

# Systemprogrammierung

## Adressraum

6. Juli 2010

# Überblick

## Adressraum

Ausgangspunkt

Physikalischer Adressraum

Logischer Adressraum

Virtueller Adressraum

Zusammenfassung

Bibliographie

# Grundlagen der Rechnerarchitektur, 2. Semester

Ergänzung, Verfeinerung bzw. Vertiefung von GRA



- ▶ Multi-Tasking, **MMU** [1], speziell...
  - ✓ Segmentierung
  - ✓ Memory Management Unit
  - ✓ Adressumsetzung

... beleuchtet im Kontext der Maschinenprogrammzebene, genauer:

- ▶ Artefakte der Hardware
- ▶ Abstraktion durch ein Betriebssystem

# Adressraumkonzepte

## Betriebssystemansicht

**physikalischer Adressraum**  $\models$  nicht-linear adressierbarer E/A- und Speicherbereich, dessen Größe der Adressbreite der CPU entspricht

- ▶  $2^N$  Bytes, bei einer Adressbreite von  $N$  Bits
- ▶ von Lücken durchzogen  $\rightsquigarrow$  **ungültige Adressen**

**logischer Adressraum**  $\models$  linear adressierbarer Speicherbereich von  $2^M$  Bytes bei einer Adressbreite von  $N$  Bits

- ▶  $M = N$  z.B. im Fall einer *Harvard-Architektur*
  - ▶ getrennter Programm-, Daten- und E/A-Adressraum
- ▶  $M < N$  sonst

**virtueller Adressraum**  $\models$  logischer Adressraum, der  $2^K$  Bytes umfasst

- ▶  $K > N$  bei Speicherbankumschaltung, Überlagerungstechnik
- ▶  $K \leq N$  sonst

# Adressraumorganisation

Adressenbelegung (engl. *address assignment*)

Adressbereich	Größe (KB)	Verwendung
00000000–0009ffff	640	RAM (System)
000a0000–000bffff	128	Video RAM
000c0000–000c7fff	32	BIOS Video RAM
000c8000–000dffff	96	keine
000e0000–000effff	64	BIOS Video RAM ( <i>shadow</i> )
000f0000–000fffff	64	BIOS RAM ( <i>shadow</i> )
00100000–090fffff	147456	RAM (Erweiterung)
09100000–fffdffff	4045696	keine
fffe0000–fffeffff	64	SM-RAM ( <i>system management</i> )
ffff0000–ffffffffff	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

# Ungültige Adressen

Zugriff  $\leadsto$  **Busfehler** (engl. *bus error*)

Adressbereich	Größe (KB)	Verwendung
00000000–0009ffff	640	RAM (System)
000a0000–000bffff	128	Video RAM
000c0000–000c7fff	32	BIOS Video RAM
<b>000c8000–000dffff</b>	<b>96</b>	<b>keine</b>
000e0000–000effff	64	BIOS Video RAM ( <i>shadow</i> )
000f0000–000fffff	64	BIOS RAM ( <i>shadow</i> )
00100000–090fffff	147456	RAM (Erweiterung)
<b>09100000–fffdffff</b>	<b>4045696</b>	<b>keine</b>
fffe0000–fffeffff	64	SM-RAM ( <i>system management</i> )
ffff0000–ffffffffff	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

# Reservierte Adressen

Zugriff  $\leadsto$  **Schutzfehler** (engl. *protection fault*)

Adressbereich	Größe (KB)	Verwendung
00000000–0009ffff	640	RAM (System)
000a0000–000bffff	128	Video RAM
000c0000–000c7fff	32	BIOS Video RAM
000c8000–000dffff	96	keine
000e0000–000effff	64	BIOS Video RAM ( <i>shadow</i> )
000f0000–000fffff	64	BIOS RAM ( <i>shadow</i> )
00100000–090fffff	147456	RAM (Erweiterung)
09100000–fffdffff	4045696	keine
fffe0000–fffeffff	64	SM-RAM ( <i>system management</i> )
ffff0000–ffffffffff	64	BIOS ROM

Toshiba Tecra 730CDT, 1996

# Freie Adressen

Hauptspeicher (engl. *main memory*)

