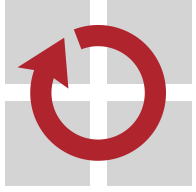
Übung zu Betriebssystemtechnik

Privilegienisolation und Systemaufrufe

2. Mai 2019

Andreas Ziegler
Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

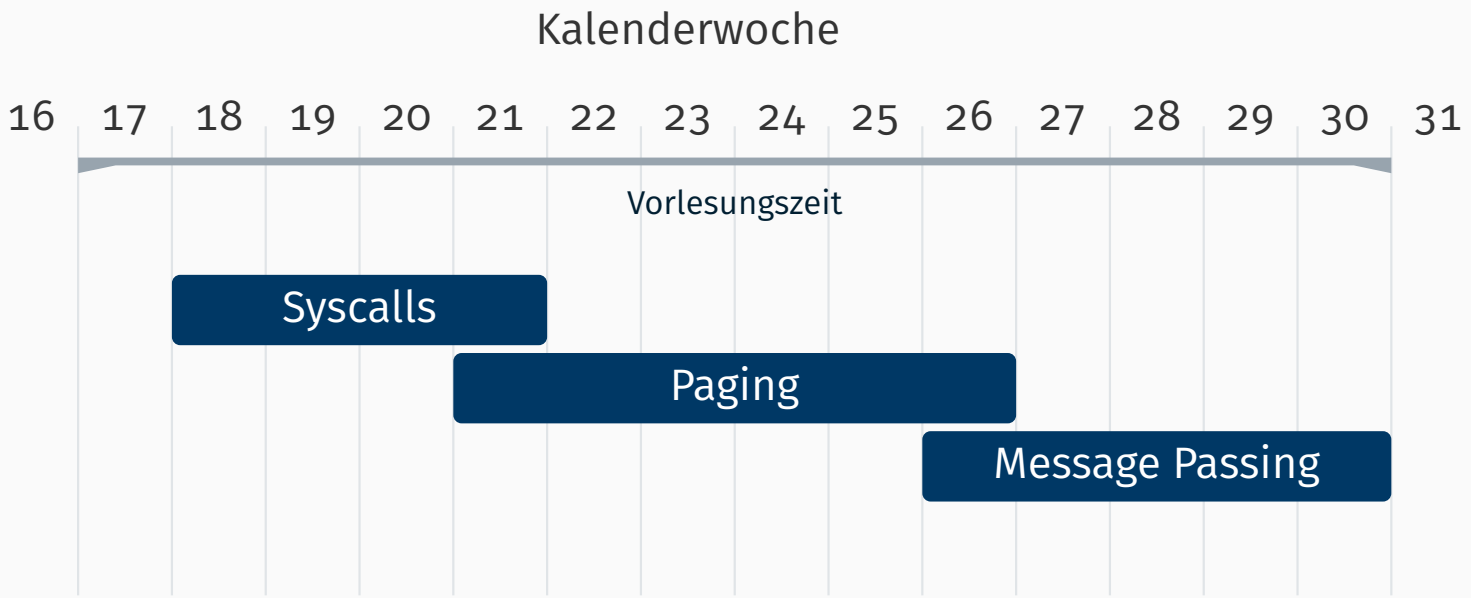
Organisation des Übungsbetriebs

StuBSmI

single-core

5 oder 7.5 ECTS Modul

Zeitplan


Kalenderwoche
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
Vorlesungszeit
Syscalls
Paging
Message Passing

Zeitplan

April 2019

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Mai 2019

		1	**2**	3	4	5
6	**7**	8	**9**	10	11	12
13	**14**	15	**16**	17	18	19
20	**21**	22	**23**	24	25	26
27	**28**	29	30	31		

Juni 2019

					1	2
3	**4**	5	**6**	7	8	9
10	11	12	**13**	14	15	16
17	**18**	19	20	21	22	23
24	**25**	26	**27**	28	29	30

Juli 2019

1	**2**	3	**4**	5	6	7
8	**9**	10	11	12	13	14
15	**16**	17	**18**	19	20	21
22	**23**	24	25	26	27	28
29	30	31				

Tafelübung
im WinCIP (01.153)

Rechnerübung
im WinCIP (01.153)

Abgabe der Aufgabe in der **Rechnerübung**

- Abgabe nur in festen **2er Gruppen**
- eine (obligatorische) Tafelübung pro Aufgabe
- Anmeldung zur Tafelübung (bis 9. Mai) im Waffel auf **waffel.informatik.uni-erlangen.de/signup?course=375**
- Informationen und Aufgabenstellung auf **www4.cs.fau.de/Lehre/SS19/V_BST/**
- Quelltextvorlage der Aufgaben im Gitlab

  `https://gitlab.cs.fau.de/i4/bs/stubsmi`

Die Übung wird zwar durch Folien unterstützt – diese dienen jedoch ausschließlich als ergänzende Veranschaulichung des zu vermittelnden Stoffes und haben somit keinen Anspruch auf Vollständigkeit.
Sie sind nicht zum Selbststudium geeignet und auch nicht auf Druck optimiert!

Hilfe

Selbsthilfe

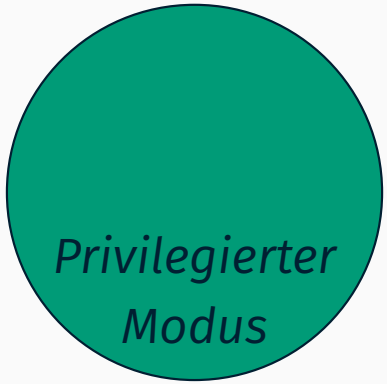
- Aufpassen in der Tafelübung
- Internet (wiki.osdev.org, lowlevel.eu, Stack Overflow)
- Intel Handbuch

