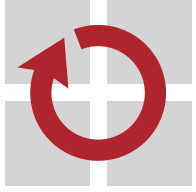
Übung zu Betriebssystemtechnik

Privilegienisolation und Systemaufrufe

23. April 2018

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
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
Vorlesungszeit
Syscalls
Paging
Message Passing

Zeitplan

April 2018

						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 2018

	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 2018

			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 2018

						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 Aquarium (0.031)

Rechnerübung
im Ruhe-CIP (00.153)

Abgabe der Aufgabe in der **Rechnerübung**

- Abgabe nur in festen **2er Gruppen**
- eine (obligatorische) Tafelübung pro Aufgabe
- Anmeldung zur Tafelübung (bis 24. April) im Waffel auf **waffel.informatik.uni-erlangen.de/signup?course=338**
- Informationen und Aufgabenstellung auf **www4.cs.fau.de/Lehre/SS18/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

Ziel der Übung

Auftrennung Usermodus $\Leftrightarrow$ Systemmodus

Unterschied?

Ziel der Übung

Auftrennung Usermodus $\Leftrightarrow$ Systemmodus

Unterschied?

- Erlaubte Befehle (`hlt`, `cli/sti`, `iret`, `inb/outb`, …)
- Zugriff auf Kontrollregister: `cr{0,2,3,4}`
  $\Rightarrow$ CPU-/MMU-Konfiguration

Ziel der Übung

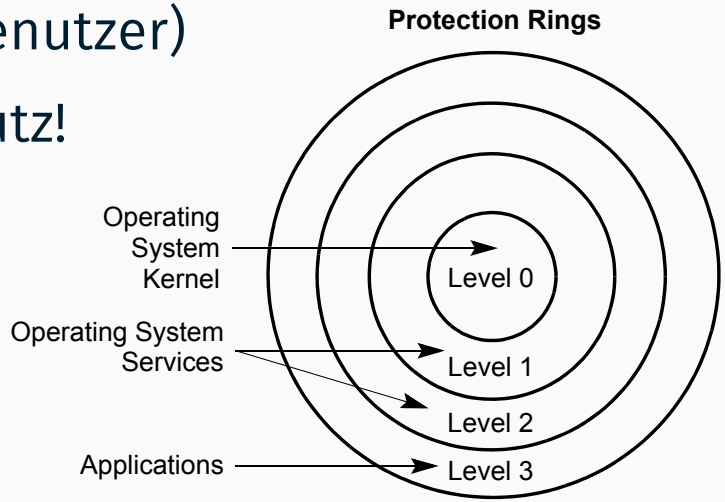
Auftrennung Usermodus $\Leftrightarrow$ Systemmodus

Unterschied?

- Erlaubte Befehle (`hlt, cli/sti, iret, inb/outb, …`)
- Zugriff auf Kontrollregister: `cr{0,2,3,4}`
  $\Rightarrow$ CPU-/MMU-Konfiguration

x86: **Ringmodell**

- Insgesamt 4 Ringe: 0 (System) – 3 (Benutzer)
- Beinhaltet noch keinen Speicherschutz!
- Auf Ring 3: Privilegierte Befehle und Register geschützt



Umsetzung

Was ist zu tun?

Schutzringe

- Konfiguration des Prozessors
- Wechsel zwischen den Modi

Was ist zu tun?