Adressbereich	Größe (KB)	Verwendung
00000000–0009ffff	640	RAM (System)
000a0000–000bffff	128	Video RAM
000c0000–000c7fff	32	BIOS Video RAM
000c8000–000dffff	96	keine
000e0000–000effff	64	BIOS Video RAM ( <i>shadow</i> )
000f0000–000fffff	64	BIOS RAM ( <i>shadow</i> )
00100000–090fffff	147456	RAM (Erweiterung)
09100000–fffdffff	4045696	keine
fffe0000–fffeffff	64	SM-RAM ( <i>system management</i> )
ffff0000–ffffffffff	64	BIOS ROM

Toshiba Tecra 730CDT, 1996



# Segmentierung

Logische Unterteilung von Programmadressräumen

Ebene<sub>4</sub>-Programme sind in (mind.) zwei Segmente logisch aufgeteilt:

- ▶ **Text** → Maschinenanweisungen, Programmkonstanten
- ▶ **Daten** → initialisierte Daten, globale Variablen, Halde

Ebene<sub>3</sub>-Programme kennen (mind.) ein weiteres Segment:

- ▶ **Stapel** → lokale Variablen, Hilfsvariablen, aktuelle Parameter

Betriebssysteme verwalten diese Segmente im physikalischen Adressraum

- ▶ ggf. mit Hilfe einer MMU (engl. *memory management unit*)
  - ▶ Hardware, die nur logische in physikalische Adressen umsetzt
  - ▶ für die Verwaltung des Speichers ist das Betriebssystem verantwortlich
- ▶ die MMU legt eine **Organisationsstruktur** auf den phys. Adressraum
  - ▶ sie unterteilt ihn in Seiten fester oder Segmente variabler Länge

# Ausprägungen von Programmadressräumen

## Seitennummerierter oder segmentierter Adressraum

eindimensional in **Seiten** aufgeteilt (engl. *paged*)

- ▶ eine Programmadresse  $A_P$  bildet ein Tupel  $(p, o)$ :

$$p = A_P \operatorname{div} 2^N \rightsquigarrow \text{Seitennummer (engl. page number)}$$

$$o = A_P \operatorname{mod} 2^N \rightsquigarrow \text{Versatz (engl. byte offset)}$$

- ▶ mit  $2^N$  gleich der Seitengröße (engl. *page size*) in Bytes
- ▶ Seite  $\mapsto$  **Seitenrahmen** (auch: Kachel) des phys. Adressraums

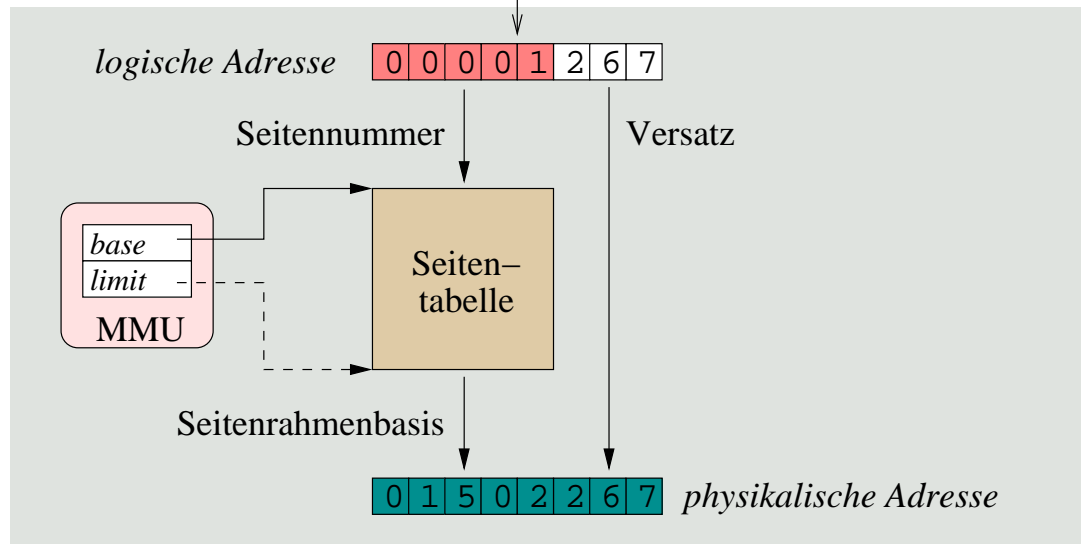
zweidimensional in **Segmente** aufgeteilt (engl. *segmented*)

