cdot 2^{10} = 2^{22}$.

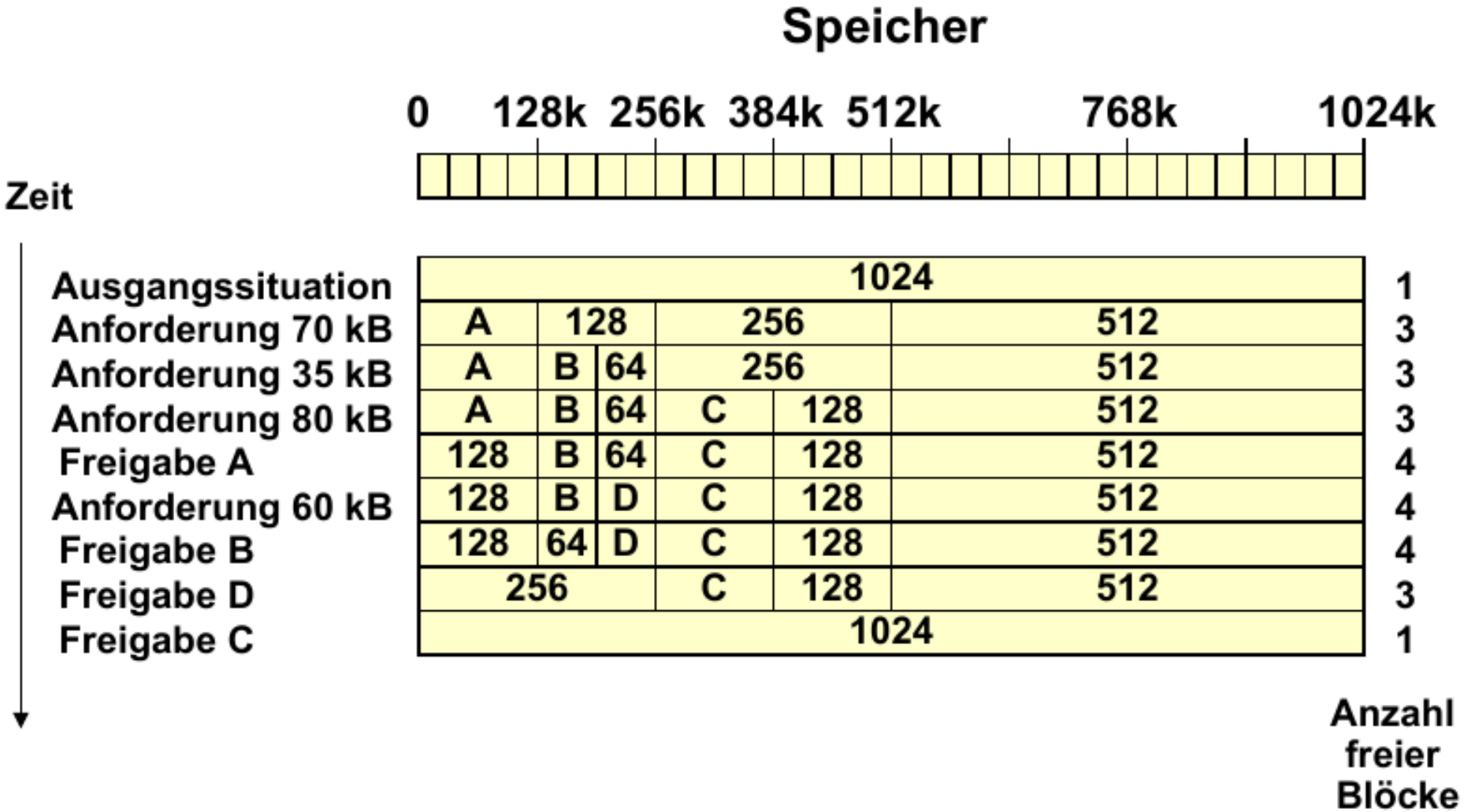
Buddy-System Arbeitsweise (2)

Hochschule RheinMain

- Kein freier Block der gesuchten Länge: Halbiere einen Block der nächsthöheren 2er Potenz in 2 freie Blöcke der gesuchten Länge. Diese werden ***Buddies*** genannt (deutsch: „Kumpel“).
- Freigabe eines Blocks der Länge 2^k: wenn sein Buddy bereits ein freier Block ist, mit diesem zu einem neuen freien Block der Länge 2^{k+1} verschmelzen.
- Es lassen sich **nicht** zwei beliebige benachbarte Blöcke gleicher Länge 2^k zusammenfassen!
- Die Entscheidung ist einfach aufgrund der Adresse der freien Blöcke möglich:



Buddy-System Arbeitsweise: Beispiel



Fazit Buddy-System

Vorteile:

- Bei Speicheranforderung insgesamt schnelles Auffinden oder Erzeugen eines freien Blocks.
- Bei Freigabe muss nur *eine* Freiliste durchsucht werden. Falls Buddy bereits frei ist, einfaches Verschmelzen zu einem größeren Block.

Nachteile

- Ineffizient in der Speicherausnutzung, da immer auf die nächste 2er Potenz aufgerundet werden muss.
- Diese Verschwendung wird als **interne Fragmentierung** bezeichnet, da sie innerhalb des allokierten Speicherbereichs auftritt (im Gegensatz zur **externen Fragmentierung**).

Zusammenfassung: Speicherverwaltungsalg.

Hochschule RheinMain

- Es wurden Verfahren zur Freispeicherverwaltung besprochen:
  - Verwaltung mittels Bitmaps
  - Verwaltung mittels verketteter Listen
  - Buddy-System
- Es wurden Strategien zur Speicherallokation besprochen:
  - First Fit
  - Rotating First Fit / Next Fit
  - Best Fit
- Wichtige Begriffe und Regeln:
  - Externe und interne Fragmentierung
  - 50% Regel, ungenutzte Speicher Regel

Virtueller Speicher

Hochschule RheinMain

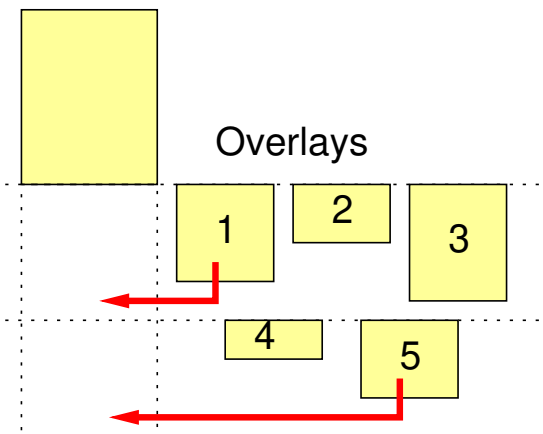
Problem:

- Die Größe eines einzelnen Prozesses kann die Größe des insgesamt zur Verfügung stehenden Speichers übersteigen.

→ Swapping (d.h. Auslagern ganzer Prozesse) ist keine Lösung mehr.

Historische Lösung: *Overlays* (Überlagerungen)

- Funktionsteile werden bei Bedarf (D.h. auf explizite Anfrage des Prozesses) dynamisch vom Hintergrundspeicher nachgeladen.
- Overlay-Struktur des Programms wird vom Programmierer vorgeplant und vom Binder erzeugt.



heute z.T. wieder aktuell bei Echtzeitsystemen (wegen planbarer Nachlade-Zeitpunkte)

Modernerer Ansatz: Virtueller Speicher

- Den Prozessen wird die Existenz eines großen Speicherraums vorgespiegelt.
- Dieser **virtuelle Speicher** wird vom Betriebssystem auf Basis von realem Hauptspeicher und Hintergrundspeicher realisiert.
- Das Betriebssystem hält dazu die „gerade in Benutzung“ befindlichen Teile eines Programms im Hauptspeicher, Rest auf dem Hintergrundspeicher.
- Ausgelagerte Teile werden bei Bedarf (*on Demand*) eingelagert.
- Für das Programm ist der Unterschied nicht erkennbar, daher sind keine besonderen Vorkehrungen bei der Programmierung nötig.

Definition: virtueller Adressraum

Die von den Instruktionen eines Programms generierten Adressen werden als **virtuelle Adressen** bezeichnet. Sie bilden zusammen den **virtuellen Adressraum** des Prozesses, der das Programm ausführt.

- Zwei Ansätze:
  - **Paging**: eindimensionaler virtueller Speicher. Heute am stärksten verbreitet.
  - **Segmentierung**: zweidimensionaler virtueller Speicher.

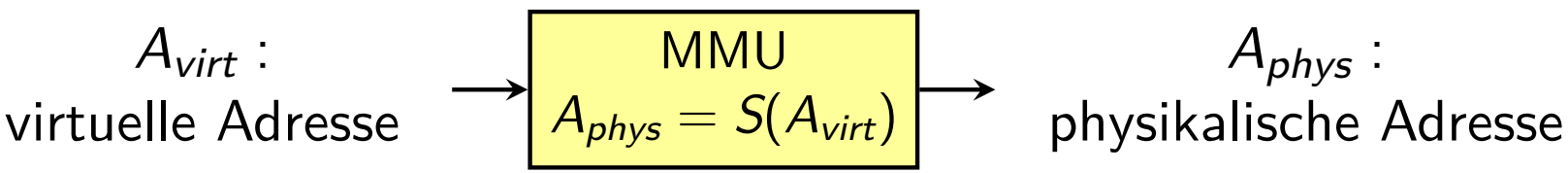
Speicherverwaltungseinheit (MMU)

Hochschule RheinMain

Defintion MMU

Die **Speicherverwaltungseinheit** (*Memory Management Unit*, **MMU**) wandelt virtuelle Adressen in reale (physikalische) Adressen um.

- Abbildung wird abstrakt auch als **Speicherfunktion** bezeichnet

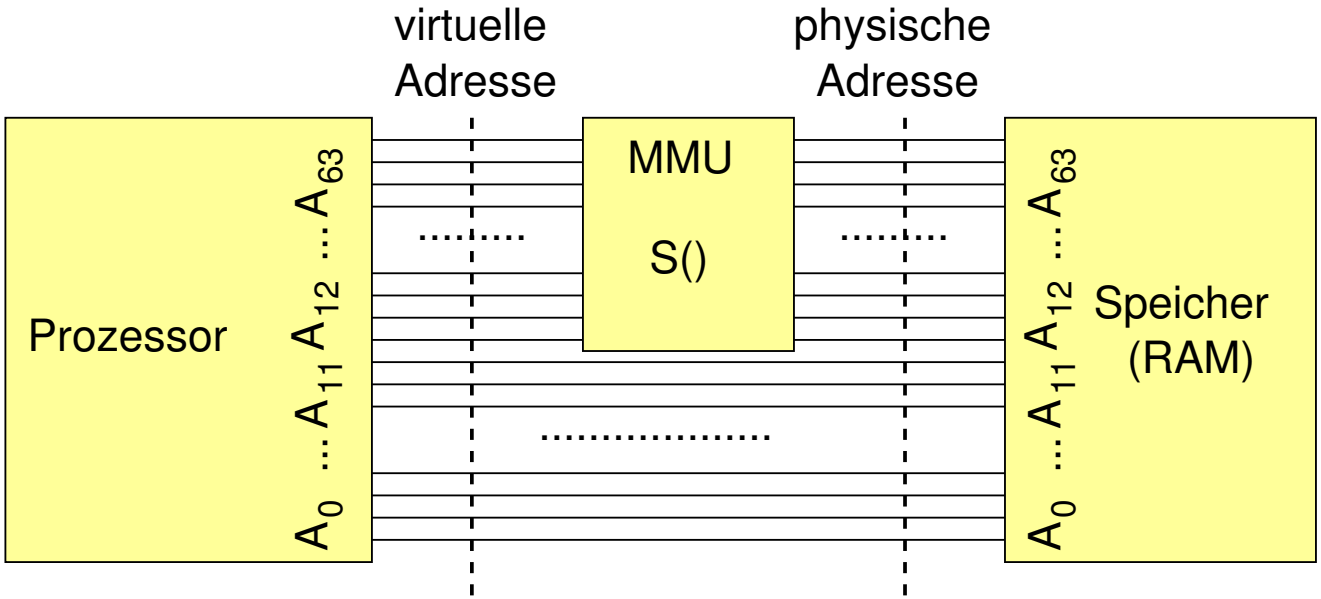


N.B. $A_{virt} \rightarrow A_{phys}$ ist eindeutig, die **Umkehrung aber nicht**!

Technische Umsetzung

Hochschule RheinMain

- Nur die höherwertigen $n - m$ Adressbits[6] werden umgewandelt:



⇒ Alle virtuellen Adressen, die sich nur in ihren m niederwertigen Adressbits unterscheiden, gehören gemeinsam zu einer **Seite** und werden durch die MMU auf denselben **Seitenrahmen** abgebildet.

- Die Größe der Seiten sowie der Seitenrahmen ist dabei 2^m. (im Beispiel: $2^{12} = 4096$)

[6] Beispiel hier: $A_{12} \ldots A_{63} \rightarrow n - m = 52,\ n = 64, m = 12$

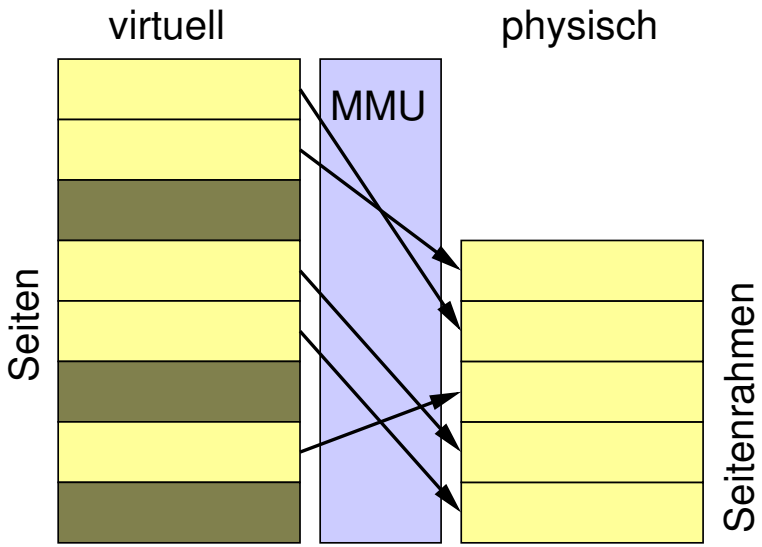
Grundbegriffe des Paging

Hochschule RheinMain

Definition: Seiten, Seitenrahmen

- Der virtuelle Adressraum ist in Einheiten fester Länge unterteilt, diese heißen **Seiten (*Pages*)**.
- Der physikalische Speicher wird ebenfalls in Einheiten dieser Länge unterteilt, diese heißen **Kacheln** oder **Seitenrahmen (*Page Frames*)**.

- Der Transfer zwischen Hauptspeicher und Hintergrundspeicher erfolgt grundsätzlich in Einheiten von Seiten.



Heute typische Größen:

- Virtueller Adressraum:
  - 2^{32} Bytes (4 GB) (32-bit x86)
  - 2^{48} Bytes (256 TB) (64-bit AMD64, Intel64)
- Physikalischer Speicher:
  - i.d.R. kleiner (z.B. 4 - 16 GB für Arbeitsplatzrechner)
- Seiten / Seitenrahmen: 4 KB oder 8 KB

Grundbegriffe des Paging (2)

Definition: Seitentabelle

- Die abstrakte Speicherfunktion $S()$ kann bildlich durch eine Tabelle repräsentiert werden, bezeichnet als **Seitentabelle**.
- Die Seitentabelle ordnet jeder Seite des virtuellen Adressraums einen Seitenrahmen des Hauptspeichers **oder** ein Kennzeichen zu, dass die Seite sich nicht im Hauptspeicher befindet (nicht eingelagert ist). Dieses Kennzeichen wird **Present/Absent-Bit** genannt.