Eskalationsstufen

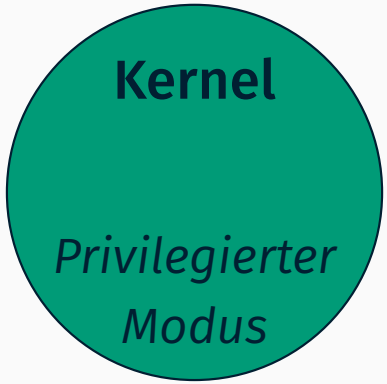
- Rechnerübung
- **#faui4bs** im IRCnet
- XMPP-MUC **i4bs@conference.cs.fau.de**
- Mail an **bsstud@lists.informatik.uni-erlangen.de**
- **Raum 0.055** in der Martensstr. 1 (begründeter Notfall)

Einleitung

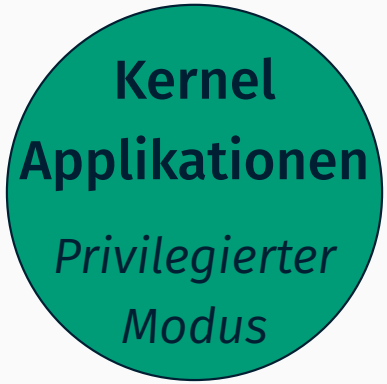
Privilegien in OO-/MPStuBS



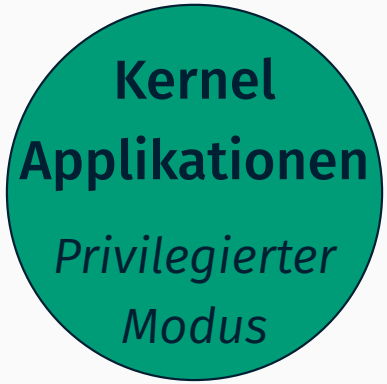
Privilegien in OO-/MPStuBS



Privilegien in OO-/MPStuBS

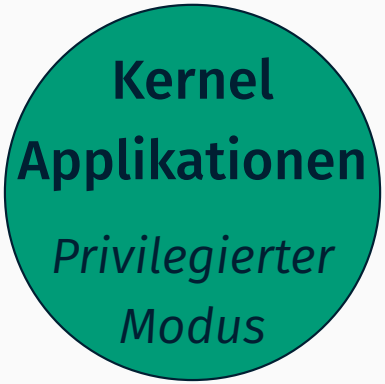


Privilegien in OO-/MPStuBS



Probleme?

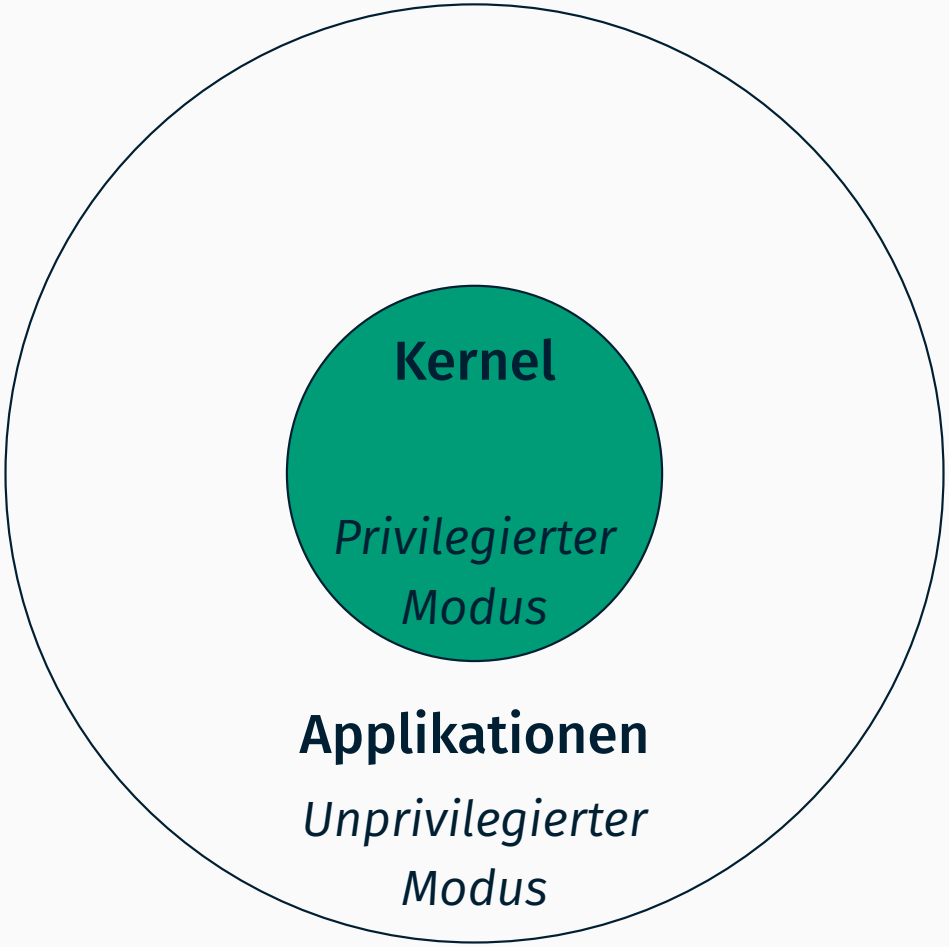
Privilegien in OO-/MPStuBS



Probleme?