- ▶ eine Programmadresse  $A_S$  bildet ein Paar  $(S, A)$ 
  - ▶ mit der Adresse  $A$  relativ zu Segment(name/nummer)  $S$
  - ▶ bei seitennummerierten Segmenten wird  $A$  als  $A_P$  interpretiert
- ▶ Segment  $\mapsto$  Folge von Bytes/Seitenrahmen des phys. Adressraums

# Adressumsetzung

Seitennumerierter Adressraum (engl. *paged address space*)

```
char *p = 4711; /* 0x1267 */
```



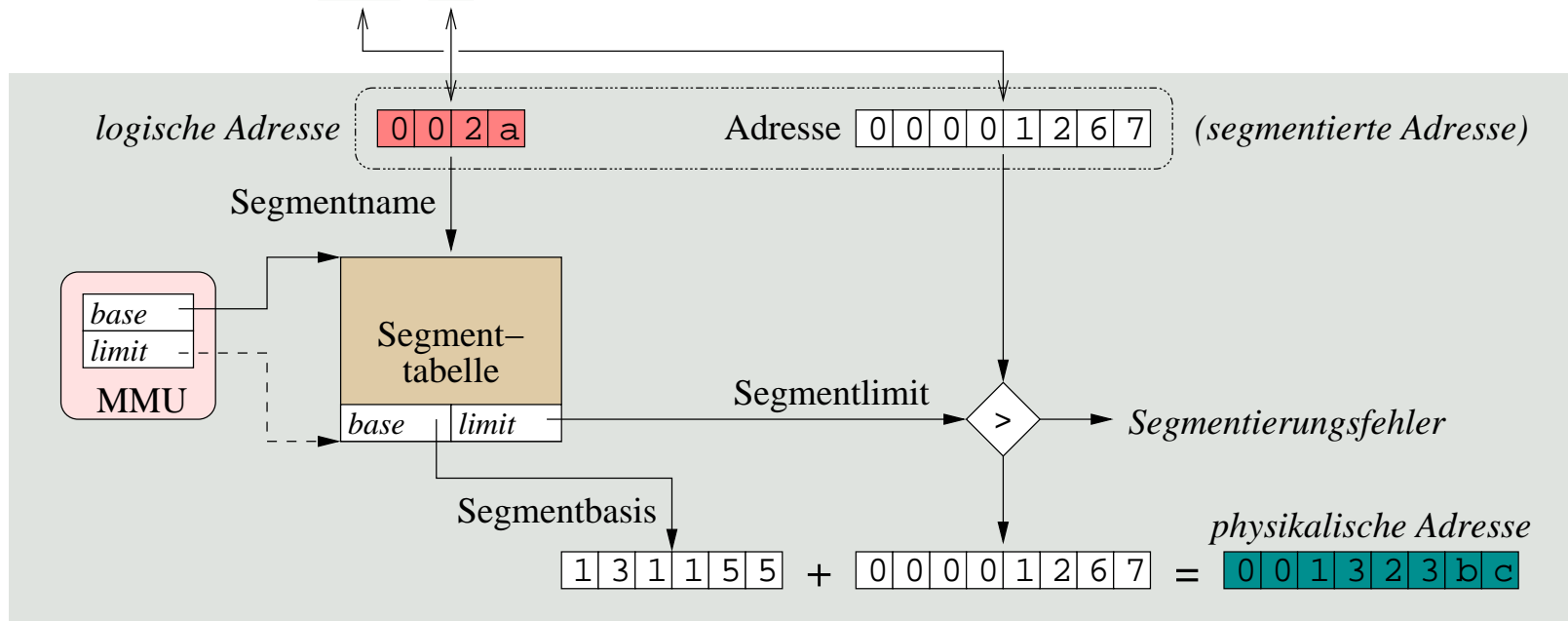
- ▶ Seitennummer ist Index in Seitentabelle
  - ▶ pro Prozess
  - ▶ dimensioniert durch die MMU
- ▶ **ungültiger Index** führt zum *Trap*