- Dereferenziert ein Programm eine virtuelle Adresse, deren zugehörige Seite nicht eingelagert ist, so erzeugt die MMU eine Unterbrechung der CPU. Diese Situation wird als **Seitenfehler (*Page Fault*)** bezeichnet.

MMU-Funktionsprinzip

Hochschule RheinMain

Beispiel: MMU, 16 Seiten zu 4 KB

15	0101	1
14	0000	0
13	0000	0
12	1001	0
11	0000	0
	.....	
6	0000	0
5	0011	1
4	0100	1
3	0000	1
2	0110	1
1	0001	1
0	0010	1

virtuelle Adresse: 8192

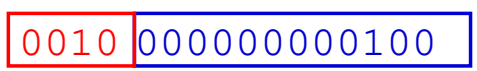
MMU-Funktionsprinzip

Hochschule RheinMain

Beispiel: MMU, 16 Seiten zu 4 KB

15	0101	1
14	0000	0
13	0000	0
12	1001	0
11	0000	0
	.....	
6	0000	0
5	0011	1
4	0100	1
3	0000	1
2	0110	1
1	0001	1
0	0010	1

virtuelle Adresse: 8192



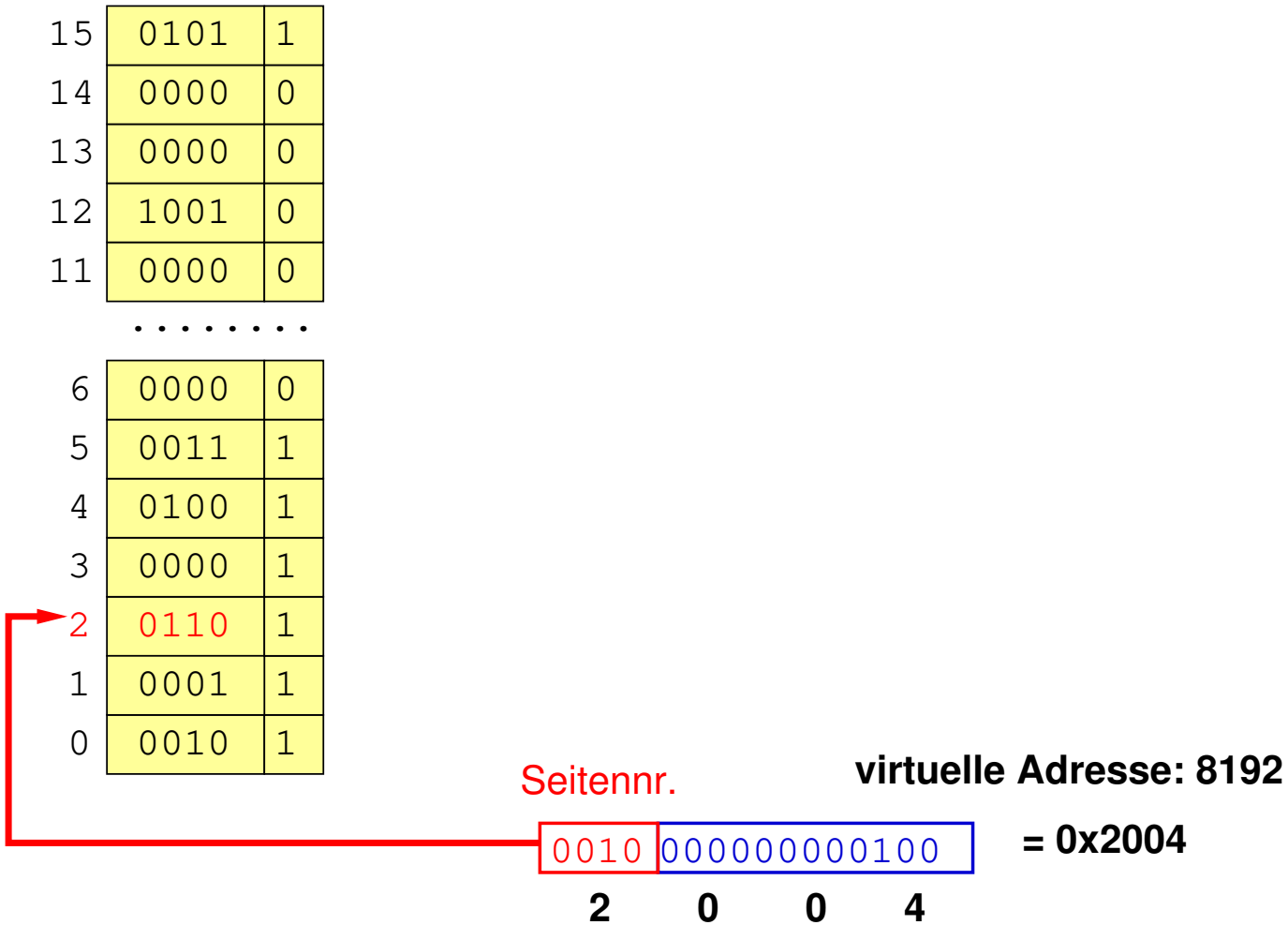
= 0x2004

2 0 0 4

MMU-Funktionsprinzip

Hochschule RheinMain

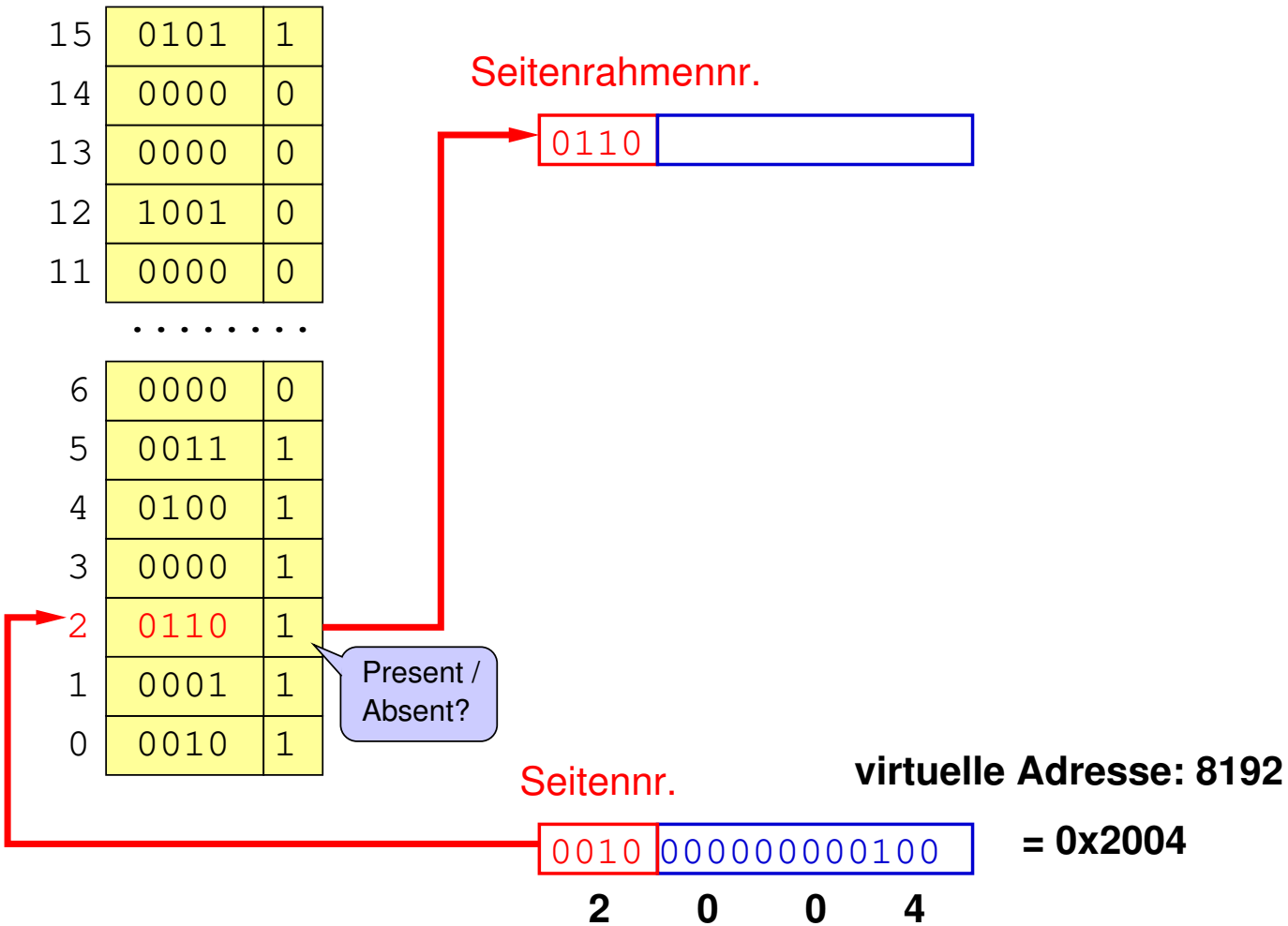
Beispiel: MMU, 16 Seiten zu 4 KB



MMU-Funktionsprinzip

Hochschule RheinMain

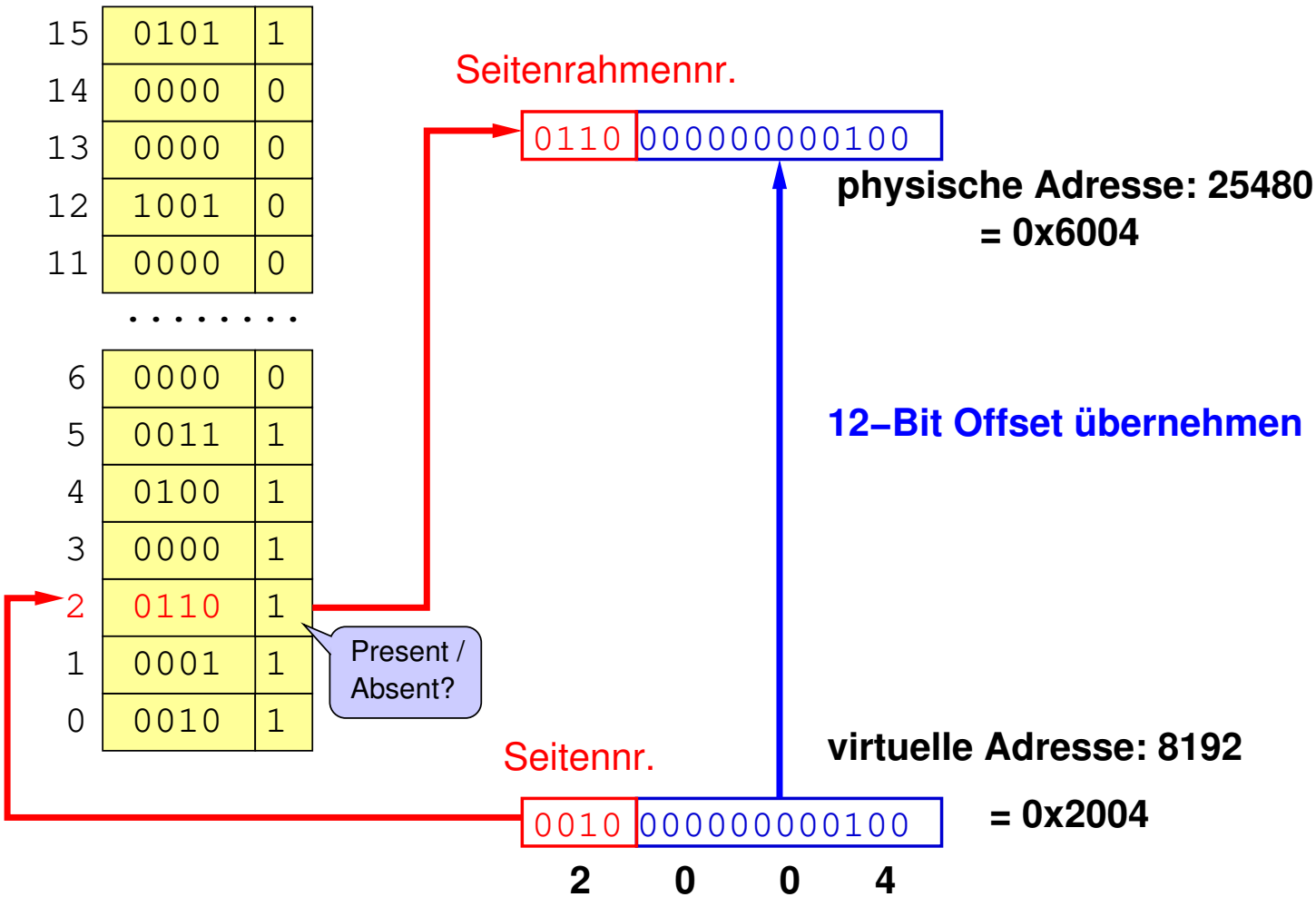
Beispiel: MMU, 16 Seiten zu 4 KB



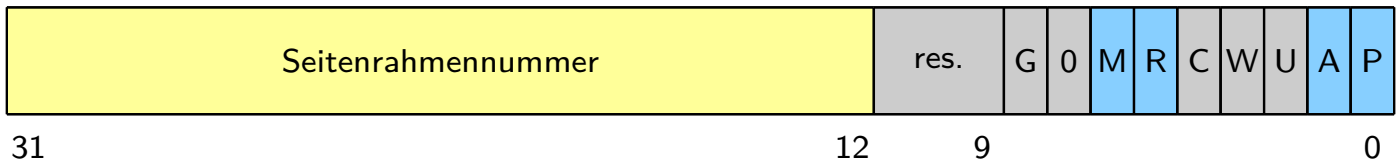
MMU-Funktionsprinzip

Hochschule RheinMain

Beispiel: MMU, 16 Seiten zu 4 KB



Seitentabelleneintrag

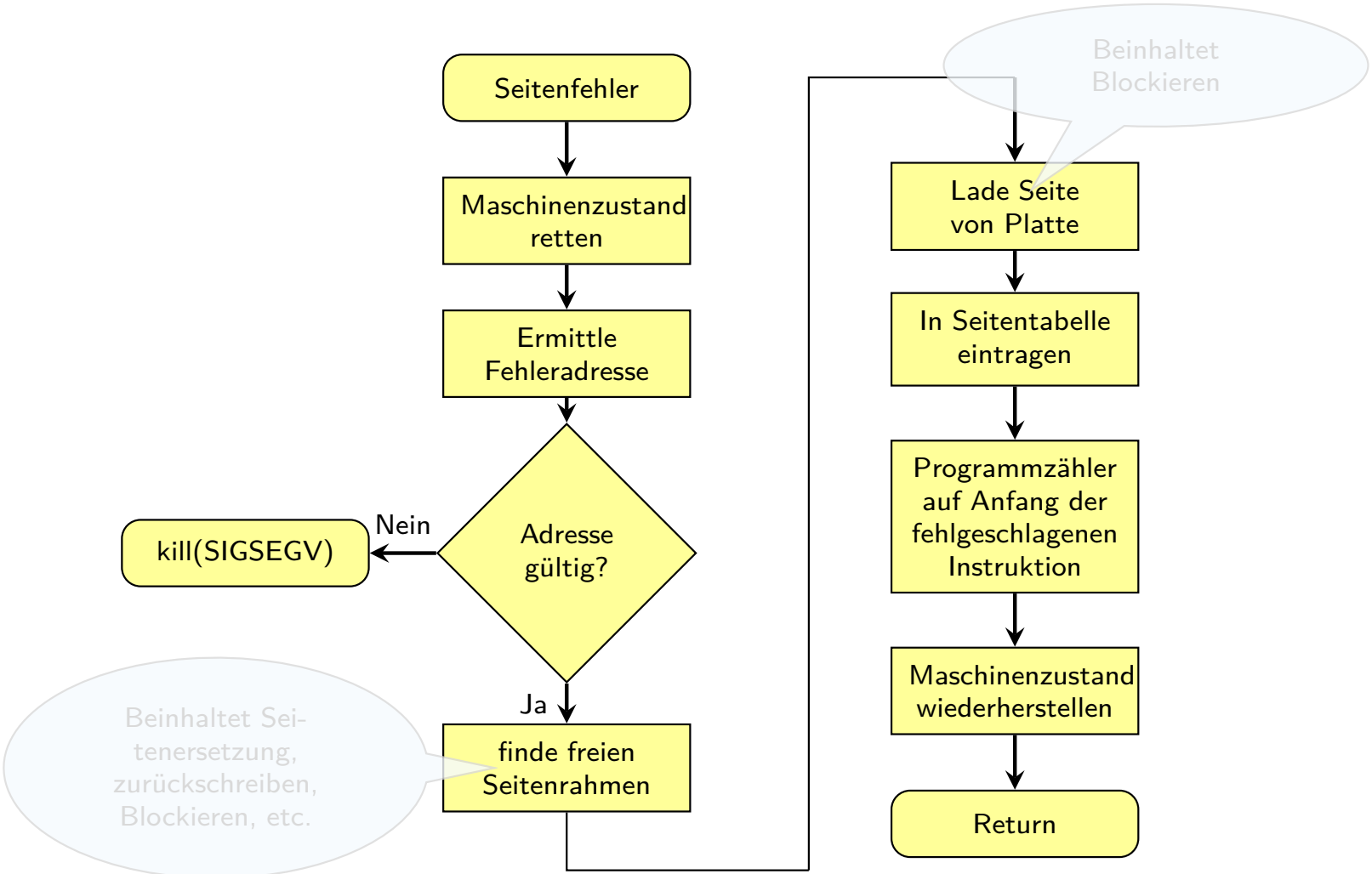


- Neben der Seitenrahmennummer und dem Present/Absent-Bit („P“) enthält ein Seitenrahmentabelleneintrag in der Regel weitere Informationen:
  - R Referenced-Flag: auf die Seite wurde lesend und / oder schreibend zugegriffen
  - M Modified-Flag: Seite wurde verändert; „dirty bit“
  - A Access Rights: Zugriffsrechte: Seite ist lesbar / schreibbar / ausführbar
  - Weitere (z.B. non-cacheable (C), cache write through (W), ...)

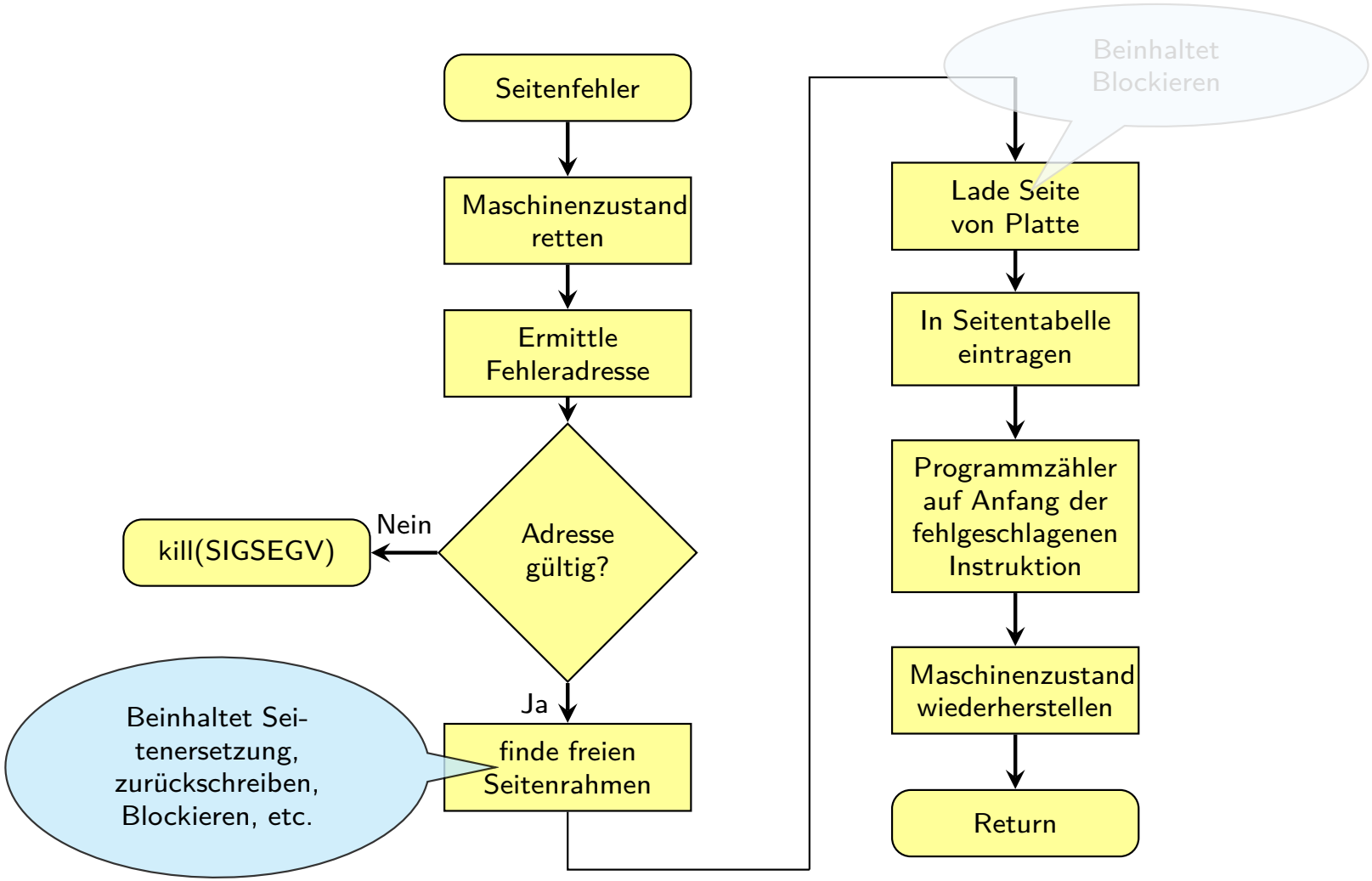
Behandlung eines Seitenfehlers (1)

- Behandlung eines Seitenfehlers besteht prinzipiell in der **Einlagerung der fehlenden Seite** vom Hintergrundspeicher in einen freien Seitenrahmen des Arbeitsspeichers.
- Einlagerung als Reaktion auf einen Seitenfehler wird als ***Demand Paging*** bezeichnet.
- Liegt kein freier Seitenrahmen vor, muss ein solcher gewonnen werden. I.d.R. hält das Betriebssystem einen Vorrat an freien Rahmen vor. (Algorithmen für die Auswahl zu ersetzender Seiten werden detailliert in 9.6 besprochen).