- Zugriff auf privilegierte Befehle (`hlt`, `cli`/`sti`, `inb`/`outb`, ...)
- Zugriff auf CPU-Kontrollregister (z.B. MMU-Konfiguration)

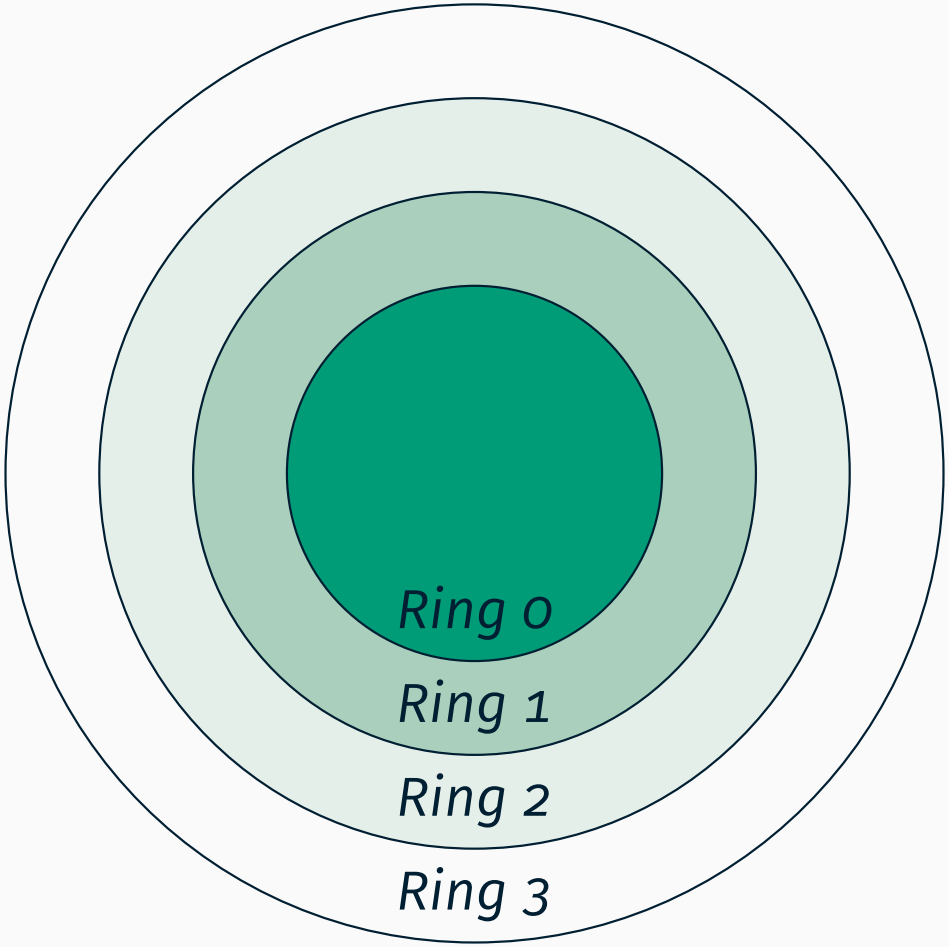
Privilegien in OO-/MPStuBS



Probleme?

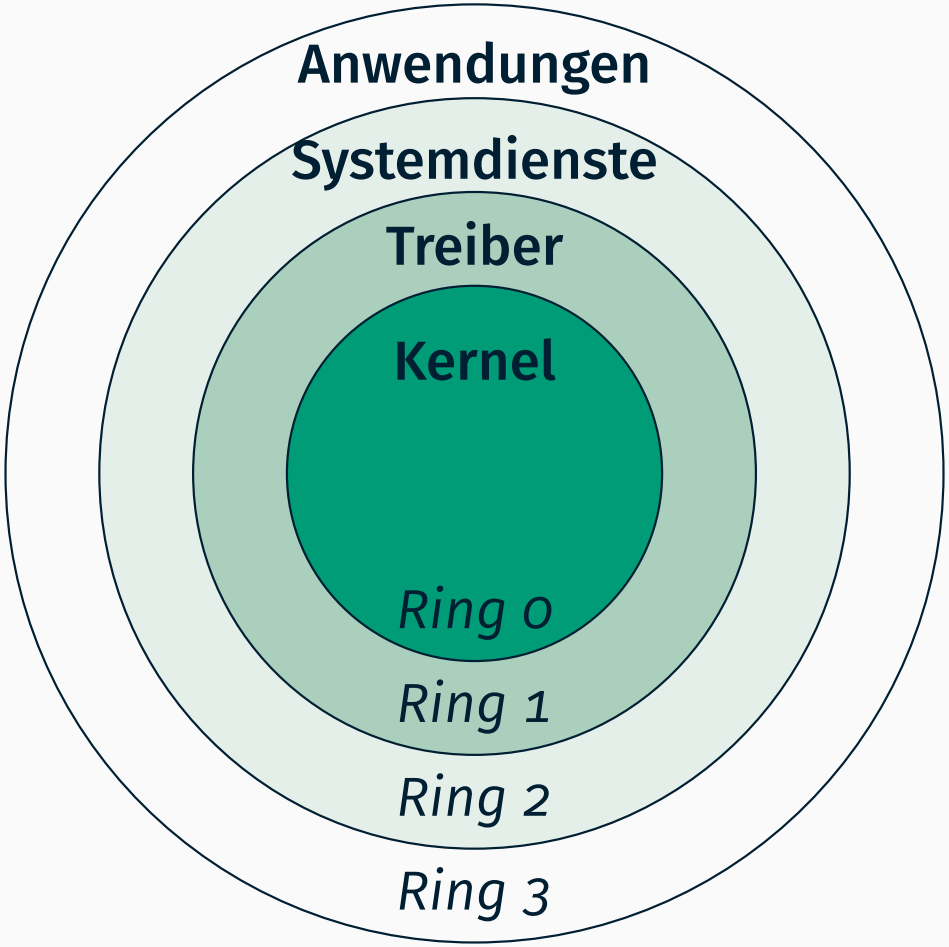
- Zugriff auf privilegierte Befehle (`hlt`, `cli`/`sti`, `inb`/`outb`, ...)
- Zugriff auf CPU-Kontrollregister (z.B. MMU-Konfiguration)