- ▶ die indizierte Adressierung (der MMU) liefert einen Seitendeskriptor
  - ▶ enthält die **Seitenrahmennummer**, die die Seitennummer ersetzt
  - ▶ entspricht der **Basisadresse des Seitenrahmens** im phys. Adressraum
- ▶ setzen der Basis-/Längenregister der MMU  $\rightsquigarrow$  Adressraumwechsel

# Adressumsetzung (Forts.)

Segmentierter Adressraum (engl. *segmented address space*)

```
char *p = 4711@42; /* 0x1267@0x2a */
```

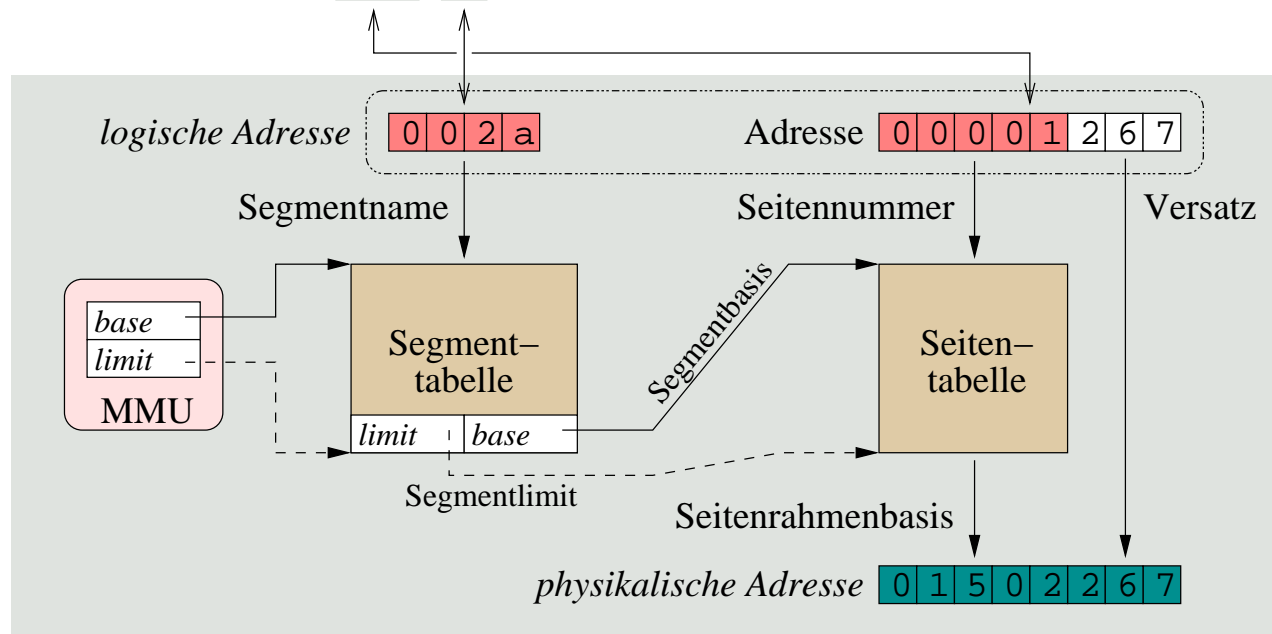


- ▶ Segmentname ist Index in die Segmenttabelle eines Prozesses
  - ▶ dimensioniert durch die MMU, ungültiger Index führt zum Trap
- ▶ indizierte Adressierung (der MMU) ergibt den Segmentdeskriptor
  - ▶ enthält Basisadresse und Länge des Segments (engl. base/limit)
  - ▶  $address_{phys} = address > limit ? Trap : base + address$

# Adressumsetzung (Forts.)

Segmentierter seitennummerierter Adressraum (engl. *page-segmented address space*)

```
char *p = 4711@42; /* 0x1267@0x2a */
```