- Falls die zu ersetzende Seite während ihrer Lebenszeit im Speicher verändert wurde (als *Dirty Page* bezeichnet), wird sie auf die Platte zurückgeschrieben (*Update*). Der belegte Seitenrahmen ist danach frei.
- Falls die zu ersetzende Seite nicht verändert wurde (enthält z.B. Programmcode), ist kein Zurückschreiben erforderlich.

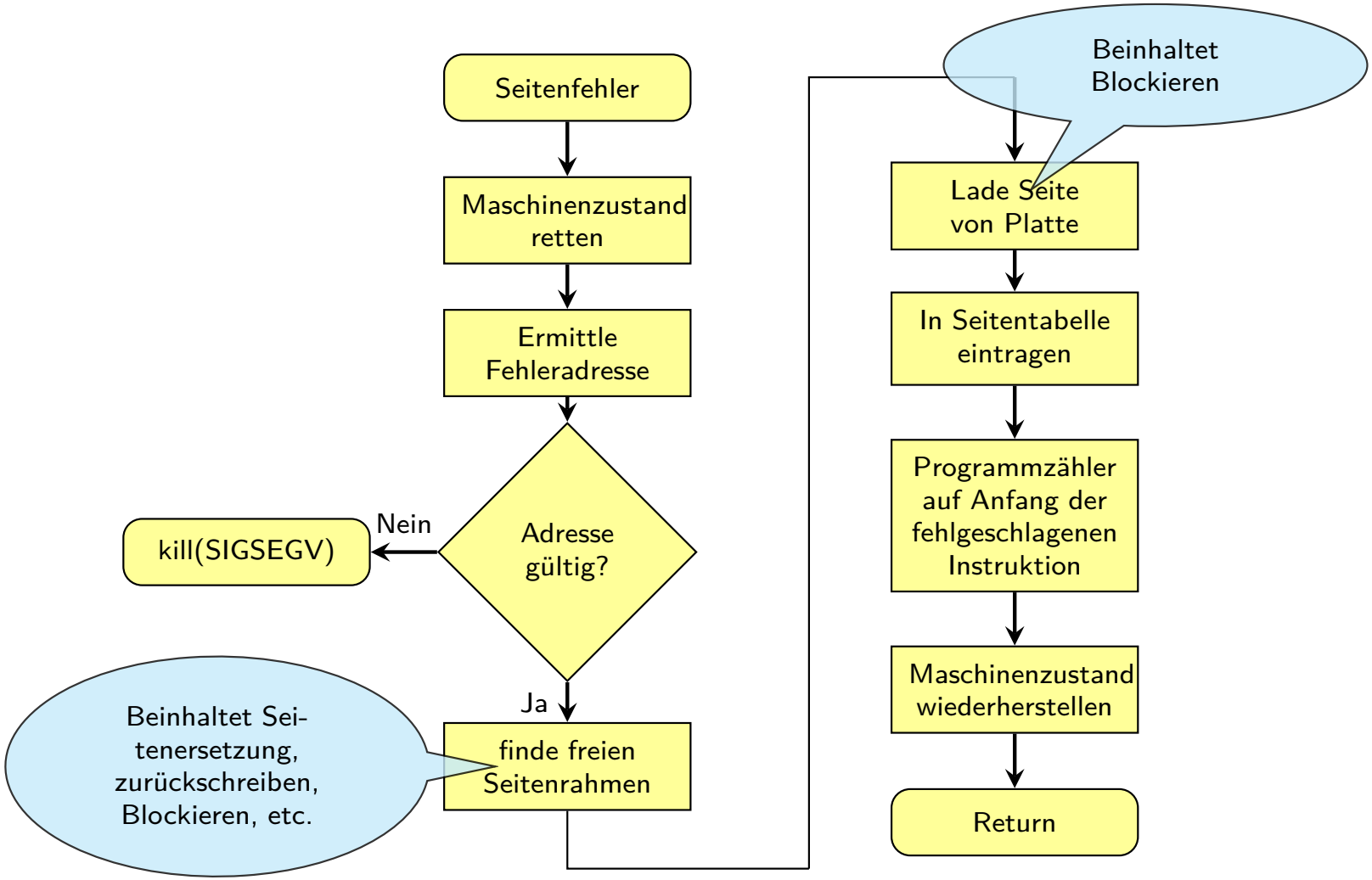
Behandlung eines Seitenfehlers (2)



Behandlung eines Seitenfehlers (2)



Behandlung eines Seitenfehlers (2)



Probleme mit Seitentabellen

Hochschule RheinMain

Die Seitentabelle kann sehr groß werden, z.B.:

- Seitengröße: 4 KB ($= 2^{12}$ Bytes)
- 32-Bit Architektur:
  - D.h. Adressen sind 32 Bit lang.

  $\Rightarrow$ Seitentabelle braucht $\frac{2^{32}}{2^{12}} = 2^{20} = 1048576$ Einträge à 4 Byte $\hat{=}$ 4 MB!
  - **und**: jeder Prozess hat seinen eigenen Adressraum, damit auch seine eigene Seitentabelle!
- 64-Bit Architektur:
  - D.h. Adressen sind 64 Bit lang.

  $\Rightarrow$ Seitentabelle braucht $\frac{2^{64}}{2^{12}} = 2^{52} \approx 4.5 \cdot 10^{15}$ Einträge!

$\rightarrow$ **Probleme:**

1. Hoher Platzbedarf zum Halten der Seitentabelle (s.o.)
2. **Jede** Operation des Prozessors (Befehl holen, Daten lesen / schreiben) erfordert eine Referenzierung der Seitentabelle $\rightarrow$ Langsam!

Lösungen (im Folgenden diskutiert):

- Platzproblem: Mehrstufige Seitentabellen
- Geschwindigkeitsproblem: Translation Lookaside Buffer (TLB)

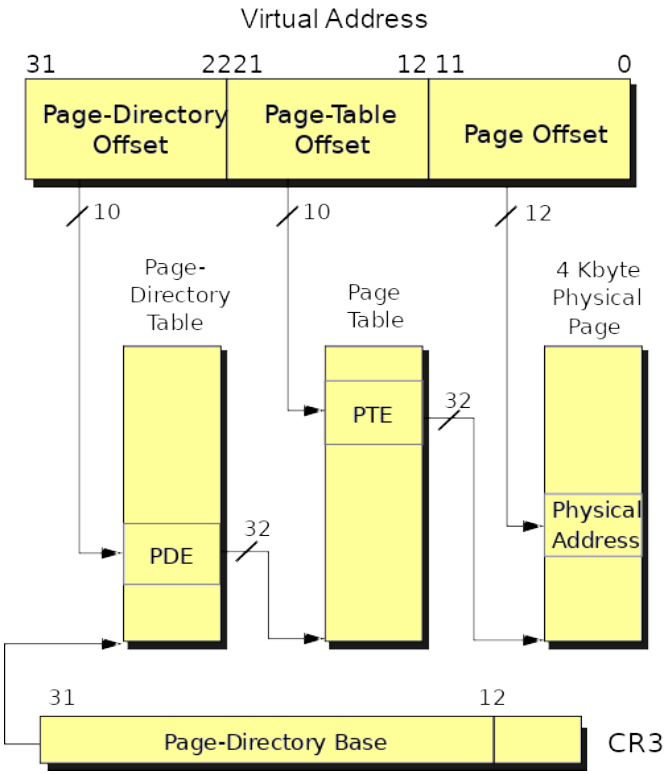
Mehrstufige Seitentabellen

Hochschule RheinMain

Prozesse belegen in der Regel nicht den gesamten Adressraum

- viele Seitentabelleneinträge bleiben also leer (P = 0).
- verschwenden damit selbst Speicherplatz.

Lösung: Mehrstufige Seitentabellen (*wieso* spart das Speicher?)



Quelle: AMD64 Architecture Programmer's Manual, Vol. 2

Seitennummer (hier: 20-Bit) wird aufgeteilt in 2 x 10-Bit:

- **Page Directory Offset**: Index in 1. Tabellenstufe, führt zu 2. Stufe
- **Page Table Offset**: Index in 2. Stufe, führt zu Seitentabelleneintrag

Translation Lookaside Buffer (TLB)

Prozesse zeigen i.d.R. Lokalitätsverhalten

→ Einige (wenige) Seitentabelleneinträge werden immer wieder verwendet.

