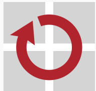
Übung zu Betriebssystemtechnik

Aufgabe 1: Privilegientrennung

26. April 2022

Bernhard Heinloth, Phillip Raffeck & Dustin Nguyen

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



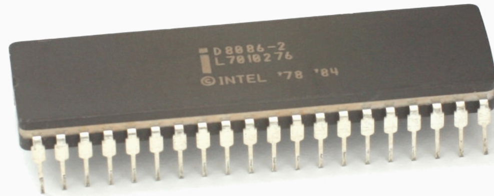
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Ziel der 1. Aufgabe: Anwendungen sollen in einem unpriviligierten Modus laufen

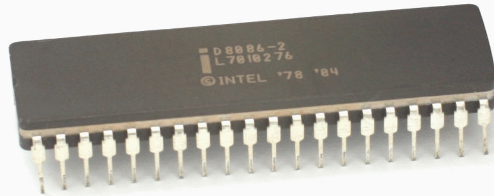
Quelle: Wikipedia

Real Mode



Quelle: Wikipedia

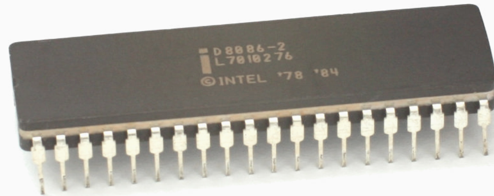
Real Mode



Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

Real Mode

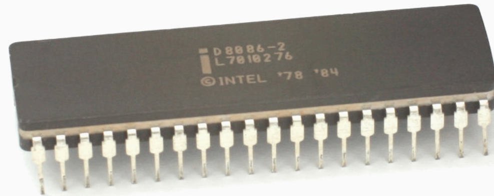


Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

- Segementierung mit 20 bit Adressbus

Real Mode



Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

- Segementierung mit 20 bit Adressbus
- 14 Register

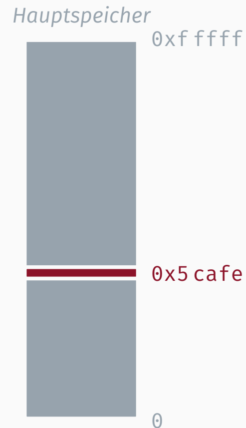
Real Mode Register

Real Mode Register

Real Mode Segmentierung

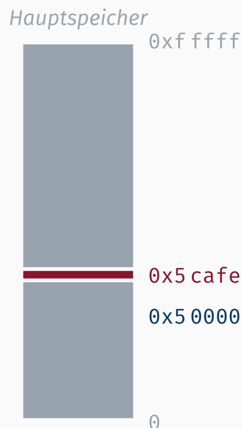
Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5 cafe`



Zugriff auf Zieldaten in `0x5cafe`

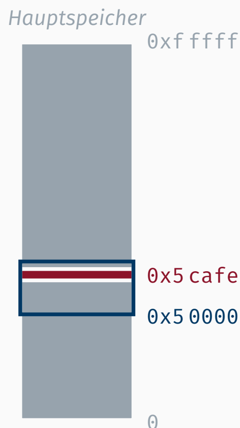
1. Setze **ds** auf `0x5000`



Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze **ds** auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)