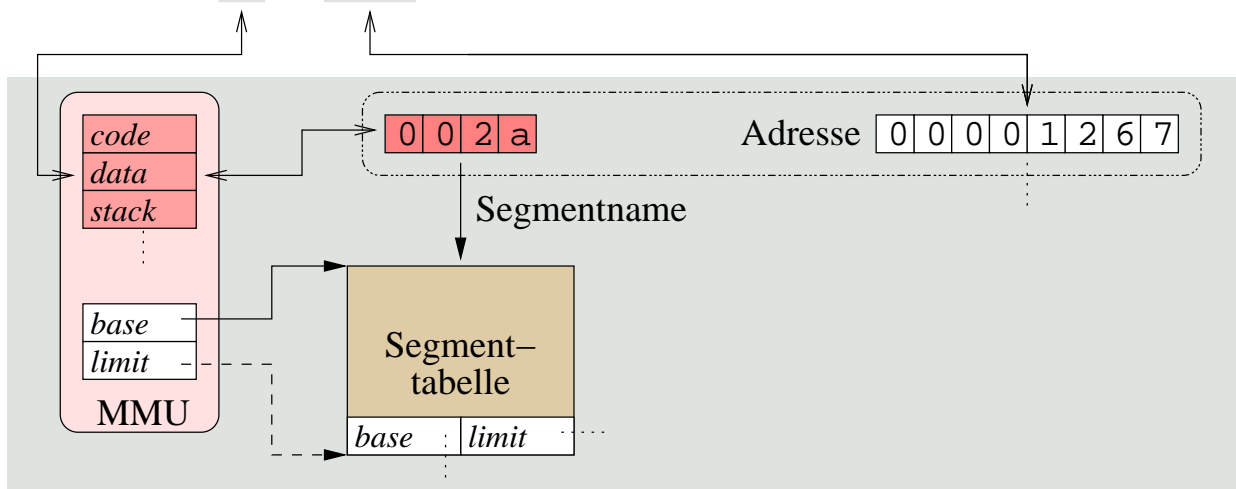


- ▶ **Reihenschaltung** von zwei Adressumsetzungseinheiten der MMU:
  - Segmenteinheit** löst eine segmentierte Adresse auf
    - ▶ adressiert und begrenzt die Seitentabelle
  - Seiteneinheit** generiert die physikalische Adresse
- ▶ jeder Prozess hat (mind.) eine Segment- und eine Seitentabelle

# Adressumsetzung (Forts.)

Segmentregister bzw. Segmentselektor (engl. *segment selector*)

```
char *p = 4711; /* 0x1267@data */
```



- ▶ je nach Art des Speicherzugriffs selektiert die MMU implizit das passende Segment

Befehlsabruf (engl. *instruction fetch*)  $\mapsto$  *code* ✓

Operandenabruf (engl. *operand fetch*)  $\rightsquigarrow$  Text, Daten oder Stapel ?

- ▶ Direktwerte  $\mapsto$  *code*
- ▶ globale/lokale Daten  $\mapsto$  *data/stack*

▶ `char *foo (char bar) { return &bar; }`

☞ Programme können weiterhin 1-dimensionale log. Adressen verwenden

# Seiten- bzw. Segmentdeskriptor

Abbildung steuernder Verbund

Adressumsetzung basiert auf **Deskriptoren** der MMU, die für jede Seite/Segment eines Prozesses Relokations- und Zugriffsdaten verwalten

- ▶ die **Basisadresse** des Seitenrahmens/Segments im phys. Adressraum
- ▶ die **Zugriffsrechte** des Prozesses
  - ▶ lesen (*read*), schreiben (*write*), ggf. ausführen (*execute*)

Segmente sind (im Geg. zu Seiten) von variabler, dynamischer Größe und benötigen daher zusätzliche Verwaltungsdaten  $\leadsto$  **Segmentdeskriptor**

- ▶ die **Segmentlänge**, um Segmentverletzungen abfangen zu können
  - ▶ Basis-/Längenregister (S. 6-32)  $\subset$  Segmentdeskriptor
- ▶ die **Expansionsrichtung**: Halde „*bottom-up*“, Stapel „*top-down*“