Lösung: TLB: Cache für Seitentabelleneinträge

- Schneller Assoziativspeicher, in MMU integriert
- Hält kleinere Anzahl der zuletzt genutzten Seitentabelleneinträge vor
- Bei TLB Hit: Seitentabelleneintrag wird schnell geliefert
- Bei TLB Miss: „Page Table Walk“, gefundener Seitentabelleneintrag wird gespeichert
- Dabei Verdrängungsstrategie: z.B. LRU oder Random

→ **Im Regelfall große Beschleunigung, aber zu beachten:**

- Bei Prozessumschaltung: TLB muss ggf. vom Betriebssystem gelöscht werden
- Nach Prozessumschaltung: TLB nachladen kostet zusätzliche Zeit

→ Zu häufige Prozesswechsel bewirken viel Overhead
(Vgl. 4.3.1, processor sharing für Grenzfall $q \longrightarrow 0$)

Seitenersetzungsalgorithmen

Problem:

- Geschickte Auswahl von zu ersetzenden Seiten des Hauptspeichers:

  Wird eine häufig benutzte Seite ausgelagert, ist die Wahrscheinlichkeit groß, dass diese schon bald wieder eingelagert werden muss, was Zusatzaufwand für das Betriebssystem und Blockierungszeiten für den betroffenen Prozess mit sich bringt.

Ziel:

- Gute Performance des Systems.
- „Günstig“ auszulagernde Seiten *schnell* identifizieren.

Seitenersetzungsalgorithmen (*Page Replacement Algorithms*) sind ein klassisches Problem der Betriebssysteme, vielfach theoretisch und experimentell untersucht.

Überblick

In der BS-Literatur betrachtete Seitenersetzungsverfahren:

- Der optimale Seitenersetzungsalgorithmus
- Not-Recently-Used (NRU)
- First-In, First-Out (FIFO) und Second-Chance
- Clock-Algorithmus
- Least-Recently-Used (LRU)
- Aging-Algorithmus

„Optimales“ Verfahren

Hochschule RheinMain

Algorithmus:

- Wird ein Seitenrahmen benötigt, bestimme für alle eingelagerten Seiten, wieviele Instruktionen lang die Seite **zukünftig** durch die noch auszuführenden Instruktionen nicht referenziert werden wird.
- Wähle die Seite zur Ersetzung aus, die am längsten nicht referenziert werden wird.

Bemerkungen:

- Der Algorithmus schiebt den Seitenfehler, der dazu führt, dass die auszulagernde Seite wieder eingelagert werden muss, am weitesten in die Zukunft.
- Die benötigte Information über das **zukünftige Programmverhalten** ist aber i.d.R. nicht vorhanden!
- Der Algorithmus ist daher nicht realisierbar, nur für Vergleichszwecke interessant.

Not-Recently-Used (NRU)

Voraussetzungen:

- Verwendung der im Seitentabelleneintrag gespeicherten Status-Bits je Seite (vgl.9.5.2):
  - **R-Bit** (*referenced*): wird gesetzt, wenn auf die Seite **lesend oder schreibend** zugegriffen wird.
  - **M-Bit** (*modified*): wird gesetzt, wenn auf die Seite **schreibend** zugegriffen wird).
- Status-Bits werden durch die Hardware automatisch gesetzt.
- Gesetzte Status-Bits können durch das Betriebssystem (softwaremäßig) zurückgesetzt werden.

Algorithmus

- Bei Prozessstart werden beide Status-Bits für alle Seiten des Prozesses zurückgesetzt.
- Periodisch, z.B. bei jeder Uhr-Unterbrechung, wird das Referenced-Bit aller Seiten durch das Betriebssystem zurückgesetzt. Dadurch können im folgenden referenzierte Seiten von nicht referenzierten unterschieden werden.
- Uhr-Unterbrechungen löschen das Modified-Bit **nicht**. Es wird weiter für die Entscheidung über das Zurückschreiben benötigt.
- Tritt ein Seitenfehler auf, teile alle eingelagerten Seiten in die folgenden Klassen entsprechend der aktuellen Werte des R- und des M-Bits ein:
  - Klasse 0: nicht referenziert, nicht modifiziert.
  - Klasse 1: nicht referenziert, modifiziert.
  - Klasse 2: referenziert, nicht modifiziert.
  - Klasse 3: referenziert, modifiziert.
- Wähle zufällig aus der kleinst-nummerierten nicht-leeren Klasse eine Seite zur Ersetzung aus.

Bemerkungen

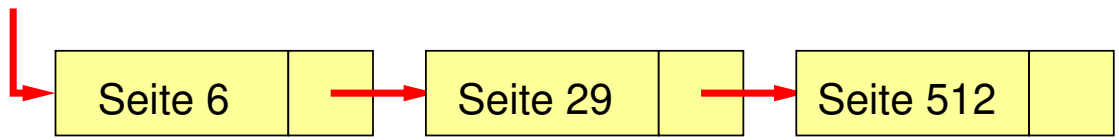
Der Not-Recently-Used-Algorithmus ist ...

- ...Einfach
- ...effizient implementierbar
- ...nicht optimal, aber Performance ist akzeptabel

First-In, First-Out (FIFO)

Hochschule RheinMain

- **Idee:** Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert
- Verwaltung über eine Liste:
  - Bei Einlagerung wird Eintrag am Listenende angehängt
  - Bei Seitenfehler: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.

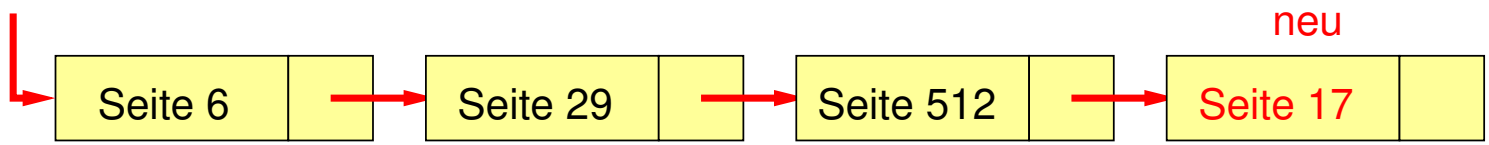


- Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

First-In, First-Out (FIFO)

Hochschule RheinMain

- **Idee:** Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert
- Verwaltung über eine Liste:
  - Bei Einlagerung wird Eintrag am Listenende angehängt
  - Bei Seitenfehler: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.

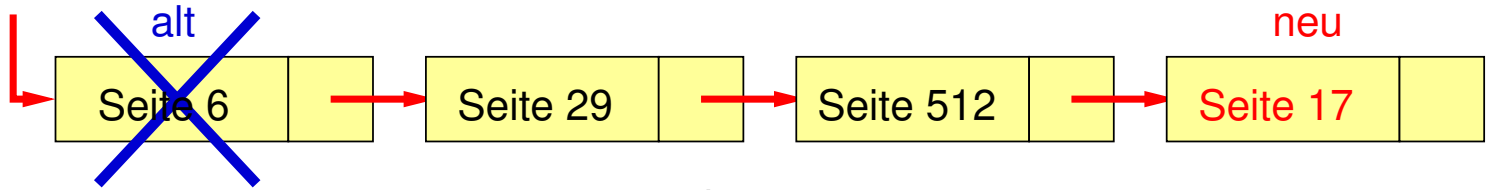


- Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

First-In, First-Out (FIFO)

Hochschule RheinMain

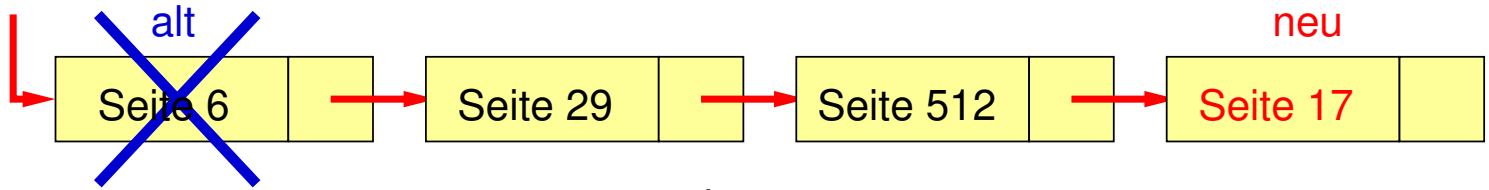
- **Idee:** Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert
- Verwaltung über eine Liste:
  - Bei Einlagerung wird Eintrag am Listenende angehängt
  - Bei Seitenfehler: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.



- Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

First-In, First-Out (FIFO)

- **Idee:** Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert
- Verwaltung über eine Liste:
  - Bei Einlagerung wird Eintrag am Listenende angehängt
  - Bei Seitenfehler: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.

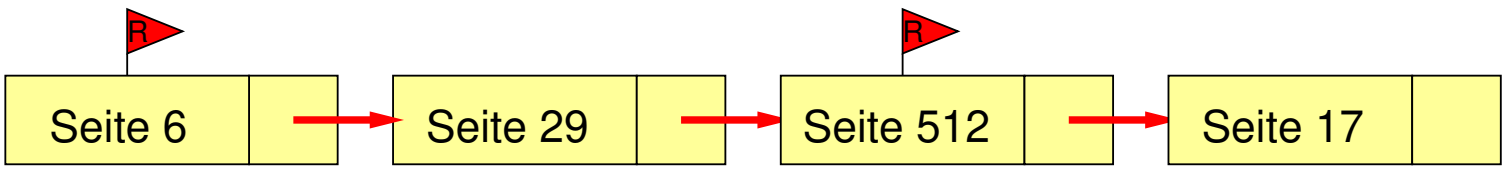


- Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

Second Chance

Hochschule RheinMain

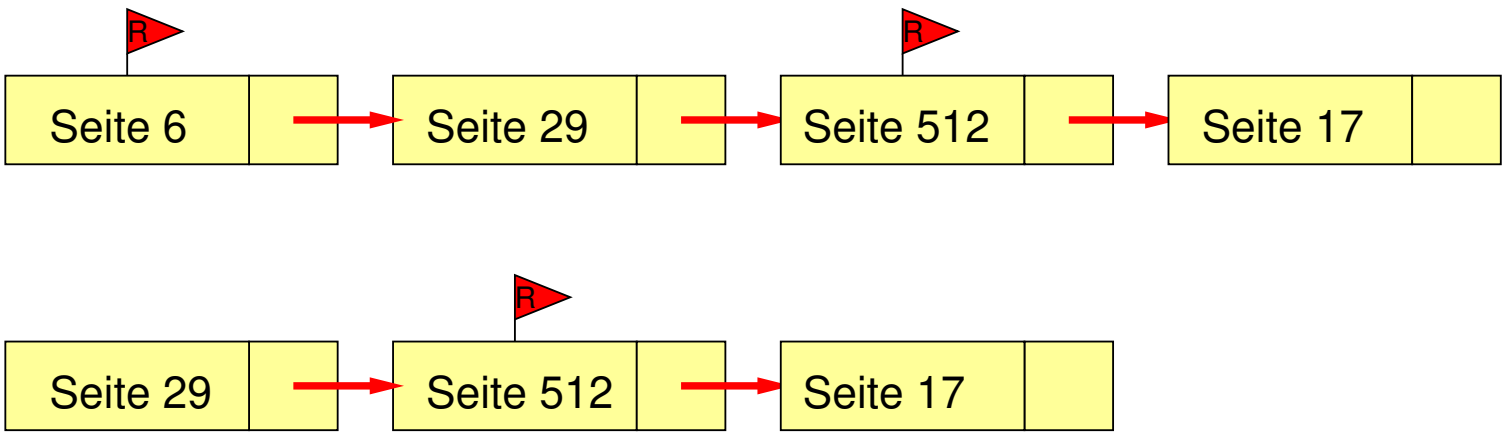
- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
  - Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
  - Wenn R-Bit nicht gesetzt: Seite auslagern, fertig
- Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)



Second Chance

Hochschule RheinMain

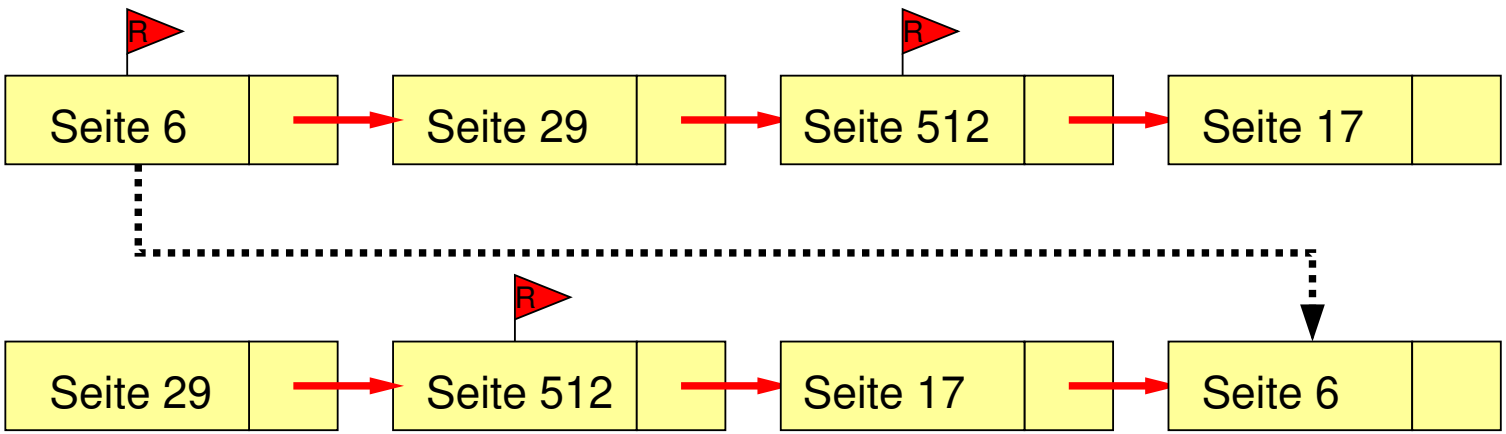
- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
  - Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
  - Wenn R-Bit nicht gesetzt: Seite auslagern, fertig
- Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)



Second Chance

Hochschule RheinMain

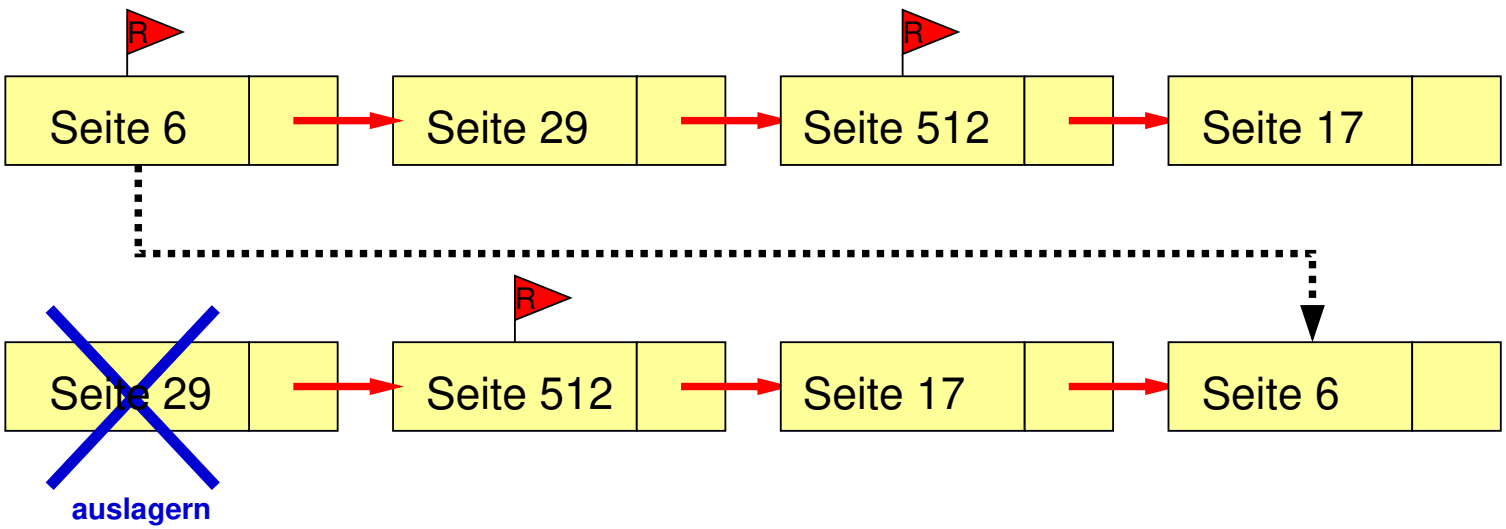
- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
  - Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
  - Wenn R-Bit nicht gesetzt: Seite auslagern, fertig
- Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)