Schutzringe

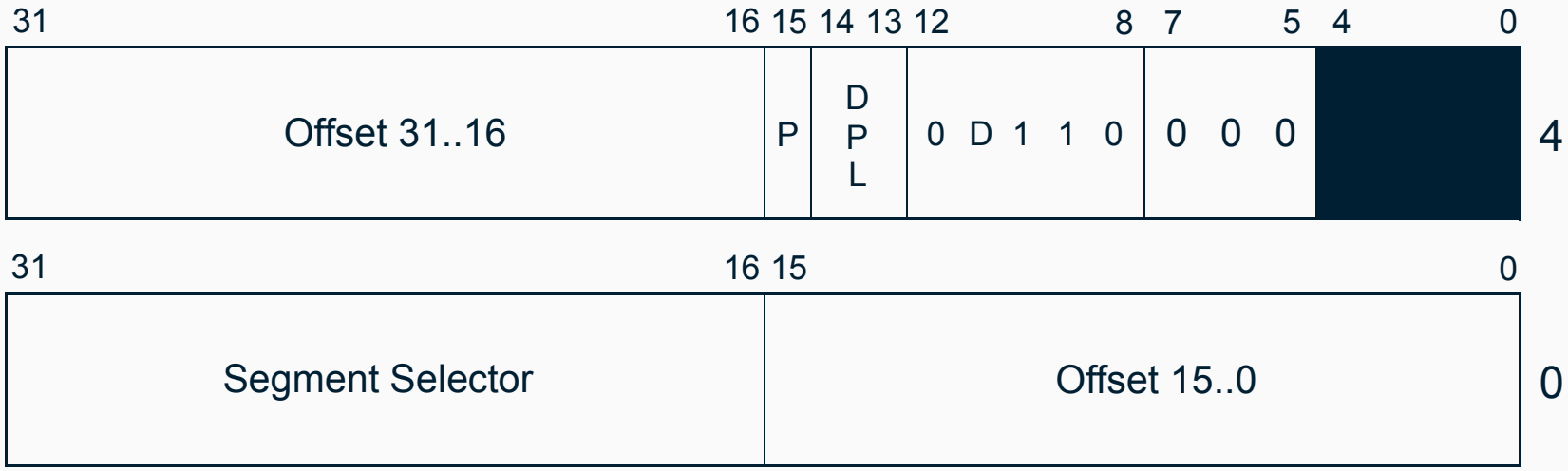
- Konfiguration des Prozessors
- Wechsel zwischen den Modi

Systemaufrufe

- Erlauben von Traps aus Ring 3
- Übergabe von Argumenten

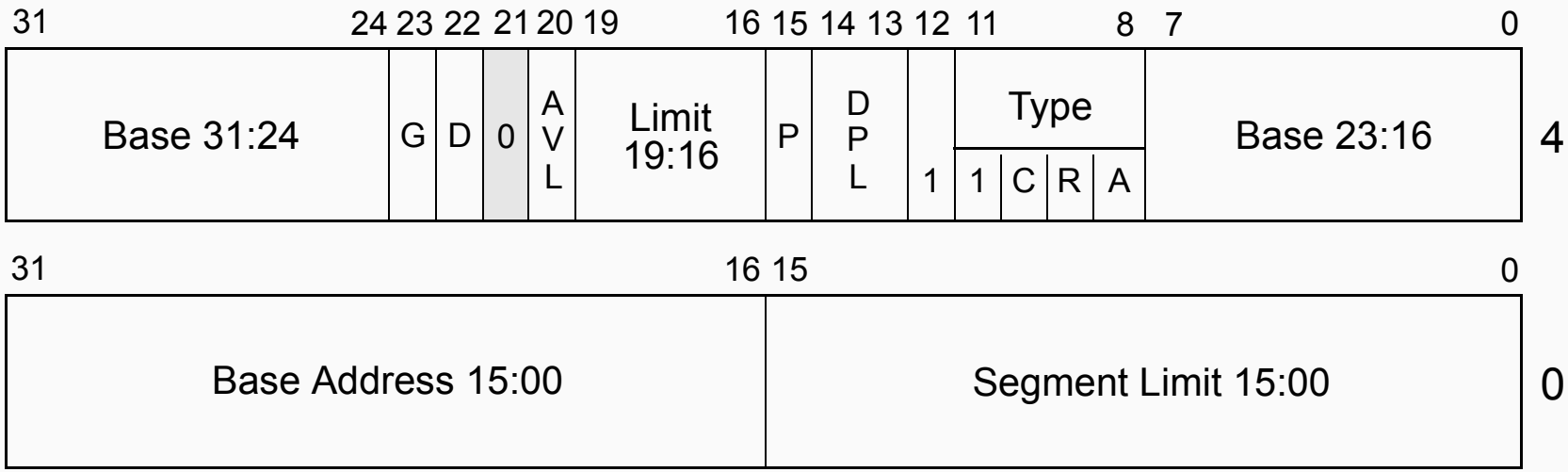
Interrupt Descriptor Table (IDT)

Deskriptoren für Interrupts



- Descriptor Privilege Level (DPL): Erlaubter Ring
- Offset: Angesprungene Funktion