**Deskriptorprogrammierung** erfolgt zur Programmlade- und -laufzeit

- ▶ bei Erzeugung/Zerstörung schwer- und leichtgewichtiger Prozesse
- ▶ bei Anforderung/Freigabe von Arbeitsspeicher

# Seiten- bzw. Segmenttabelle

Adressraum beschreibende Datenstruktur

Deskriptoren des Adressraums eines Prozesses sind in einer **Tabelle** im Arbeitsspeicher zusammengefasst

- ▶ die **Arbeitsmenge** (engl. *working set*) von Deskriptoren eines Prozesses wird im Zwischenspeicher (engl. *cache*) gehalten
  - ▶ TLB (engl. *translation lookaside buffer*) der MMU
- ▶ Adressraumwechsel als Folge eines Prozesswechsels bedeutet:
  1. zerstören der Arbeitsmenge (TLB „*flush*“; teuer, schwergewichtig)
  2. Tabellenwechsel (Zeiger umsetzen; billig, federgewichtig)

**Basis-/Längenregister** (engl. *base/limit register*)

- ▶ beschreibt eine Tabelle und damit exakt einen Prozessadressraum
- ▶ bei der Adressumsetzung wird eine **Indexprüfung** durchgeführt:
  - ▶  $descriptor = index \leq limit ? \&base[index] : Trap$
  - ▶ wobei *index* die Seitennummer/den Segmentnamen repräsentiert



# Adressraumabbildung einhergehend mit Ein-/Ausgabe

Integration von Vorder- und Hintergrundspeicher

**Abstraktion** von Größe und Örtlichkeit des verfügbaren Hauptspeichers

- ▶ vom Prozess nicht benötigte Programmteile können ausgelagert sein
  - ▶ sie liegen im **Hintergrundspeicher**, z.B. auf der Festplatte
- ▶ der Prozessadressraum könnte über ein Rechnernetz verteilt sein
  - ▶ Programmteile sind über die Hauptspeicher anderer Rechner verstreut

Zugriffe auf ausgelagerte Programmteile fängt der Prozessor ab: *Trap*

- ▶ sie werden stattdessen **partiell interpretiert** vom Betriebssystem
- ▶ der unterbrochene Prozess wird in einen E/A-Stoß gezwungen
  - ▶ er erwartet die erfolgreiche Einlagerung eines Programmteils
  - ▶ ggf. sind andere Programmteile aus dem Hauptspeicher zu verdrängen
- ▶ Wiederaufnahme des CPU-Stoßes  $\rightsquigarrow$  Wiederholung des Zugriffs

# Seiten- bzw. Segmentdeskriptor (Forts.)

## Zusätzliche Attribute

Adressumsetzung unterliegt einer **Steuerung**, die „transparent“ für den zugreifenden Prozess die Einlagerung auslöst

- ▶ die **Gegenwart** eines Segments/einer Seite wird erfasst: *present bit*
  - 0  $\mapsto$  ausgelagert; *Trap*, partielle Interpretation, Einlagerung
    - ▶ die Basisadresse ist eine Adresse im Hintergrundspeicher
    - ▶ wird nach der Einlagerung vom Betriebssystem auf 1 gesetzt
  - 1  $\mapsto$  eingelagert; Befehl abrufen, Operanden lesen/schreiben
    - ▶ die Basisadresse ist eine Adresse im Vordergrundspeicher
    - ▶ wird nach der Auslagerung vom Betriebssystem auf 0 gesetzt
- ▶ das „Gegenwartsbit“ dient verschiedentlich noch anderen Zwecken:
  - ▶ um z.B. Zugriffe zu zählen oder ihnen einen Zeitstempel zu geben
  - ▶ Betriebssystemmaßnahmen zur Optimierung der Ein-/Auslagerung

# Zugriffsfehler

Seitenfehler (engl. *page fault*) bzw. Segmentfehler (engl. *segment fault*)