Second Chance

Hochschule RheinMain

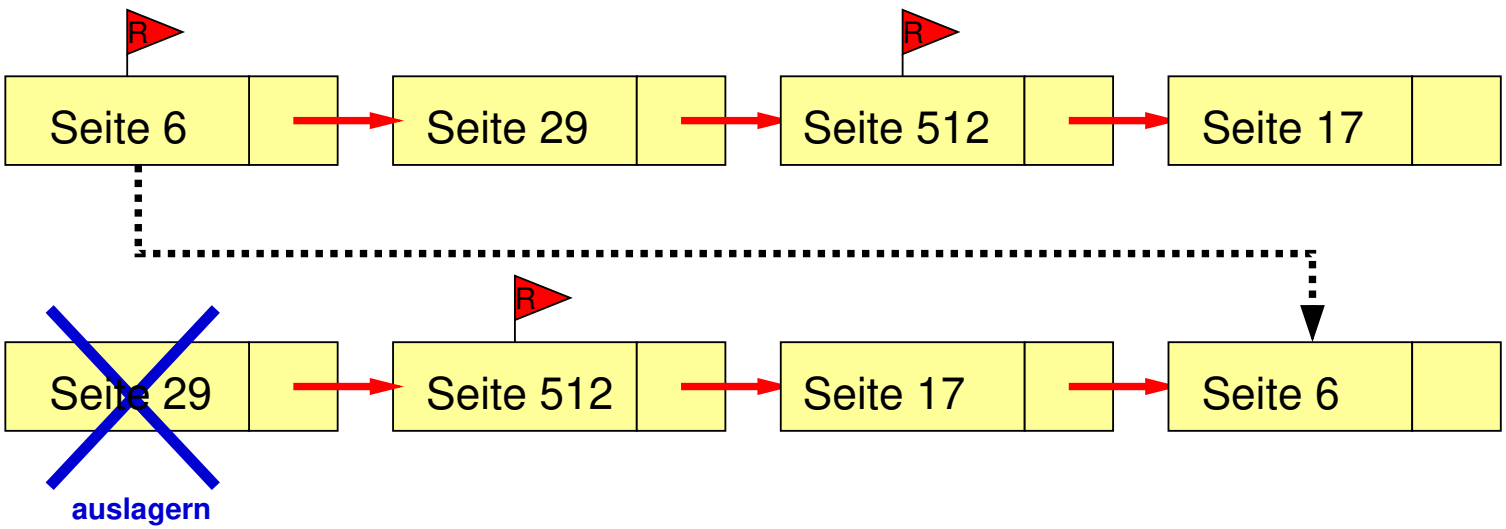
- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
  - Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
  - Wenn R-Bit nicht gesetzt: Seite auslagern, fertig
- Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)



Second Chance

Hochschule RheinMain

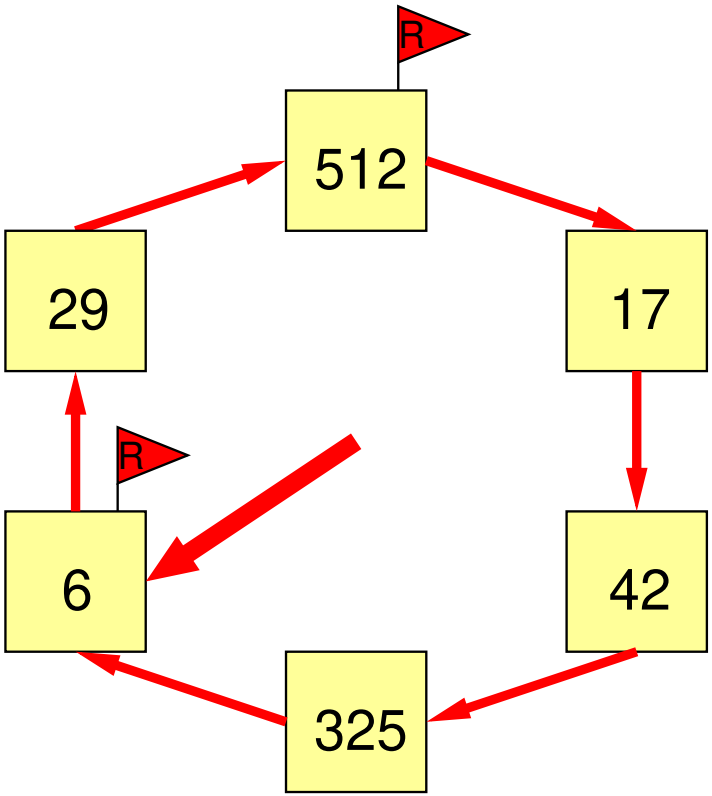
- **Idee:** Ähnlich FIFO, aber mit Beachtung des Referenced-Bits
- Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:
  - Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
  - Wenn R-Bit nicht gesetzt: Seite auslagern, fertig
- Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)



Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

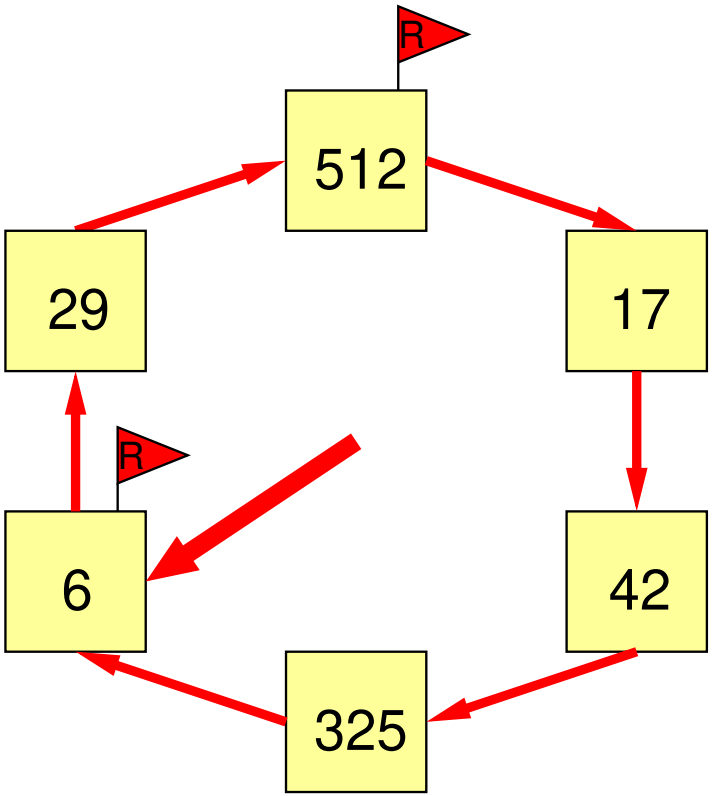
- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

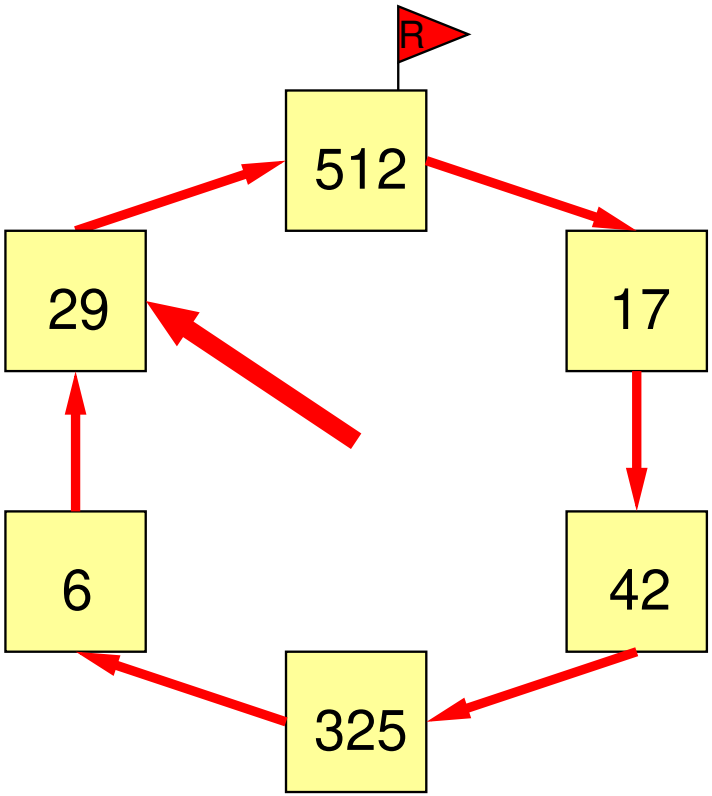
- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

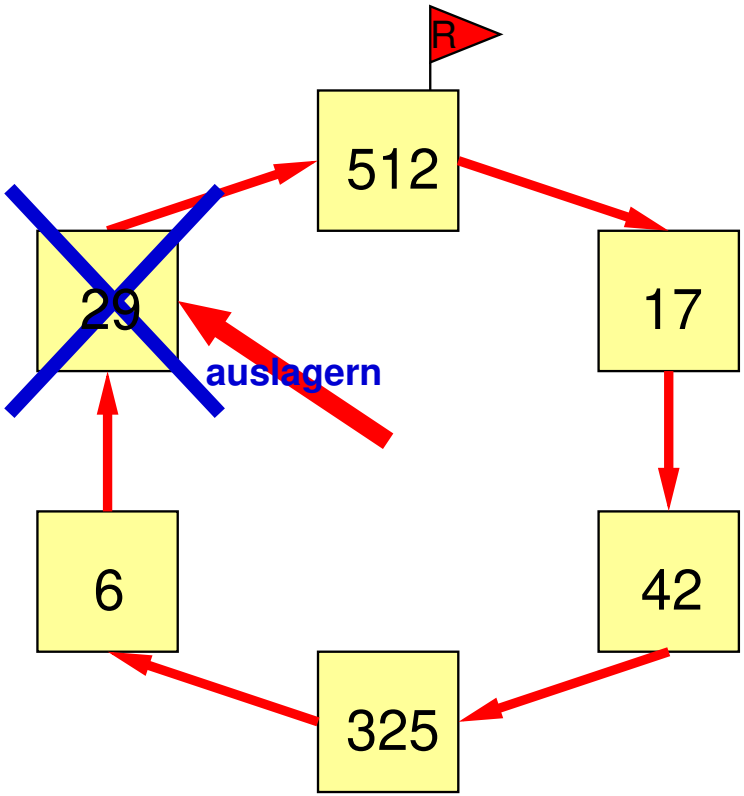
- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

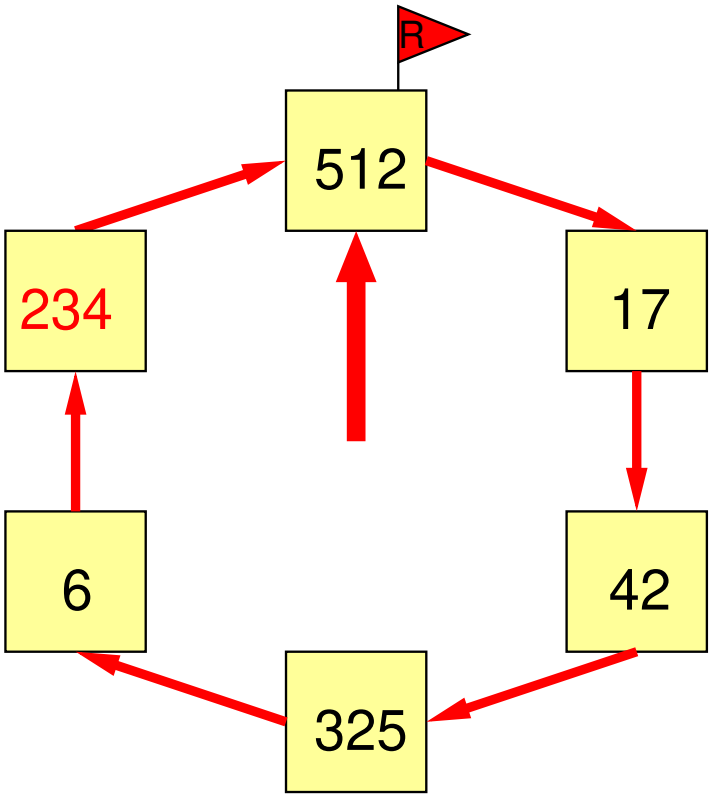
- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

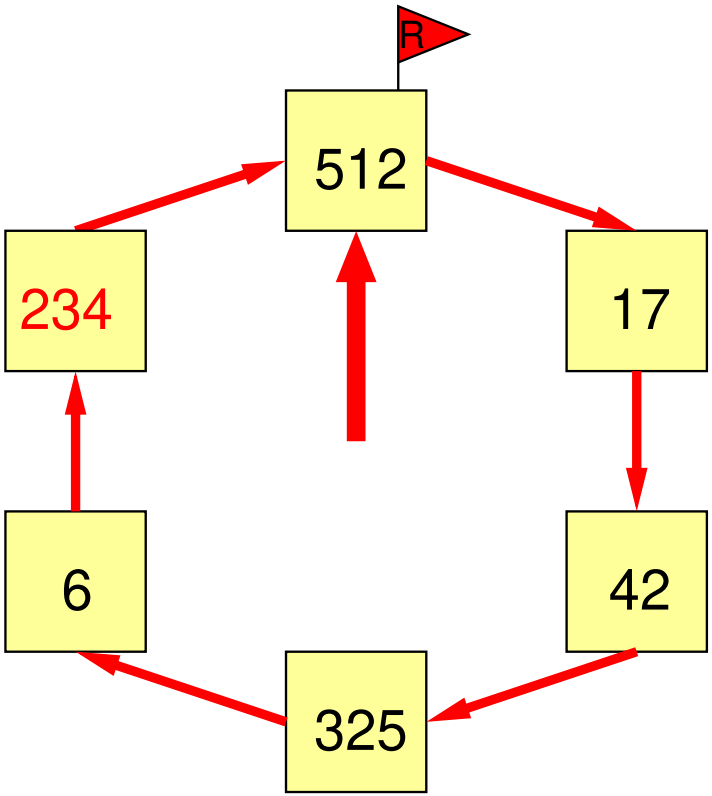
- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Clock-Algorithmus

Hochschule RheinMain

- Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet
- Der Zeiger der Uhr zeigt auf die älteste Seite.



Bei Seitenfehler: älteste Seite prüfen

- Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen
- Wenn R-Bit nicht gesetzt:
  - Seite auslagern
  - einzulagernde Seite einfügen
  - Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Least-Recently-Used (LRU)

Voraussetzungen:

- Gute Annäherung an den optimalen Algorithmus ist möglich, wenn man aus dem Zugriffsverhalten in der jüngeren Vergangenheit auf das Verhalten in der Zukunft schließen kann.
- Beobachtung: Seiten, die schon lange nicht mehr benötigt wurden, werden auch mit hoher Wahrscheinlichkeit für längere Zeit nicht mehr benötigt werden.