Segment-Selektoren? ⇒ GDT



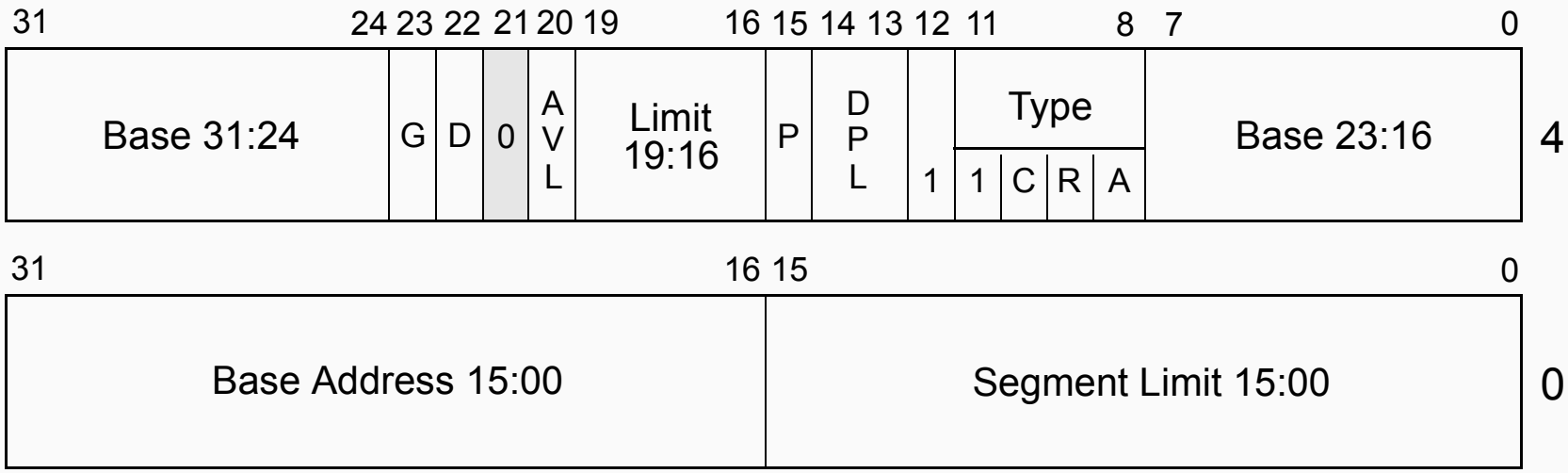
Segmentierung tief in x86 verwurzelt

Segmentregister (SS, CS, DS, ...) beinhalten Offset in GDT

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

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

Segment-Selektoren? ⇒ GDT



Segmentierung tief in x86 verwurzelt

Segmentregister (SS, CS, DS, …) beinhalten Offset in GDT

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

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

Was bedeutet `FF FF 00 00 | 00 9A CF 00`

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

- Instruction Pointer setzen
- Ring wechseln

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

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

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

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

- Zustand auf Stack sichern

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

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

- Zustand auf Stack sichern

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

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

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

- Zustand auf Stack sichern

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

```
mov esp, 0x0; int 0x42;
```

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

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

- Zustand auf Stack sichern

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

```
mov esp, 0x0; int 0x42;
```

Auslösen eines Systemaufrufs

Was passiert bei `int 0x42` aus Ring 3?