*present bit* = 0 je nach Befehlssatz und Adressierungsarten der CPU kann der **Behandlungsaufwand** im Betriebssystem und somit der **Leistungsverlust** für beträchtlich sein

```
void hello () {  
    printf("Hi!\n");  
}  
  
void (*moin)() = &hello;  
  
main () {  
    (*moin)();  
}
```

```
main:  
    pushl %ebp  
    movl  %esp,%ebp  
    pushl %eax  
    pushl %eax  
    andl  $-16,%esp  
    call  *moin  
    leave  
    ret
```

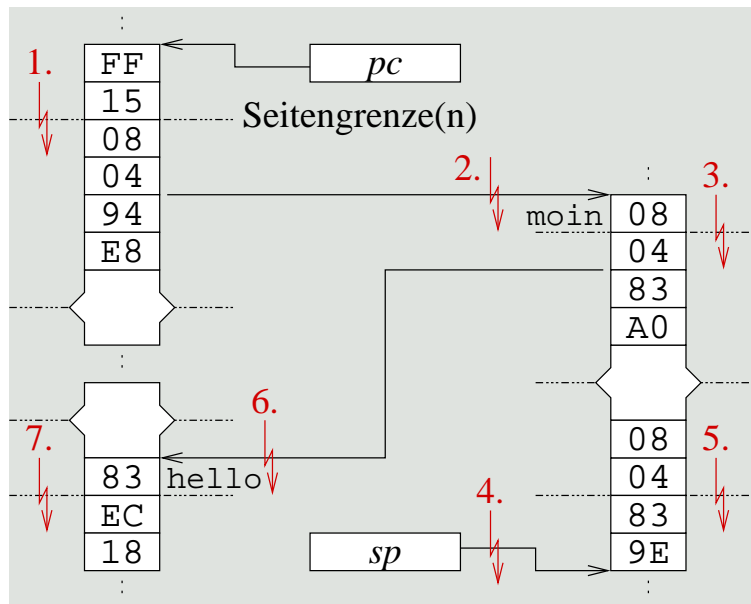
```
⋮  
FF15080494E8  
⋮
```

# Zugriffsfehler (Forts.)

Beispiel des schlimmsten Falls eines Seitenfehlers. . .

`call *moin` (x86) Aufruf einer indirekt adressierten Prozedur

► der Operationskode (FF 15) wurde bereits gelesen



1. Operandenadresse holen (08 04 94 E8)
2. Funktionszeiger lesen (08)
3. Funktionszeiger weiterlesen (04 83 A0)
4. Rücksprungadresse stapeln (08 04)
5. Rücksprungadresse weiterstapeln (83 9E)
6. Operationskode holen (83)
7. Operanden holen (EC 18)

► Seitenfehler 6. und 7. sind eigentlich bereits der Ausführung des ersten Maschinenbefehls der aufgerufenen Prozedur zuzurechnen

# Aufwandsabschätzung eines Seitenfehlers

Seitenfehler sind nicht-funktionale Programmeigenschaften

effektive Zugriffszeit (*effective access time, eat*) auf den Hauptspeicher

- ▶ hängt stark ab von der **Seitenfehlerwahrscheinlichkeit** ( $p$ ) und verhält sich direkt proportional zur **Seitenfehlerrate**:

$$eat = (1 - p) \cdot pat + p \cdot pft, 0 \leq p \leq 1$$

- ▶ angenommen, folgende Systemparameter sind gegeben:
  - ▶ 50 ns Zugriffszeit auf den RAM (*physical access time, pat*)
  - ▶ 10 ms mittlere Zugriffszeit auf eine Festplatte (*page fault time, pft*)
  - ▶ 1 % Wahrscheinlichkeit eines Seitenfehlers ( $p = 0,01$ )
- ▶ dann ergibt sich:  
 $eat = 0,99 \cdot 50 \text{ ns} + 0,01 \cdot 10 \text{ ms} = 49,5 \text{ ns} + 10^5 \text{ ns} \approx 0,1 \text{ ms}$