Privilegientrennung auf x86: Schutzringe



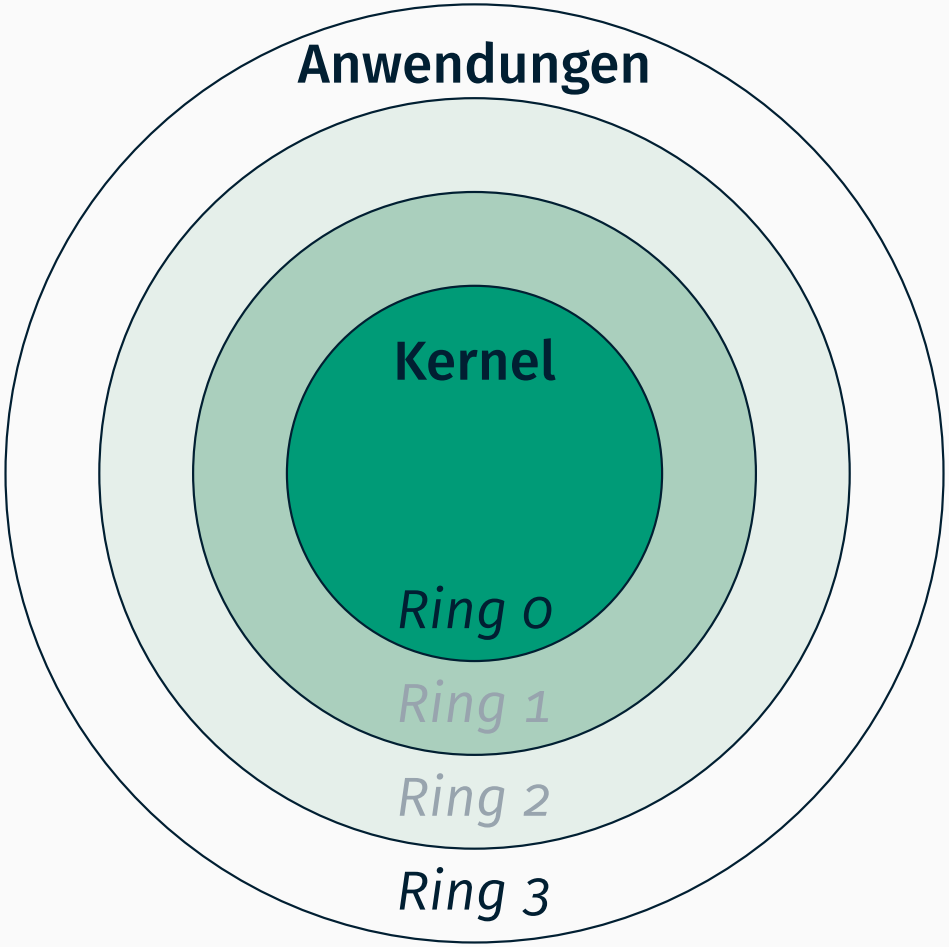
- Schutzringkonzept ursprünglich aus *Multics*
- Auf **x86** insgesamt 4 Ringe

Privilegientrennung auf x86: Schutzringe



- Schutzringkonzept ursprünglich aus *Multics*
- Auf **x86** insgesamt 4 Ringe

Privilegientrennung auf x86: Schutzringe



- Schutzringkonzept ursprünglich aus *Multics*
- Auf **x86** insgesamt 4 Ringe
- Tatsächlich verwendet: Ring 0 (System) und 3 (Benutzer)

Umsetzung

Was ist zu tun?

Schutzringe

- Konfiguration des Prozessors
- Wechsel zwischen den Modi

Was ist zu tun?

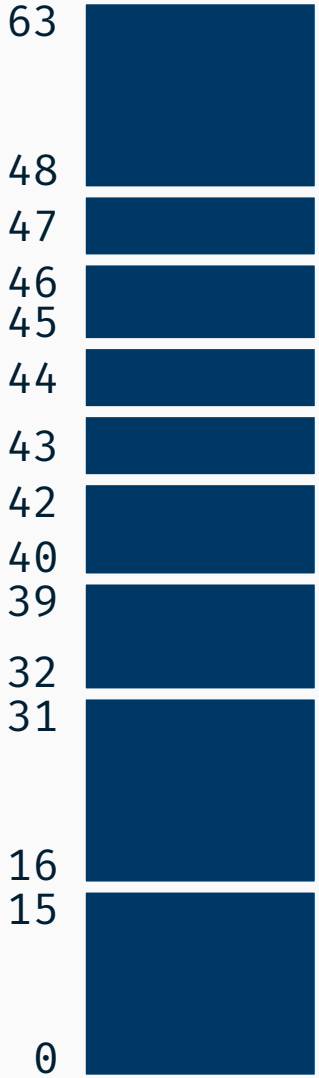
Schutzringe

- Konfiguration des Prozessors
- Wechsel zwischen den Modi

Systemaufrufe

- Erlauben von Traps aus Ring 3
- Übergabe von Argumenten

Interrupt Deskriptor



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_128`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level: Erlaubter Ring

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

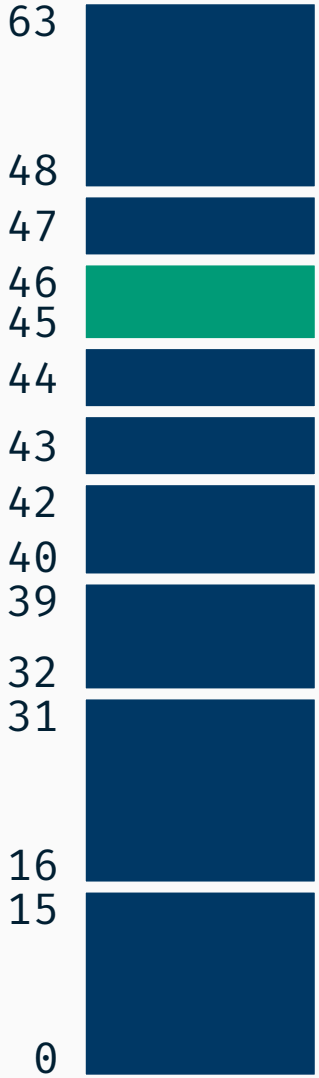
Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

Interrupt Deskriptor



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_128`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level: Erlaubter Ring