Hardware

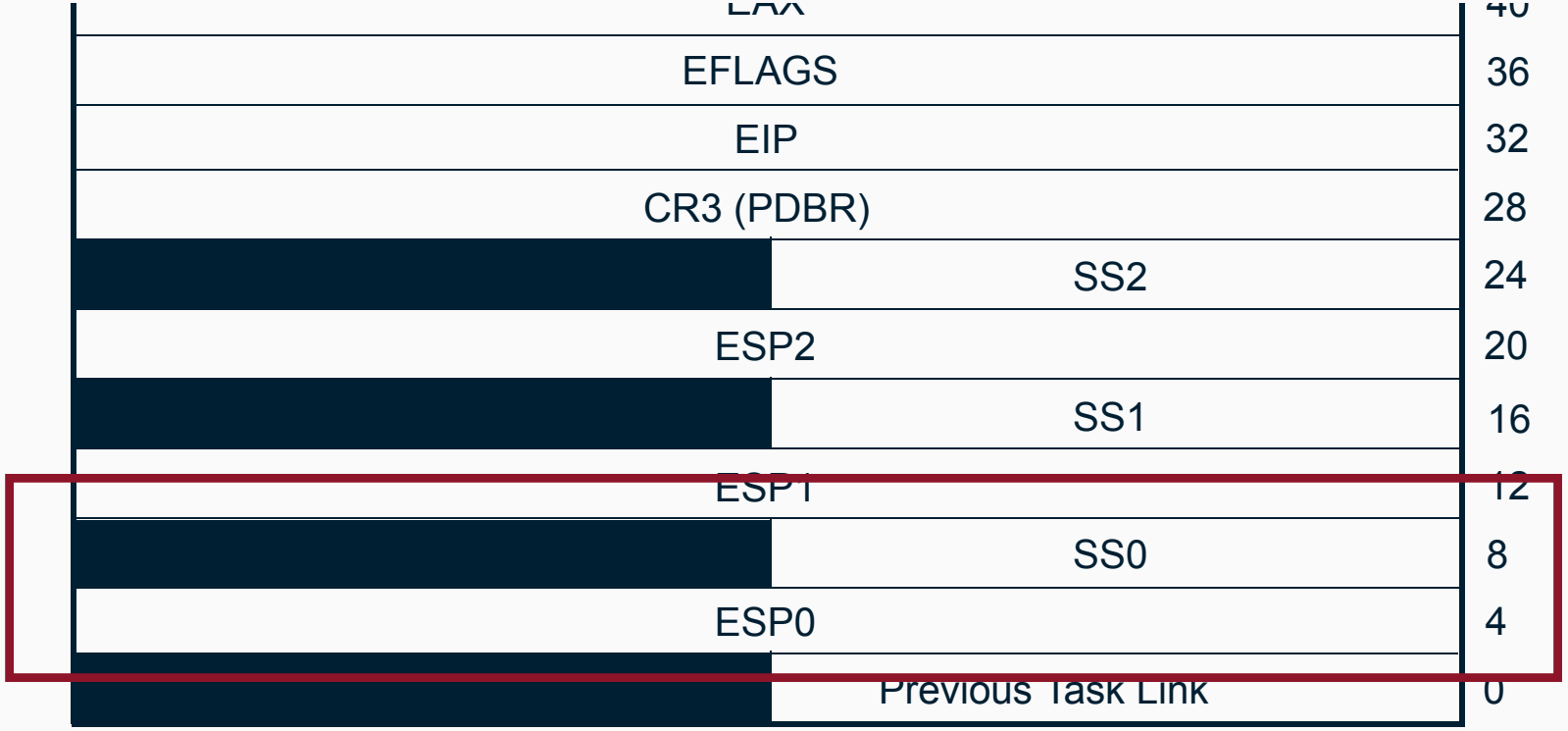
- Instruction Pointer setzen (aus IDT)
- Ring wechseln (aus IDT)
- **Auf Kernel-Stack wechseln**
- Zustand auf Stack sichern

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

```
mov esp, 0x0; int 0x42;
```