Einzelzugriffe sind im Ausnahmefall um den Faktor 2000 langsamer

# Aufwandsabschätzung eines Seitenfehlers (Forts.)

Seitenfehler sind nicht wirklich transparent

mittlere Zugriffszeit (*mean access time, mat*) auf den Hauptspeicher

- ▶ hängt stark ab von der effektiven **Seitenzugriffszeit** und der **Seitengröße** (in Bytes pro Seite bzw. Seitenrahmen):

$$mat = (eat + (sizeof(page) - 1) \cdot pat) / pat$$

- ▶ angenommen, folgende Systemparameter sind gegeben:
  - ▶ Seitengröße von 4 096 Bytes (4 KB)
  - ▶ 50 ns Zugriffszeit (*pat*) auf ein Byte im RAM
  - ▶ effektive Zugriffszeit (*eat*) wie eben berechnet bzw. abgeschätzt

- ▶ dann ergibt sich:

$$mat = (eat + 4\,095 \cdot 50\text{ ns}) / 50\text{ ns} = 6\,095,99\text{ ns} \approx 6\ \mu\text{s}$$



Folgezugriffe sind im Ausnahmefall um den Faktor 122 langsamer

# Seitenüberlagerung „*Considered Harmful*“

## Pro und Contra

**Virtuelle Adressräume** sind ...

**vorteilhaft** wenn übergroße bzw. gleichzeitig mehrere Programme in Anbetracht zu knappen Hauptspeichers auszuführen sind

**ernüchternd** wenn der eben durch die Virtualisierung bedingte Mehraufwand zu berücksichtigen ist und sich für ein gegebenes Anwendungsszenario als problematisch bis unakzeptabel erweisen sollte

**Seitenfehler** sind ...

- ▶ nicht wirklich transparent, wenn zeitliche Aspekte relevant sind
  - ▶ z.B. im Fall von Echtzeitverarbeitung oder Hochleistungsrechnen
- ▶ erst zur Laufzeit ggf. entstehende nicht-funktionale Eigenschaften

# Adressräume

## Ebenen der Abstraktion

**physikalischer Adressraum** enthält gültige und ungültige Adressen

**ungültige Adressen** Zugriff führt zum Busfehler

**gültige Adressen** Zugriff gelingt, ist jedoch zu bedenken...

- ▶ reservierte Adressbereiche  $\rightsquigarrow$  Schutz

**logischer Adressraum** enthält gültige Adressen

- ▶ **Zugriffsrechte** der Prozesse stecken **Gültigkeitsbereiche** ab
  - ▶ Zugriff auf reservierte Adressen führt ggf. zum Schutzfehler
- ▶ Prozesse sind in ihrem Programmadressraum abgeschottet, isoliert
  - ▶ Zugriff auf „fremde“ freie Adressen führt zum Schutzfehler

**virtueller Adressraum** enthält „flüchtige Adressen“

- ▶ die **Bindung** der Adressen zu den Speicherzellen ist nicht fest
- ▶ sie variiert phasenweise zwischen Vorder- und Hintergrundspeicher



# Adressraumdeskriptoren

## Seitennumerierte und segmentierte Adressräume

Abbildung steuernde **Verbunde** zur Erfassung einzelner Adressraumteile

- ▶ speichern Attribute von **Seiten** oder **Segmente**
  - ▶ d.h., Relokations- und Zugriffsdaten, Zugriffsrechte
- ▶ bilden Seiten fester Größe auf gleichgroße Seitenrahmen ab
  - ▶ seitennumerierter Adressraum
- ▶ bilden Segmente variabler Größe auf Byte- oder Seitenfolgen ab
  - ▶ segmentierter und ggf. seitennumerierter Adressraum

**Abbildungstabellen** fassen Deskriptoren (eines Adressraums) zusammen

- ▶ die Tabellen liegen im Arbeitsspeicher des Betriebssystems
  - ▶ der dafür erforderliche Speicherbedarf kann beträchtlich sein
- ▶ im TLB sind **Arbeitsmengen** von Deskriptoren zwischengespeichert
  - ▶ ein *Cache* der MMU, ohne dem **Adressumsetzung** ineffizient ist

**Zugriffsfehler** sind intransparente, nicht-funktionale Eigenschaften

# Literaturverzeichnis

- [1] Volkmar Sieh.  
Grundlagen der Rechnerarchitektur.  
<http://www3.informatik.uni-erlangen.de/Lehre/GRa/SS2008/sk2008>.  
Lecture Notes.