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

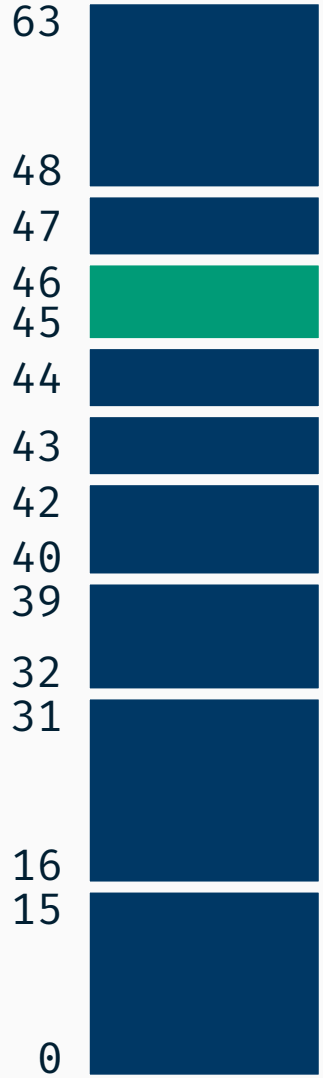
Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

Interrupt Deskriptor (Beispiel)



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_128`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level: Erlaubter Ring

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

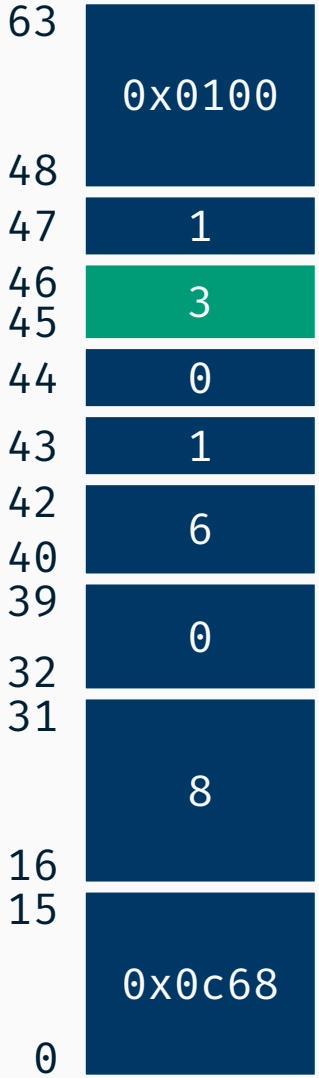
Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für **`100 0c68 <irq_entry_128>`**

Interrupt Deskriptor (Beispiel)



Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_entry_128`)

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level: Erlaubter Ring

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32-bit (`1`)

Type: Task (`5`), Interrupt (`6`) oder Trap (`7`)?

Unused – muss `0` sein

Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)

Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für **`100 0c68 <irq_entry_128>`** → **`0x0100 ee00 0008 0c68`**

Segmente

- Segmentierung tief in X86 verwurzelt
- Segmentregister (`ss`, `cs`, `ds`, ...) beinhalten Selektoren
- Selektoren (16-bit Werte) werden direkt in Register geladen:

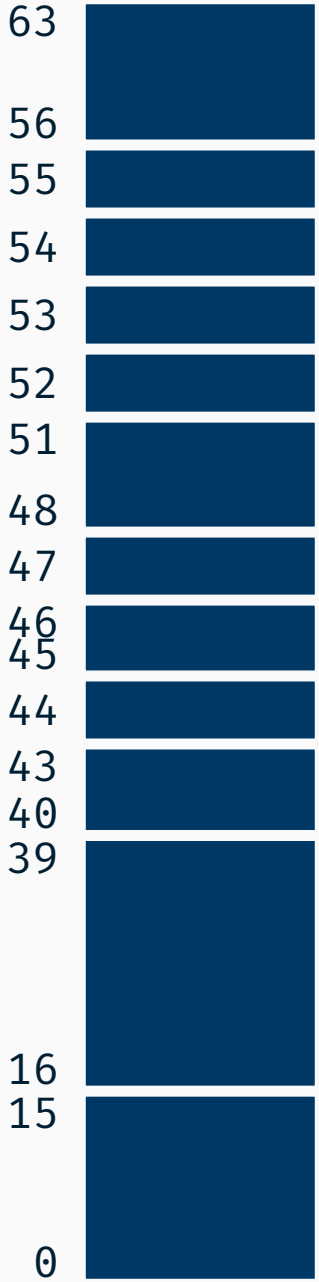
```
mov ax, 0x10
mov ds, ax
```

- Segmentierter Zugriff:

```
mov [fs:eax], 42
```

- Selektoren bestehen aus Segmentdeskriptorindex in GDT und Requested Privilege Level (RPL)

Global Deskriptor Table Entry



Base Address (high): *Oberer* Teil der Startadresse

Granularity: Byte (`0`) oder 4KiB-Block (`1`)
Operation Size: 16-bit (`0`) oder 32-bit (`1`)
64-bit Segment: Bei uns `0` für Protected Mode
Available – zur freien Verwendung

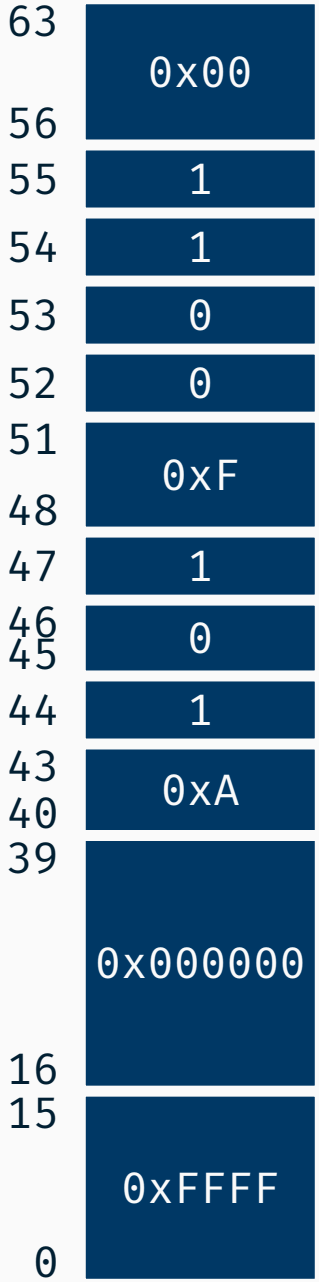
Segment Limit (high): *Oberer* Teil der Segmentgröße