⇒ **Task State Segment**

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 Assembly 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
```

Systemaufruf (fast) in C

Alternative: 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

Erinnerung: Hardware wechselt Stack und Ring, setzt `eip`

Systemaufruf auf Kernel-Seite

Erinnerung: Hardware wechselt Stack und Ring, setzt `eip`

- Alle Argumentregister auf den (Kernel!) Stack legen
- System-Call-Handler (in C++) aufrufen
- `iret`

$\Rightarrow$ System-Call-Handler (in C++) ist `if-else`-Kaskade bzw. großes `switch/case`

Systemaufruf auf Kernel-Seite (cont.)

Was fehlt noch?

Systemaufruf auf Kernel-Seite (cont.)

Was fehlt noch?

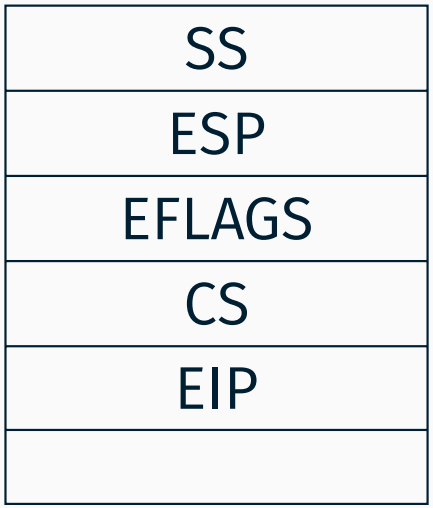
$\Rightarrow$ initialer Wechsel in Ring 3

Systemaufruf auf Kernel-Seite (cont.)

Was fehlt noch?

$\Rightarrow$ initialer Wechsel in Ring 3

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



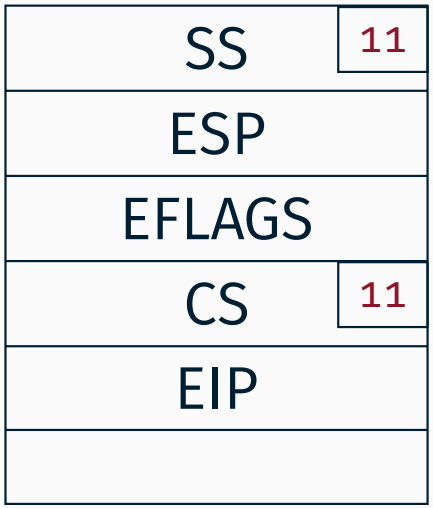
(Kernel-Stack)

Systemaufruf auf Kernel-Seite (cont.)

Was fehlt noch?

$\Rightarrow$ 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 $\rightarrow$ Deskriptor & `ltr <offset>`
- User-Stacks für Anwendungen einführen
  ($\rightarrow$ `toc_settle` weiterhin auf Kernel-Stack)

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

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)

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

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)

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

$\Rightarrow$ `sysenter & sysexit`

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, ...)

Eigentlich für uns nur Ringwechsel nötig

$\Rightarrow$ `sysenter & sysexit`

`sysenter` sichert keinen Zustand $\Rightarrow$ auch kein `eip`/`esp`!

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

Alternative zu Traps: `sysenter`

Traps sind langsam (Zustandssicherung, Segmentierung, ...)

Eigentlich für uns nur Ringwechsel nötig

$\Rightarrow$ `sysenter & sysexit`

`sysenter` sichert keinen Zustand $\Rightarrow$ auch kein `eip/esp`!

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

$\Rightarrow$ 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

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

```
%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

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

```
%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`

Evtl. Hardware-Kontext nachbauen ($\rightarrow$ später `fork()`)

`sysenter`: zu beachten

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

`sysenter`: zu beachten

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

$\Rightarrow$ Model-specific registers (MSRs)

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

`sysenter`: zu beachten

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

$\Rightarrow$ 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`?

$\Rightarrow$ Model-specific registers (MSRs)

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

$\Rightarrow$ `esp` bei Task-Wechsel schreiben, Rest nur initial

`sysenter`: zu beachten

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

$\Rightarrow$ Model-specific registers (MSRs)

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

$\Rightarrow$ `esp` bei Task-Wechsel 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)

⇒ `esp` bei Task-Wechsel 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

Intel bietet Zählerregister an, in jedem Takt inkrementiert

Auslesen mittels `rdtsc`-Instruktion

`clang/gcc`-Builtin: `uint64_t __builtin_ia32_rdtsc()`

Zu beachten:

- eigentlich™ unsauber
  - ⇒ Out-of-order Ausführung
  - ⇒ Serialisierende Instruktion (`cpuid`) und/oder `rdtscp`
- Whitepaper “How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures”

⇒ für uns Builtin ausreichend (oder `cpuid` nutzen)

Erinnerung: Entwicklungsumgebung

Makefile Targets

make qemu	QEMU **ohne** Hardware-Virtualisierung
make kvm	QEMU **mit** Hardware-Virtualisierung
make qemu-gdb	starte in QEMU und verbinde zu integrierten GDB-Stub $\rightarrow$ Fehlersuche mit gdb.
make netboot	für Boot am Test-Rechner ins NFS kopieren

Suffix `-noopt` um Optimierungen auszuschalten (sonst `-O3`)

Fragen?