Algorithmus:

- Wenn ein Seitenfehler auftritt, ersetze die Seite, die am längsten unbenutzt ist (*least-recently-used*, LRU).

Least-Recently-Used (LRU) (2)

Bemerkungen:

- LRU ist realisierbar, Implementierung ist aber aufwändig.
- Implementierungsidee:
  - Die Menge aller Seiten im Speicher wird mit einer verketteten Liste verwaltet.
  - Die gerade benutzte Seite steht am Kopf der Liste, die am längsten nicht benutzte Seite am Ende.
  - Bei jedem Speicherzugriff wird die zugegriffene Seite in der Liste gesucht und an den Anfang gestellt (Umordnung!).
  - Wähle die Seite am Ende der Liste als zu ersetzende Seite.
- Eine effiziente Implementierung ist nur mit spezieller Hardwareunterstützung möglich und nur in Ausnahmefällen sinnvoll (z.B. für die Verwaltung von TLBs mit wenigen Einträgen).

Aging-Algorithmus (Alterung)

Ziel:

- Approximation von LRU in Software.

Algorithmus:

- Jeder Seite wird Software-Zähler der Länge L Bit zugeordnet (z.B. 8-Bit lang), initialisiert mit 0.
- Bei jeder Uhr-Unterbrechung für jede eingelagerte Seite:
  - Den Zähler der Seite wird um 1 Bit nach rechts schieben (Aging = Altern).
  - Das R-Bit der Seite ersetzt die führende Stelle des Zählers.
  - Das R-Bit wird gelöscht. (Es wird durch die Hardware im Falle eines weiteren Zugriffs automatisch neu gesetzt).
- Wenn ein Seitenfehler auftritt, wird die Seite mit dem kleinsten Zählerwert ersetzt.

Aging: Beispiel

Hochschule RheinMain

R-Bits und Zähler der Seiten $0 \ldots 5$ nach $t = 1 \ldots 6$ Uhr-Unterbrechungen

t = 1

Seite	R	Zähler
0	1	10000000
1	0	00000000
2	1	10000000
3	0	00000000
4	1	10000000
5	1	10000000

t = 2

Seite	R	Zähler
0	1	11000000
1	1	10000000
2	0	01000000
3	0	00000000
4	1	11000000
5	0	01000000

t = 3

Seite	R	Zähler
0	1	11100000
1	1	11000000
2	0	00100000
3	1	10000000
4	0	01100000
5	1	10100000

t = 4

Seite	R	Zähler
0	1	11110000
1	0	01100000
2	0	00010000
3	0	01000000
4	1	10110000
5	0	01010000

t = 5

Seite	R	Zähler
0	0	01111000
1	1	10110000
2	1	10001000
3	0	00100000
4	0	01011000
5	0	00101000

t = 6

Seite	R	Zähler
0	1	10111100
1	0	01011000
2	1	11000100
3	0	00010000
4	0	00101100
5	0	00010100

Aging: Beispiel

Hochschule RheinMain

R-Bits und Zähler der Seiten $0 \ldots 5$ nach $t = 1 \ldots 6$ Uhr-Unterbrechungen

t = 1

Seite	R	Zähler
0	1	10000000
1	0	00000000
2	1	10000000
3	0	00000000
4	1	10000000
5	1	10000000

t = 2

Seite	R	Zähler
0	1	11000000
1	1	10000000
2	0	01000000
3	0	00000000
4	1	11000000
5	0	01000000

t = 3

Seite	R	Zähler
0	1	11100000
1	1	11000000
2	0	00100000
3	1	10000000
4	0	01100000
5	1	10100000

t = 4

Seite	R	Zähler
0	1	11110000
1	0	01100000
2	0	00010000
3	0	01000000
4	1	10110000
5	0	01010000

t = 5

Seite	R	Zähler
0	0	01111000
1	1	10110000
2	1	10001000
3	0	00100000
4	0	01011000
5	0	00101000

Kleinster Zähler:

→ Seite 3 auslagern!

t = 6

Seite	R	Zähler
0	1	10111100
1	0	01011000
2	1	11000100
3	0	00010000
4	0	00101100
5	0	00010100

Aging: Bemerkungen

Bemerkungen zum Verhältnis zu LRU:

- Information über das Zugriffsverhalten auf eine Seite während eines Uhr-Intervalls wird auf ein einziges Bit vergröbert.
- Information, die älter als L Zeitintervalle ist, wird als wertlos eingestuft (geht durch Rechts-Shifts verloren).
- Eine Seite, die während n Uhr-Unterbrechungen nicht referenziert wurde, besitzt n führende Nullen im Zähler.
- Die Auswertung des Zählers muss nur nach Ablauf dieses Zeitintervalls erfolgen, nicht bei jeder Speicherreferenz.
- Die Aufzeichnung von nur einem Bit je Zeitintervall erlaubt damit nicht die Unterscheidung einer früheren oder einer späteren Referenz innerhalb des Intervalls.
- Insgesamt ist die Approximation von LRU hinreichend gut.

Entwurfsprobleme bei Paging-Systemen

Ziel:

- Kennenlernen von Aspekten, die beim Entwurf von Paging-Systemen besondere Beachtung finden, da sie starken Einfluss auf die Performance haben.

Überblick:

1. Working Set Modell
2. Lokale oder globale Seitenersetzung
3. Seitengröße
4. Implementierungsprobleme

Das Working Set Modell (1)

- Prozesse zeigen i.d.R. ein **Lokalitätsverhalten**.

Definition: Working Set

Die Menge der Seiten, die ein Prozess augenblicklich nutzt, wird sein **Working Set** (**Arbeitsbereich**) genannt.
(P. Denning, 1968).

- Ist der zugeordnete Hauptspeicher zu klein, um das Working Set aufzunehmen, so entstehen sehr viele Page Faults.
- Wenn nach wenigen Zugriffen die Seite wieder benötigt wird, die gerade verdrängt wurde, so sinkt die Ausführungsgeschwindigkeit auf einige Instruktionen je Seiteneinlagerung (typisch mehrere Millisekunden). Man bezeichnet eine solche Situation als **Thrashing**.

Das Working Set Modell (2)

- Einlagern von Seiten, obwohl noch keine entsprechenden Seitenfehler vorliegen, wird als **Prepaging** bezeichnet (im Gegensatz zum üblichen **Demand Paging**).
- Ist das Working Set eines Prozesses bekannt, kann mittels Prepaging nach einem Prozesswechsel verhindert werden, dass der Prozess durch eine große Anzahl von Seitenfehlern seine Umgebung zunächst wieder aufbauen muss, in dem das Working Set vollständig eingelagert wird. (Working Set Model, Denning, 1970)
- Durch das Working Set Modell kann die Seitenfehlerrate drastisch gesenkt werden.

Das Working Set Modell (3)

Ansatz zur Bestimmung des Working Sets:

- Verwendung des Aging-Algorithmus (siehe 9.6.6).
- Eine 1 in den höherwertigen n Bits des Alterungszählers kennzeichnet die zugehörige Seite als zum Working Set gehörig.
- Wird eine Seite für n Uhrticks nicht referenziert, so scheidet sie aus dem Working Set aus.

Verbesserung des Clock-Algorithmus (siehe 9.6.4) ist möglich:

- Seiten, die zum Working Set gehören, werden bei gelöschtem R-Bit übersprungen.
- Dieser Algorithmus wird **wsclock** (*Working Set Clock*) genannt.

Mindestgröße

Wie viele Seiten kann eine einzige Maschineninstruktion referenzieren?

1. Befehlscode lesen
2. Quelloperand holen
3. Zieloperand speichern

- Operand/Maschineninstruktion i.d.R. > 1 Byte

$\rightarrow$ Jeder kann (worst-case) über max. 2 Seiten verteilt sein.

$\Rightarrow$**Mindestens 6 Seitenrahmen[7] für einen Prozess, sonst u.U. „Thrashing innerhalb einer Instruktion“!**

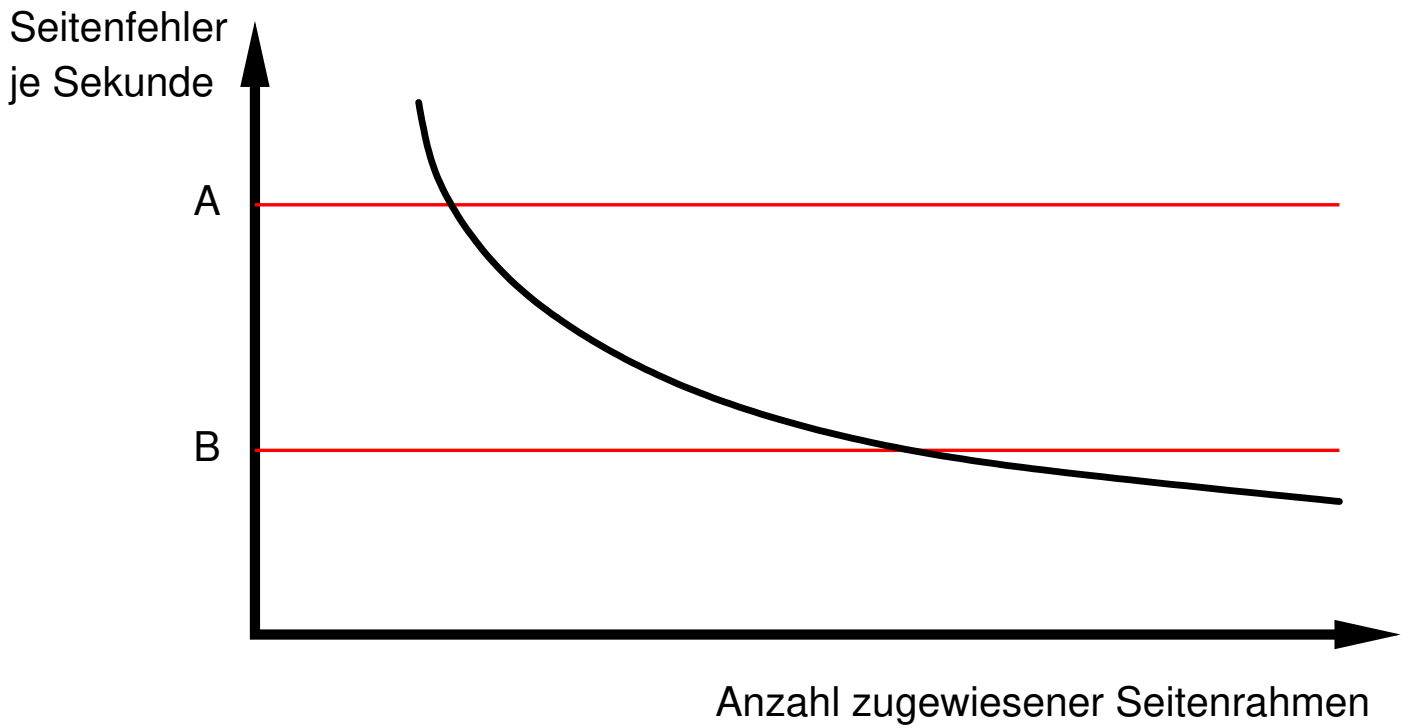
[7] Architekturabhängig: z.B. MIPS: nur 2 Seitenrahmen

Lokale oder globale Seitenersetzung

- Im Mehrprogrammbetrieb haben mehrere Prozesse Seiten im Speicher.
- **Lokale Seitenersetzung**: bei der Auswahl einer zu ersetzenden Seite werden nur die Seiten des Prozesses selbst betrachtet.
- **Globale Seitenersetzung**: alle im Speicher befindlichen Seiten, also auch die anderer Prozesse, werden in Betracht gezogen.
- Lokale Seitenersetzungs-Algorithmen ordnen jedem Prozess eine feste Anzahl von Seitenrahmen zu, globale Algorithmen alloziieren Seitenrahmen dynamisch.
- **Globale Algorithmen** arbeiten i.A. mit **besserer Performance**.
- Im Falle eines globalen Algorithmus muss das Betriebssystem dynamisch entscheiden, wie viele Seitenrahmen jeder Prozess zugewiesen bekommt. Einfacher Ansatz: Betrachtung des Working Set, verhindert aber kein Thrashing.

Lokale oder globale Seitenersetzung (2)

- Bester Ansatz: Kontrolle der Seitenfehlerfrequenz (Page Fault Frequency PFF) über die Anzahl der zugeordneten Seitenrahmen.
- Typischer Verlauf:



- Seitenfehlerrate zwischen A und B wird als akzeptabel angesehen. Steigt die Seitenfehlerrate über A, werden zusätzliche Seitenrahmen zugeordnet, sinkt sie unter B, werden Seitenrahmen entzogen.

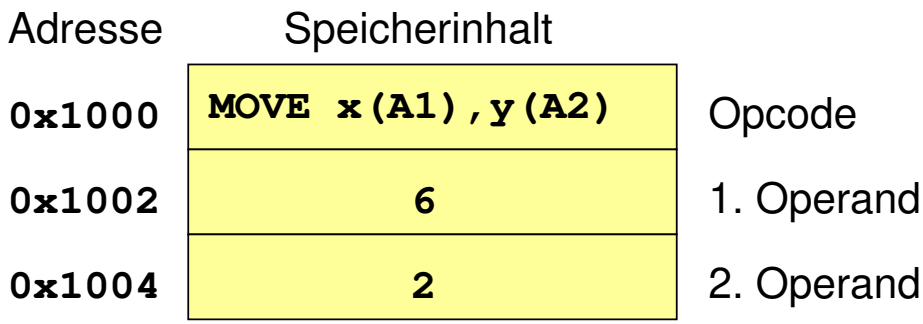
Seitengröße

- Bestimmung der optimalen Seitengröße erfordert ein Ausbalancieren von verschiedenen widersprechenden Faktoren:
  - Geringe interne Fragmentierung spricht für eine kleine Seitengröße.
  - Kleine Seiten erfordern eine große Seitentabelle.
  - Kosten für die Ein-/Auslagerung sind stark von Positionierzeiten und Rotationsverzögerung der Platte beeinflusst. Spricht für große Seitengröße.
  - Hardware-Aufwand für Memory Management Unit.
- Heutige Seitengrößen: 512 B - 8 KB, typisch 4 KB

Implementierungsprobleme (1)

Instruction Retry nach Seitenfehler:

- Eine Instruktion besteht aus mehreren Worten.
- Beispiel (MC68000): `MOVE 6(A1), 2(A2)`



- Bei Seitenfehler kann PC=0x1000, 0x1002 oder 0x1004 sein
- „Wieviel von der Instruktion ist schon abgearbeitet?“
- „Wo/wie muss nach Seitenfehler fortgefahren werden?“
- Kann kompliziert werden ...

Implementierungsprobleme (2)

Sperren von Seiten im Speicher

- z.B. für Pufferbereiche während gestarteter I/O-Aufträge.
- Entsprechende Seiten von blockierten Prozessen dürfen dann nicht ausgelagert werden (reserviert für DMA-Transfer).
- Typische Operationen: pin / unpin auf Seiten.

Memory Sharing

- Gemeinsam benutzte Seiten verschiedener Prozesse (insbesondere Code-Seiten) müssen berücksichtigt werden.
- Auslagern würde Seitenfehler für Partner verursachen.
- I.d.R. werden spezielle Datenstrukturen mit Referenzzählern verwendet.

Implementierungsprobleme (3)

Paging-Dämonen

Definition: Deamon

Ein **Dämon** (*Daemon*) ist ein Hintergrundprozess, der nicht mit einem Benutzer in Interaktion tritt (hat kein Terminal).