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)
Descriptor Privilege Level: Erlaubter Ring
Deskriptortyp: System (`0`) oder Code/Daten (`1`)
Type: Legt Berechtigungen fest (R/W/X)

Base Address (low): *Unterer* Teil der Startadresse

Segment Limit (low): *Unterer* Teil der Segmentgröße

Global Deskriptor Table Entry



Base Address (high): *Oberer* Teil der Startadresse

Granularity: Byte (`0`) oder 4KiB-Block (`1`)

Operation Size: 16-bit (`0`) oder 32-bit (`1`)

64-bit Segment: Bei uns `0` für Protected Mode

Available – zur freien Verwendung

Segment Limit (high): *Oberer* Teil der Segmentgröße

Present: Eintrag aktiv (`1`) oder inaktiv (`0`)

Descriptor Privilege Level: Erlaubter Ring

Deskriptortyp: System (`0`) oder Code/Daten (`1`)

Type: Legt Berechtigungen fest (R/W/X)

Base Address (low): *Unterer* Teil der Startadresse

Segment Limit (low): *Unterer* Teil der Segmentgröße

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen
- Ring wechseln

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)

- Zustand auf Stack sichern

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)

- Zustand auf Stack sichern

Woher kommt der Stack-Pointer (und -Segment)?

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)

- Zustand auf Stack sichern

Woher kommt der Stack-Pointer (und -Segment)?

```
mov esp, 0x0
int 0x80
```

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)

- Zustand auf Stack sichern

Woher kommt der Stack-Pointer (und -Segment)?

```
mov esp, 0x0
int 0x80
```

Auslösen eines Systemaufrufs

Was passiert bei `int 0x80` aus Ring 3?

Hardware:

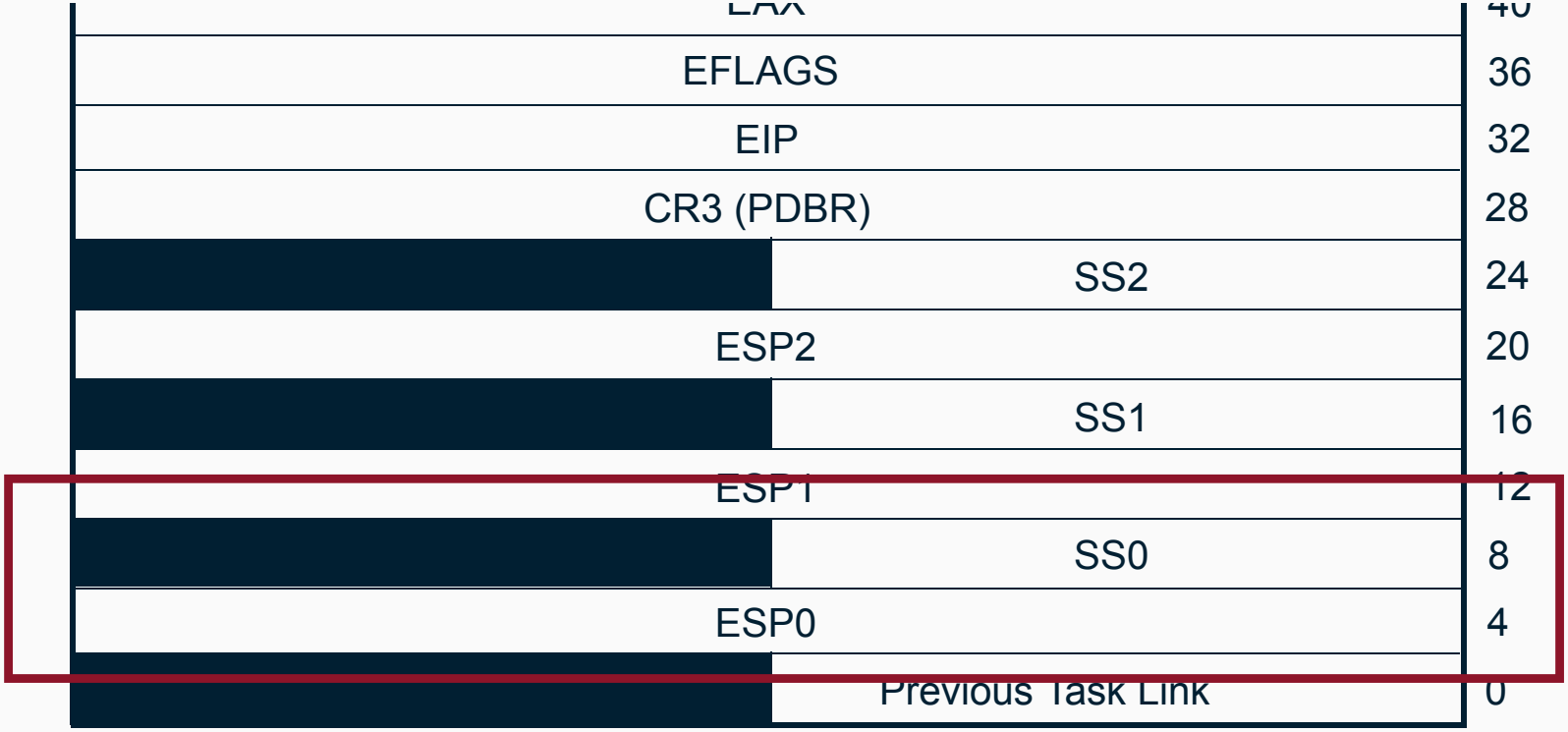
- Instruction Pointer setzen (aus **IDT**)
- Ring wechseln (aus **IDT**)
- **Auf Kernel-Stack wechseln**
- Zustand auf Stack sichern

Woher kommt der Stack-Pointer (und -Segment)?

```
mov esp, 0x0
int 0x80
```

➔ Task State Segment (TSS)

Task State Segment



Intel unterstützt Hardware-Tasks $\Rightarrow$ Zustandssicherung

Ein Deskriptor in der **GDT**, Base = Adresse des **TSS**

Wird mittels `mov ax, <offset>; ltr ax` geladen

Bei uns: **ein** globales **TSS**, jeder Task hat eigenen Kernel Stack

Parameterübergabe

Syscall-Nummer und Parameter in Registern übergeben

Reihenfolge (z. B.): `eax, ecx, edx, ebx, esi, edi`

```
int sys_foo(int p1, int p2, int p3);
```

Parameterübergabe

Syscall-Nummer und Parameter in Registern übergeben

Reihenfolge (z. B.): `eax, ecx, edx, ebx, esi, edi`

```
int sys_foo(int p1, int p2, int p3);
```

Entweder: alles in purem Assembler schreiben

```
[GLOBAL sys_foo]
sys_foo:
    push ebx
    mov eax, FOO_SYSNUM
    mov ecx, [esp + 8]
    mov edx, [esp + 12]
    mov ebx, [esp + 16]
    int 0x42
    pop ebx
    ret
```

Parameterübergabe

Syscall-Nummer und Parameter in Registern übergeben

Reihenfolge (z. B.): `eax, ecx, edx, ebx, esi, edi`

```
int sys_foo(int p1, int p2, int p3);
```

Oder: Systemaufruf (fast) in C mit *Inline Assembly*

```
int sys_foo(int p1, int p2, int p3) {
    int retval;
    asm volatile (
        "int $0x42\n\t"     // Befehle
        : "=a"(retval)      // Outputs
        : "a"(FOO_SYSNUM), "c"(p1), "d"(p2),
          "b"(p3)           // Inputs
        : "memory");        // Clobber
    return retval;
}
```

Systemaufruf auf Kernel-Seite

Was fehlt noch?

Systemaufruf auf Kernel-Seite

Was fehlt noch?

➔ initialer Wechsel in Ring 3

Systemaufruf auf Kernel-Seite

Was fehlt noch?

➔ initialer Wechsel in Ring 3

Ähnlich zu `toc_settle()`: Kernel-Stack passend “faken”

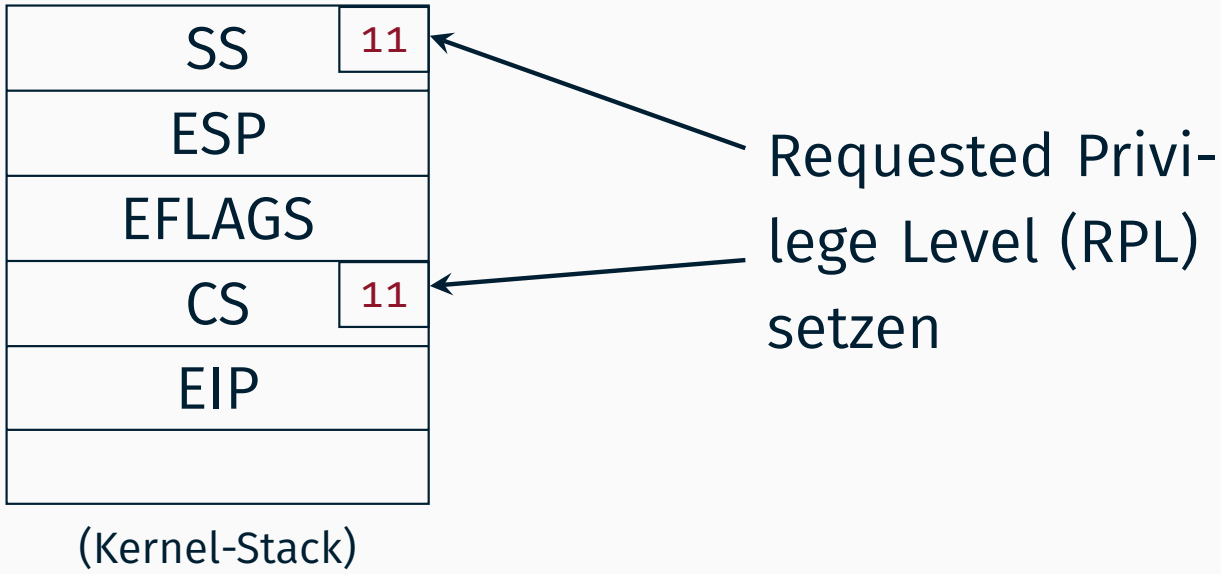
(Kernel-Stack)

Systemaufruf auf Kernel-Seite

Was fehlt noch?

➔ initialer Wechsel in Ring 3

Ähnlich zu `toc_settle()`: Kernel-Stack passend “faken”



(Kernel-Stack)

Wichtig: Segmentregister (`ds, es, fs, gs`) setzen

Danach: `iret`

Was ist (noch) nötig?

- Neue Segment-Deskriptoren anlegen (Usermode & TSS)
- Adresse des TSS $\rightarrow$ Deskriptor & `ltr <offset>`
- User-Stacks für Anwendungen einführen

  ($\rightarrow$ `toc_settle` weiterhin auf Kernel-Stack)

Was ist (noch) nötig?

- Neue Segment-Deskriptoren anlegen (Usermode & TSS)
- Adresse des TSS → Deskriptor & `ltr <offset>`
- User-Stacks für Anwendungen einführen
  (→ `toc_settle` weiterhin auf Kernel-Stack)

- Dispatcher: `tss.esp0` (um)setzen (`go` & `dispatch`)
- `kickoff` macht jetzt Wechsel in Ring 3
  ➔ `kickoff_user(Thread *obj)` mit `obj->action()`
  Woher kommt Parameter?

Was ist (noch) nötig?