- Ziel: Vorhalten einer Anzahl freier Seitenrahmen, um bei Auftreten eines Seitenfehlers keine Verzögerung für das Auslagern einer Seite in Kauf nehmen zu müssen (Performance-Aspekt).
- Der Page Daemon wird z.B. periodisch aktiviert, um die Speichersituation zu untersuchen:
  - Liegen zu wenige freie Rahmen vor, beginnt er, Seiten mithilfe des Seitenersetzungsalgorithmus zur Auslagerung auszuwählen.
  - Wurden sie modifiziert, veranlasst er das Zurückschreiben auf den Hintergrundspeicher.
- Der ursprüngliche Inhalt als „frei“ geführter Seitenrahmen bleibt bekannt.
  - Erfolgt ein erneuter Zugriff auf die ursprüngliche Seite, bevor sie überschrieben wird, wird sie aus dem „Frei“-Pool zurückgewonnen.
  - Dieser Vorgang wird **Page Reclaiming** genannt.

Segmentierung

Ziel

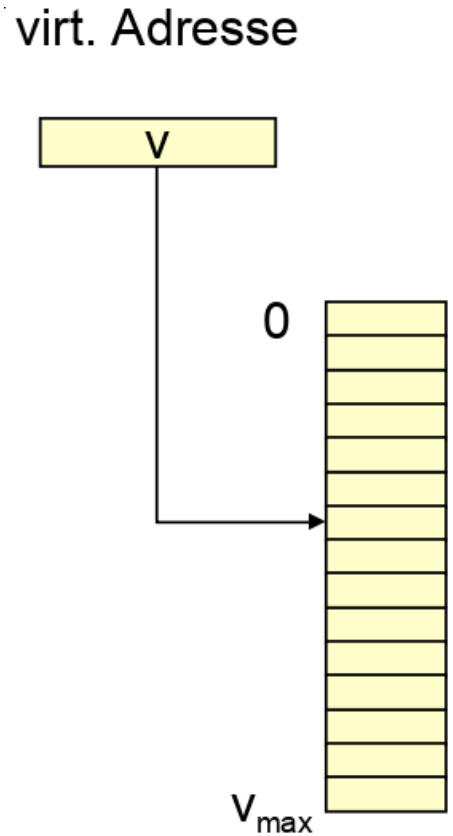
- Bisher wurden Paging-Systeme betrachtet, die einen sogenannten eindimensionalen virtuellen Speicher offerieren.
- Im folgenden wird sogenannter segmentierter virtueller Speicher besprochen.

Begriffsbildung

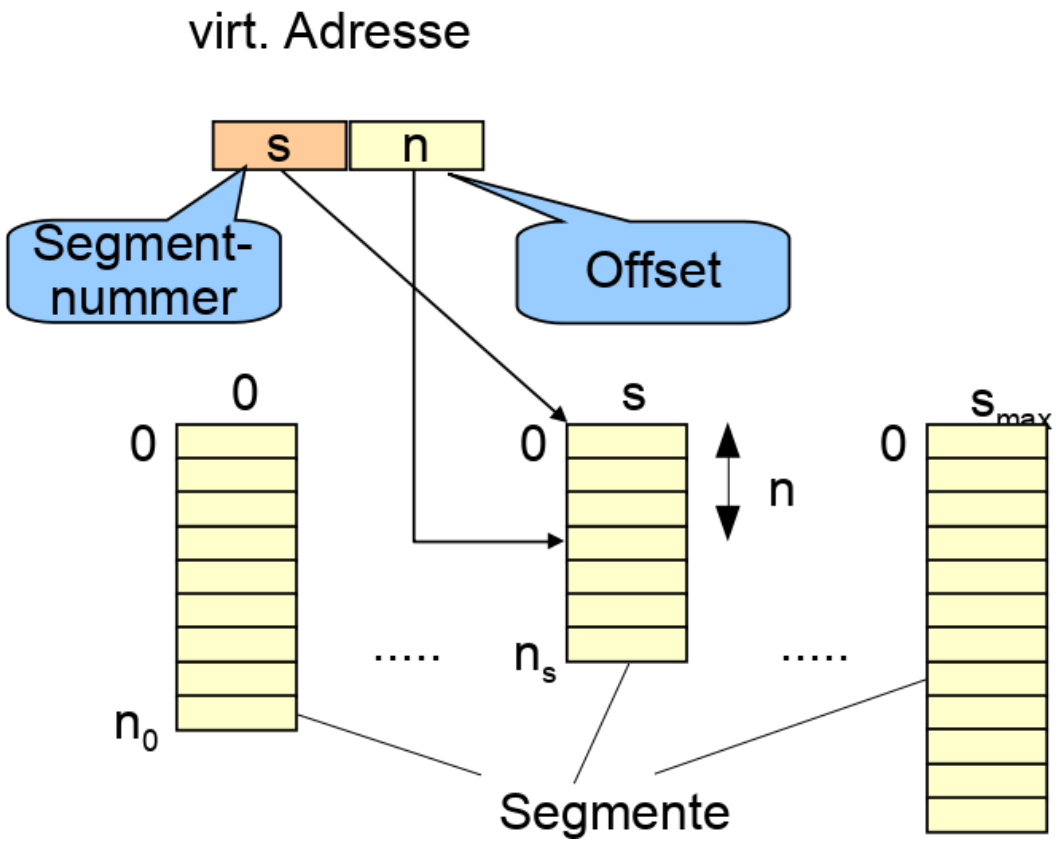
Hochschule RheinMain

Eindimensionale / Zweidimensionale virtuelle Adressräume:

eindimensional:



zweidimensional:



Begriffsbildung (2)

Definition: zweidimensionale Adresse

Eine zweidimensionale (zweiteilige) virtuelle Adresse besteht aus:

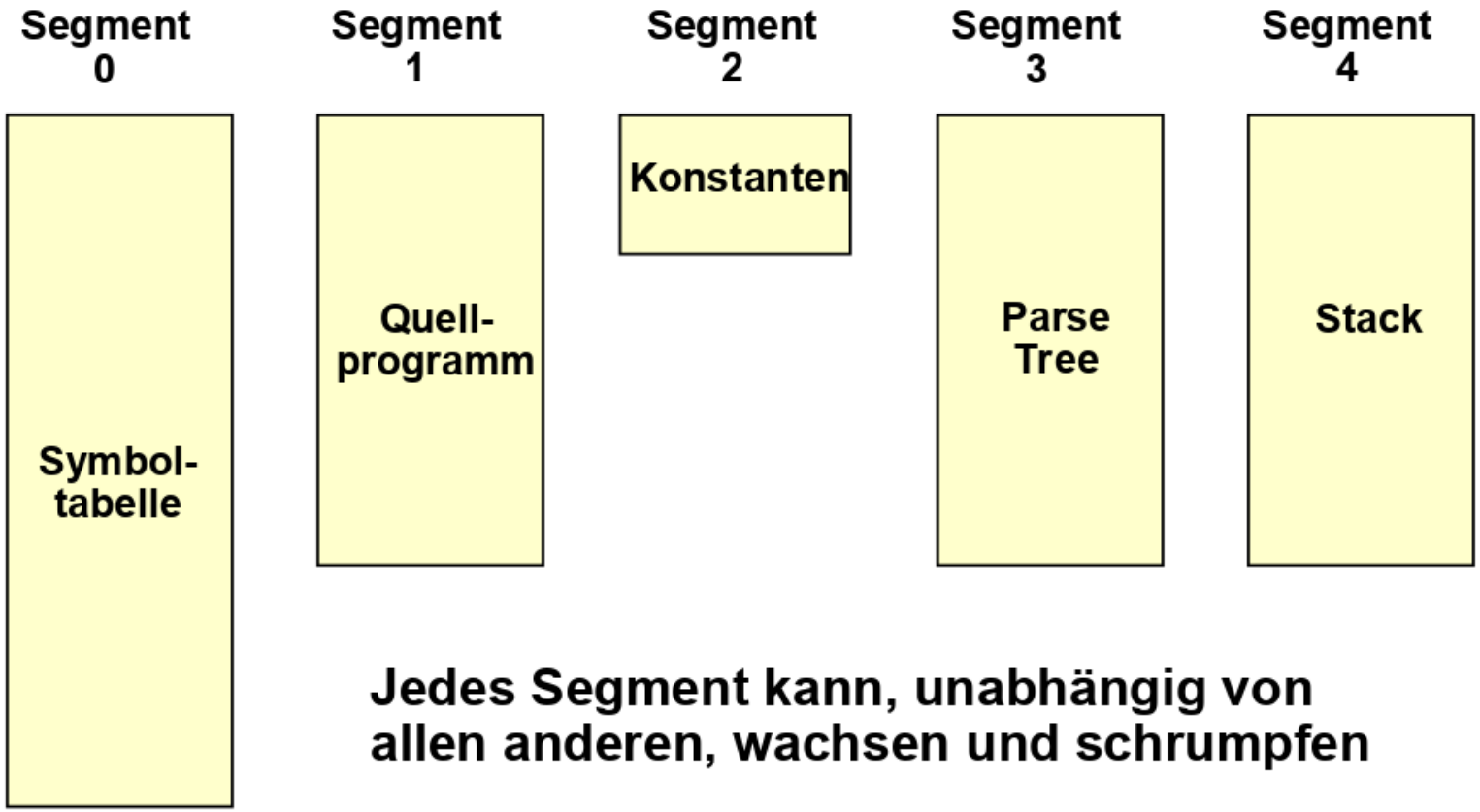
- **Segmentnummer**, die ein Segment selektiert,
- **Adresse im selektierten Segment** (Offset).

- Die Menge aller Segmente definiert den sog. Segmentraum.
- Ein Segment besitzt einen maximalen linearen Adressbereich $\{0, \ldots, L_{max}\}$. Jedes Segment hat damit die Eigenschaften eines (linearen) Adressraums.
- Verschiedene Segmente können unterschiedlich lang sein. Die Länge jedes Segments liegt zwischen 0 und dem erlaubten Maximum L_{max}.
- Die Segmentlänge darf sich i.d.R. während der Lebenszeit des Segments ändern.
- Ein Segment ist eine logische Speichereinheit und als solche dem Programmierer bewusst (sollte es zumindest sein).

Beispiel zur Nutzung

Hochschule RheinMain

Mögliche Segmente für einen Compiler-Lauf:

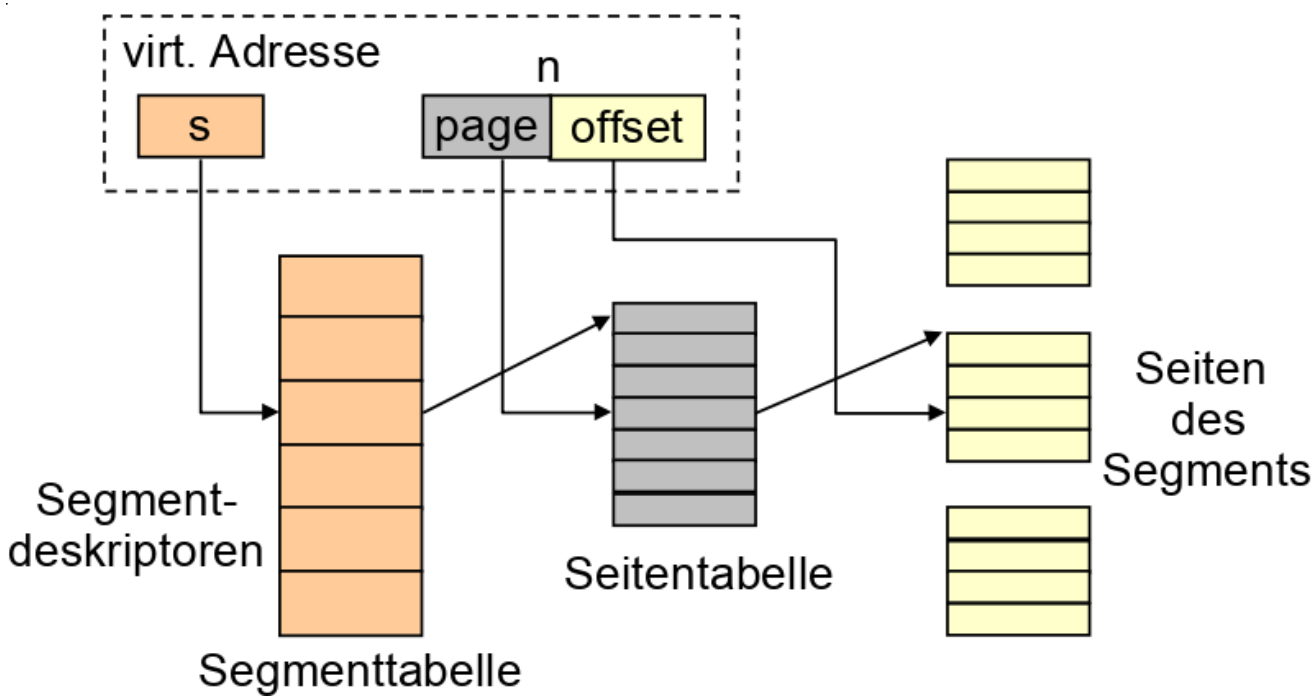


Jedes Segment kann, unabhängig von allen anderen, wachsen und schrumpfen

Segmentierung mit Paging

Hochschule RheinMain

- Ein eingelagertes Segment *muss nicht mehr* vollständig im Hauptspeicher sein.
- Jedes Segment besteht aus einer Folge von Seiten.
- Prinzipielle Adressierung:



Beispiel: Speicherverwaltung in UNIX

Hochschule RheinMain

Ziel

- Kennenlernen der UNIX-Speicherverwaltung.

Überblick:

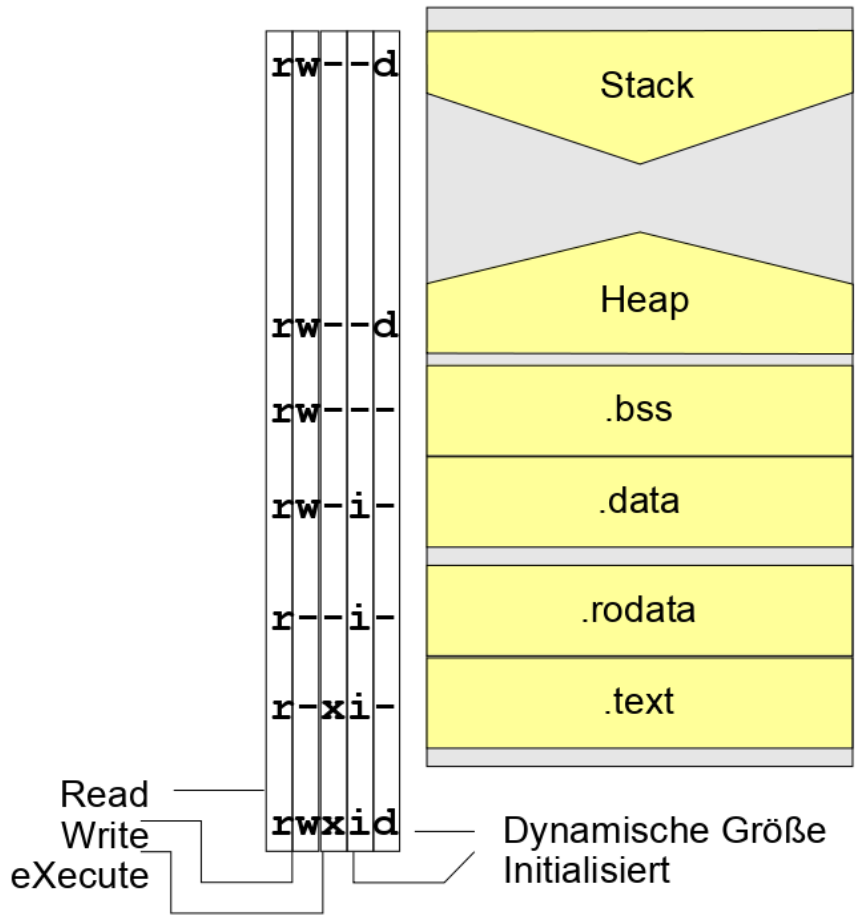
1. Speichermodell.
2. Systemaufrufe zur Speicherverwaltung.

Speichermodell

Hochschule RheinMain

(vgl. 9.1)

- Jeder Prozess besitzt einen eigenen virtuellen Adressraum.
- Der Adressraum enthält aus Benutzersicht (im Anwender-adressbereich) 3 Segmente
  1. Text-Segment
     - ★ Programmcode (.text)
     - ★ Nur-Lese Daten (.rodata)
  2. Daten-Segment
     - ★ Initialisierte Daten (.data)
     - ★ Nicht-initialisierte Daten (.bss)
     - ★ Heap (für malloc() & Co.)
  3. Stack-Segment
     - ★ für lokale Variablen etc.

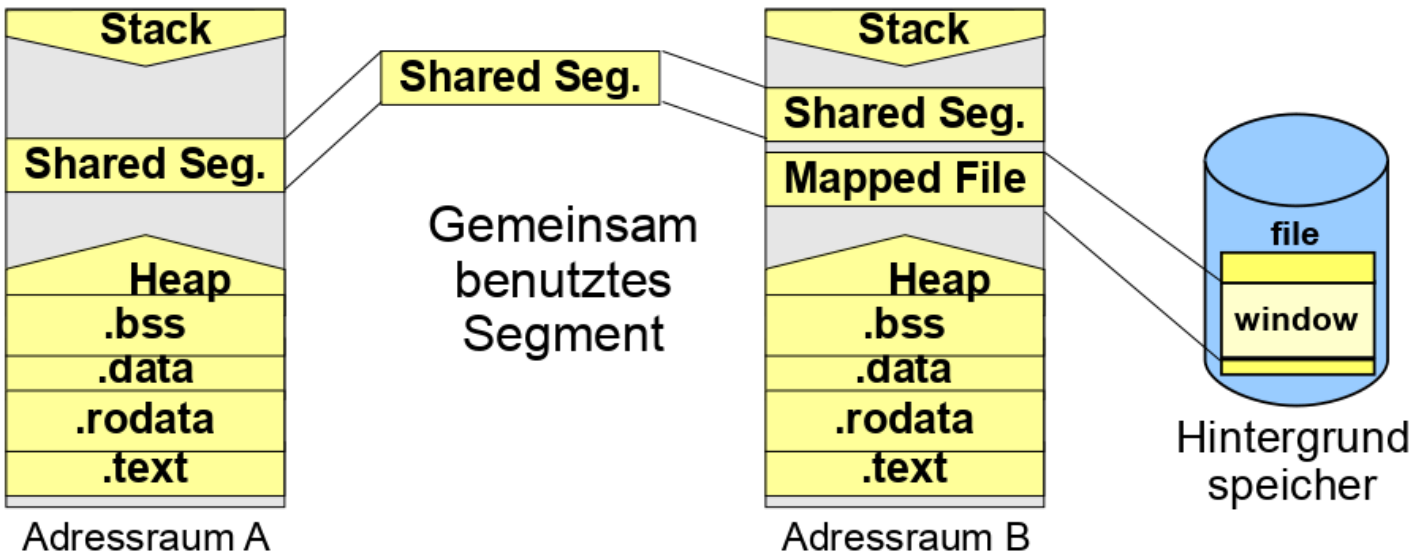


Speichermodell (2)

Hochschule RheinMain

Zusätzlich kann ein Adressraum in neueren UNIX-Systemen ein oder mehrere Segmente der folgenden beiden Typen enthalten:

- Gemeinsam benutztes Speichersegment, das gleichzeitig in mehrere Adressräume eingeblendet ist (z.B. als Shared Memory Data Segment oder Shared Library),
- Mapped File-Segment zur Aufnahme eines Fensters eines Memory Mapped Files (abgebildete Datei, siehe Kapitel Dateisysteme). Auf ein Mapped File kann ohne explizite E/A-Operationen zugegriffen werden.



Adressraumregionen

- Virtueller Adressraum wird in **Regionen** unterteilt.
- Eine Region ist ein zusammenhängender Bereich virtueller Adressen und kann ein identifizierbares Segment (Objekt) aufnehmen (Text-, Daten-, Stack-Segment und andere, s.o.).
- Einem zugeordneten Segment entspricht eine Folge von Seiten des Adressraums.
- Für jede Region können individuelle Schutzattribute festgelegt werden (read-only, read-write, read-execute). Diese übertragen sich auf die repräsentierenden Seiten.
- Gemeinsam genutztes Segment:
  - entspricht Regionen in verschiedenen Adressräumen, denen das Segment zugeordnet ist.
  - Anfangsadresse der Regionen kann in Adressräumen unterschiedlich sein.
    (**Achtung:** Adressen im gemeinsam benutzten Segment sind dann i.d.R. bedeutungslos).

Systemdienste zur Speicherverwaltung

Überblick:

- Im POSIX-Standard existieren keine expliziten Festlegungen.
- Für die Anwendungsprogrammierung wird auf ANSI C-Standardbibliotheksfunktionen zur dyn. Speicherverwaltung zurückgegriffen.
- Im folgenden werden betrachtet:
  - klassische Dienste zur Behandlung der Standard-Segmente.
  - Dienste für Shared Memory Segmente.
  - Dienste für Memory-Mapped Files.
  - Dienste zur Adressraum-Manipulation.

  Dienste für Memory-Mapped Files sowie zur Adressraum-Manipulation sind noch nicht auf allen heutigen UNIX-Varianten verfügbar.

Dienste für Standard-Segmente

- Zur absoluten bzw. relativen Veränderung der Größe des Datensegments (Wachsen oder Schrumpfen) existieren die Systemaufrufe `brk()` bzw. `sbrk()` (letzterer ist häufig eine Bibliotheksfunktion):
  `caddr_t brk(caddr_t addr); caddr_t sbrk(int incr);`
- Auf Programmebene werden i.d.R. die ANSI C-Bibliotheksfunktionen `malloc()` und `free()` benutzt. Implementierung von `malloc()` greift z.B. auf `brk()` zurück, wenn die Heap-Größe erweitert werden muss.
- Die Größe des **Stack-Segments** wird nicht auf Programmebene verändert, sondern vom Betriebssystem implizit vergrößert, wenn sich der Stack zur Ausführungszeit als zu klein erweist.
- Die Größe des **Text-Segments** ist statisch und damit nicht veränderbar.

Dienste für Shared Memory Segmente (1)

- Shared Memory Segmente sind Bestandteil der System V IPC-Mechanismen und stellen gemeinsam von mehreren Prozessen benutzbare Segmente mit festlegbaren Zugriffsattributen dar.
- Die Benutzung von Shared Memory Segmenten setzt i.d.R. die korrekte Synchronisation der Prozesse voraus (z.B. mittels Semaphoren).
- Shared Memory Segmente sind i.d.R. die effizienteste Form der Interprozesskommunikation.
- Maximale Anzahl von Shared Memory Segmenten im System (`SHMMNI`) und je Prozess (`SHMSEG`) und deren maximale und minimale Größe (`SHMMAX`, `SHMMIN`) werden bei der Konfiguration des BS-Kerns festgelegt.
- Shared Memory Segmente überleben, wie die anderen System V IPC-Objekte die sie erzeugenden Prozesse und sind ohne besondere Maßnahmen erst mit einem Neustart des Systems zerstört. Zur Information über die existierenden IPC-Objekte bzw. zu ihrer Zerstörung existieren die Kommandos `ipcs` bzw. `ipcrm`.
  (Bitte am Ende jeder Praktikumssitzung zum Aufräumen nutzen !)

Dienste für Shared Memory Segmente (2)

Operationen:

- Header-Dateien: `<sys/types.h>`, `<sys/ipc.h>`, `<sys/shm.h>`
- `int shmget(key_t key, int size, int shmflag)`

erzeugt oder öffnet ein Shared Memory Segment mit dem externen Schlüssel `key` und liefert den zugehörenden Identifier `shmid` des Segments zurück (oder `-1` bei Fehler). Die Behandlung von `key` und `shmflag` ist insgesamt analog wie bei Semaphoren.
Ist insbesondere `key` auf `IPC_PRIVATE` gesetzt, so wird ein privates Segment (ohne öffentlichen key) erzeugt, das für verwandte Prozesse genutzt werden kann.
Wird ein Schlüssel verwendet und in `shmflag` das `IPC_CREAT`-Bit gesetzt und existiert noch kein IPC-Objekt zu `key`, so wird ein neues Segment der Mindestgröße `size` erzeugt.
Für das Öffnen eines existierenden Objekts darf das `IPC_CREAT`-Flag nicht gesetzt sein. Hat `size` einen Wert größer als die aktuelle Größe des Segments, so schlägt der Aufruf fehl, der Wert `0` akzeptiert das Segment in der aktuellen Größe.
Die Zugriffsrechte für das Segment werden durch einen entsprechenden `mode` in `shmflag` bei der Erzeugung mit festgelegt, typisch `0600` für Lesen und Schreiben durch alle Prozesse des Eigentümers des Segments oder `0666` für beliebige Prozesse. Beim Öffnen eines Segments gibt der Zugriffsmodus die geforderten Rechte an.

Dienste für Shared Memory Segmente (3)

Hochschule RheinMain

- `void * shmat(int shmid, void * addr, int flag)`

blendet das Segment `shmid` in den Adressraum des Aufrufers an der virtuellen Adresse `addr` ein (attach). Wird für `addr` der Wert `NULL` gewählt, so wird die Startadresse des Segments im Adressraum vom Betriebssystem festgelegt (Konflikt ausgeschlossen), ansonsten an der spez. Adresse, die i.d.R. innerhalb des Anwenderadressbereichs auf Seitengrenze ausgerichtet sein muss. Die getroffene Startadresse wird zurückgegeben (oder `-1` bei Fehler). Ist in `flag` `SHM_RDONLY` gesetzt, so wird das Segment nur mit Leserecht in den Adressraum eingeblendet. `shmat` muss nach Ausführung von `shmget` und vor der eigentlichen Nutzung des Segments ausgeführt werden.

- `int shmdt(void * addr)`

blendet ein zuvor mittels `shmat` an die Adresse `addr` eingeblendetes Segment wieder aus dem Adressraum des Aufrufers aus (detach). Die Existenz des Shared Memory Segments bleibt hiervon unberührt.

Dienste für Shared Memory Segmente (4)

- `int shmctl(int shmid, int cmd, struct shmid_ds * arg)`

ermittelt / ändert den Status eines Shared Memory Segments oder löscht es. Kommandos (Werte für `cmd`) sind:

- `IPC_STAT` (Abfragen des Status mit Ergebnis in einer Stuktur `shmid_ds`, auf die `arg` zeigt).
- `IPC_SET` (Setzen der Eigentümer-`UID`/`GID` und der Zugriffsrechte bei passender effektiver `UID`).
- `IPC_RMID` (Löschen des Segments bei passender effektiver `UID`; wird effektiv, wenn der letzte Prozess `shmdt` ausführt, zwischenzeitlich ist kein `shmat` mehr möglich, Referenzzähler).
- `SHM_LOCK` (Sperren des Segments, nur durch Prozess mit root-Recht).
- `SHM_UNLOCK` (Entsperren des Segments, nur durch Prozess mit root-Recht).

Dienste für Memory Mapped Files (1)

- Einblenden einer Datei oder eines Gerätes in eine Region des Adressraums (Kombination von Systemfunktionen der Speicherverwaltung, des I/O-Systems und des Dateisystems)
- Eingeblendete, zuvor geöffnete Datei wird wie normales Shared Memory Datensegment zugegriffen, d.h. ohne Nutzung der Dienste `read`, `write`, `lseek`.
- Daten werden mithilfe des normalen Paging-Mechanismus aus der hinterlagerten Datei herangeschafft.
- Veränderte Seiten der Datei werden irgendwann, bedingt durch Seitenverdrängung, dorthin zurückgeschrieben, wenn der letzte Prozess die zugrundeliegende Datei schließt oder durch `sync`-Vorgänge.

Dienste für Memory Mapped Files (2)

- Vorteile:
  - Vermeiden von Kopiervorgängen.
  - Weniger Kontext-Umschaltungen, da I/O-Systemaufrufe wegfallen.
  - Bequemer Zugriff über Zeiger.
  - Veränderungen in der Datei sind für andere Prozesse, die dieselbe Datei eingeblendet haben, *unmittelbar* sichtbar.
- Die Anwendbarkeit des Einblendvorgangs für zahlreiche Geräteklassen erlaubt den Zugriff auf Hardware-Komponenten (z.B. Geräteregister, Graphik-Bildspeicher, usw.), auch aus einem Anwendungsprogramm heraus (hohe Flexibilität, hohe Performance).

Dienste für Memory-Mapped Files (3)

Operationen (Überblick):

- Header-Dateien: `<sys/types.h>`, `<sys/mman.h>`
- `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)`

blendet ein Fenster der durch `fd` angegebenen, geöffneten Datei (oder Gerät), beginnend in der Datei an der Stelle `offset`, in der Länge `len` in den Adressraum des Aufrufers an der virt. Adresse `addr` mit der Rechtefestlegung `prot` ein.

- `int munmap(caddr_t addr, size_t len):`

blendet eine zuvor eingeblendete Datei für der angegebenen Adressbereich aus.

- `int msync(caddr_t addr, size_t len, int flags)`

erlaubt in durch `flags` festlegbaren Varianten, den Inhalt der im angegebenen Adressbereich eingeblendeten Datei auf die Originaldatei zurückzuschreiben.

Dienste zur Adressraum-Manipulation

- In neueren UNIX-Varianten, die gewisse Echtzeitanforderungen erfüllen, wurden auch Funktionen zur Speicherkontrolle bereitgestellt, etwa um sicherzustellen, dass der BS-Kern einem Echtzeitprozess nicht unfreiwillig Seiten verdrängt und dadurch beim nächsten Zugriff (relativ lange) Pagein-Zeiten anfallen.
- Operationen
  - `memctl(...)`
  - `memprotect(...)`
  - `mincore(...)`
- Hier nicht weiter betrachtet.

Zusammenfassung

Was haben wir in Kap. 9 gemacht?

- Konzepte der Speicherverwaltung (Wichtig!).
- Statische Speicherverwaltung als einfachste Verfahren.
- Swapping unterstützt Systeme, deren Prozesse mehr Speicher benötigen als insgesamt zur Verfügung steht, lagern dabei aber stets ganze Prozesse aus und ein.
- Freispeicherverwaltung im Arbeitsspeicher und auf der Platte geschieht durch Verfahren basierend auf Bitmaps, verketteten Listen oder dem Buddy-System.
- Virtueller Speicher unterstützt Systeme, deren Prozesse sehr groß werden können.
- Paging und entsprechende Hardware-Unterstützung (Rechnerarchitektur) wurden grundlegend besprochen.

Zusammenfassung (2)

- Seitenersetzungsalgorithmen wurden vielfach untersucht. Clock-Algorithmus und Aging sind praktikabel und brauchbar, der optimale Algorithmus und LRU sind dagegen gut, aber nicht praktikabel.
- In konkreten Paging-Systemen gewinnen weitere Aspekte an Bedeutung, etwa die Abwägung des Working Set Modells gegenüber reinem Demand Paging oder lokale gegenüber globaler Seitenersetzung.
- Abschließend wurden die UNIX-Systemaufrufe zur Speicherverwaltung besprochen.