- Neue Segment-Deskriptoren anlegen (Usermode & TSS)
- Adresse des TSS → Deskriptor & `ltr <offset>`
- User-Stacks für Anwendungen einführen
  (→ `toc_settle` weiterhin auf Kernel-Stack)

- Dispatcher: `tss.esp0` (um)setzen (`go` & `dispatch`)
- `kickoff` macht jetzt Wechsel in Ring 3
  ➔ `kickoff_user(Thread *obj)` mit `obj->action()`
  Woher kommt Parameter? ➔ User-Stack präparieren

Was ist (noch) nötig?

- Neue Segment-Deskriptoren anlegen (Usermode & TSS)
- Adresse des TSS → Deskriptor & `ltr <offset>`
- User-Stacks für Anwendungen einführen
  (→ `toc_settle` weiterhin auf Kernel-Stack)

- Dispatcher: `tss.esp0` (um)setzen (`go` & `dispatch`)
- `kickoff` macht jetzt Wechsel in Ring 3
  ➔ `kickoff_user(Thread *obj)` mit `obj->action()`
  Woher kommt Parameter? ➔ User-Stack präparieren

- **IDT** & `IRQ`-Makro in `startup.asm` anpassen
- Systemaufruf-Stümpfe und -Zuteiler schreiben

Schnellere Systemaufrufe (7.5 ECTS)

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, …)

Eigentlich für uns nur Ringwechsel nötig

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, …)

Eigentlich für uns nur Ringwechsel nötig

➔ `sysenter` & `sysexit`

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, …)

Eigentlich für uns nur Ringwechsel nötig

➔ **`sysenter`** & **`sysexit`**

`sysenter` sichert keinen Zustand (somit auch kein `eip`/`esp`!)

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, …)

Eigentlich für uns nur Ringwechsel nötig

➔ **`sysenter`** & **`sysexit`**

`sysenter` sichert keinen Zustand (somit auch kein `eip`/`esp`!)

`sysexit` erwartet aber `esp` in `ecx` und `eip` in `edx`

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, …)

Eigentlich für uns nur Ringwechsel nötig

➔ **`sysenter`** & **`sysexit`**

`sysenter` sichert keinen Zustand (somit auch kein `eip`/`esp`!)

`sysexit` erwartet aber `esp` in `ecx` und `eip` in `edx`

$\Rightarrow$ müssen vom Aufruf-Stumpf mitgegeben werden!

`sysenter`: Aufruf-Stumpf und Kern-Wrapper

```
sysenter_foo:
    mov edx, post_label
    mov ecx, esp
    mov eax, FOO_SYSNUM
    sysenter
post_label:
    ret
```

sysenter: Aufruf-Stumpf und Kern-Wrapper

```
%macro syscall 2
[GLOBAL %1]
%1:
    mov edx, %%post_label
    mov ecx, esp
    mov eax, %2
    sysenter
%%post_label:
    ret
%endmacro
syscall sysenter_foo, FOO_SYSNUM
```

`sysenter`: Aufruf-Stumpf und Kern-Wrapper

```
%macro syscall 2
[GLOBAL %1]
%1:
    mov edx, %%post_label
    mov ecx, esp
    mov eax, %2
    sysenter
%%post_label:
    ret
%endmacro
syscall sysenter_foo, FOO_SYSNUM
```

Kernel-Seite: eigener Wrapper, Parameter holen, C++, `sysexit`

`sysenter`: zu beachten

Woher kommen Kern-`cs/eip` und `-ss/esp`?

`sysenter`: zu beachten

Woher kommen Kern-`cs`/`eip` und `-ss`/`esp`?

➔ Model-specific registers (MSRs)

```
 cs 0x174
esp 0x175
eip 0x176
```

`sysenter`: zu beachten

Woher kommen Kern-`cs`/`eip` und -`ss`/`esp`?

➔ Model-specific registers (MSRs)

`cs` `0x174`
`esp` `0x175`
`eip` `0x176`
(`ss` [Wert in `0x174`] + 8)

`sysenter`: zu beachten

Woher kommen Kern-`cs`/`eip` und -`ss`/`esp`?

➔ Model-specific registers (MSRs)

`cs` `0x174`
`esp` `0x175`
`eip` `0x176`
(`ss` [Wert in `0x174`] + `8`)

Somit `esp` bei Threadwechsel schreiben, Rest nur initial

`sysenter`: zu beachten

Woher kommen Kern-`cs/eip` und `-ss/esp`?

➔ Model-specific registers (MSRs)

```
 cs 0x174
esp 0x175
eip 0x176
 (ss [Wert in 0x174] + 8)
```

Somit `esp` bei Threadwechsel schreiben, Rest nur initial

Woher kommen User-`cs/ss` bei `sysexit`?

`sysenter`: zu beachten

Woher kommen Kern-`cs/eip` und `-ss/esp`?

➔ Model-specific registers (MSRs)

`cs`	`0x174`
`esp`	`0x175`
`eip`	`0x176`
(`ss`	[Wert in `0x174`] + 8)

Somit `esp` bei Threadwechsel schreiben, Rest nur initial

Woher kommen User-`cs/ss` bei `sysexit`?

Code-Segment	Wert in MSR `0x174` + 16
Stack-/Daten-Segment	Wert in MSR `0x174` + 24

⇒ richtiges **GDT**-Layout wichtig!

Performance-Messungen

Auslesen mittels `rdtsc`-Instruktion

```
static inline uint64_t rdtsc() {
    uint32_t low, high;
    asm volatile (
        "rdtsc\n\t"
        : "=a" (low), "=d" (high)
        :: );
    return ((uint64_t)(high) << 32) | low;
}
```

oder einfach CLANG/GCC-Builtin:

```
uint64_t __builtin_ia32_rdtsc()
```

Zu beachten:

- eigentlich™ unsauber
  - wegen *out-of-order* Ausführung
  - Lösung mittels serialisierender Instruktion (`cpuid`) oder `rdtscp`
- Whitepaper “How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures”

$\Rightarrow$ für uns Builtin ausreichend

Fragen?