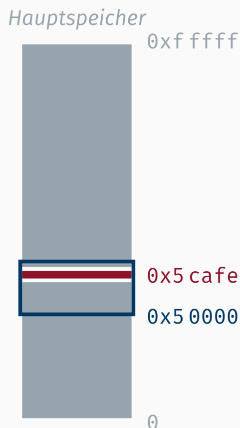


Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5 cafe`

1. Setze **`ds`** auf `0x5000`
   (Bereich `0x5 0000` – `0x5 ffff`)

2. Zugriff über **`ds:0xcafe`**

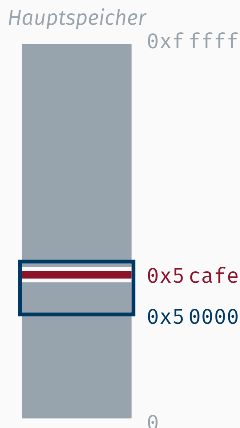


Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5 cafe`

1. Setze **`ds`** auf `0x5000`
   (Bereich `0x5 0000` – `0x5 ffff`)

2. Zugriff über **`ds:0xcafe`**
   (**`ds`** $\times$ `0x10` $+$ **`0xcafe`** = `0x5 cafe`)



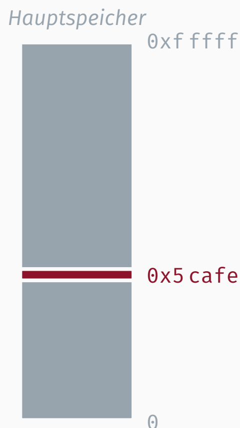
Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze **ds** auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)

2. Zugriff über **`ds:0xcafe`**
   (**ds** × `0x10` + **`0xcafe`** = `0x5cafe`)

Alternativ:



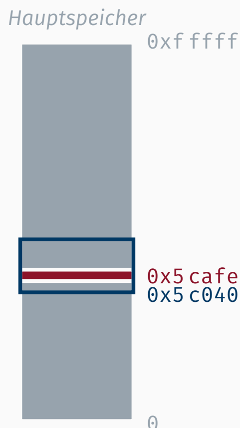
Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

1. Setze **ds** auf `0x5000`
   (Bereich `0x50000` – `0x5ffff`)
2. Zugriff über **`ds:0xcafe`**
   (**ds** × `0x10` + **`0xcafe`** = `0x5cafe`)

Alternativ:

1. Setze **ds** auf `0x5c04`
   (Bereich `0x5c040` – `0x6c039`)



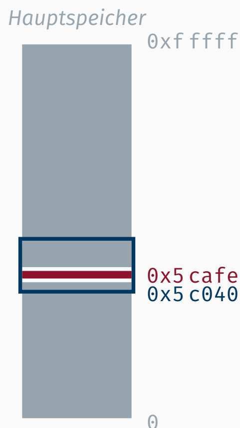
Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5 cafe`

1. Setze **ds** auf `0x5000`
   (Bereich `0x5 0000` – `0x5 ffff`)
2. Zugriff über **`ds:0xcafe`**
   (**ds** × `0x10` + **`0xcafe`** = `0x5 cafe`)

Alternativ:

1. Setze **ds** auf `0x5c04`
   (Bereich `0x5 c040` – `0x6 c039`)
2. Zugriff über **`ds:0xabe`**



Protected Mode

Quelle: Wikipedia

Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

Protected Mode

Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

- Segementierung mit 24 bit Adressbus

Protected Mode

Quelle: *Wikipedia*

Quelle: *Wikipedia*

80286 16 bit, 16 MB Speicher adressierbar

- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

Protected Mode

Quelle: Wikipedia

Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus

Protected Mode

Quelle: Wikipedia

Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Protected Mode

Name aufgrund der Schutzringe
(*Privilege Level*)

80286 16 bit, 16 MB Speicher adressierbar

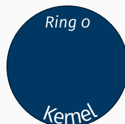
- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Protected Mode

Name aufgrund der Schutzringe (*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar

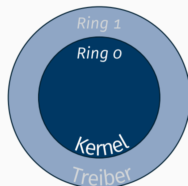
- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Protected Mode

Name aufgrund der Schutzringe
(*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar

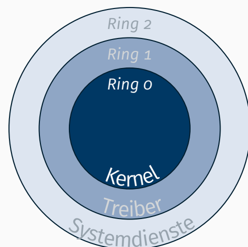
- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Protected Mode

Name aufgrund der Schutzringe (*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar

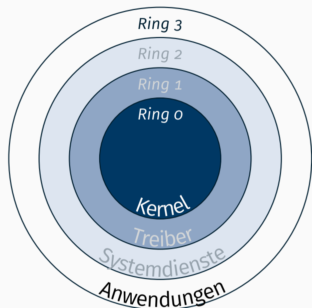
- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Protected Mode

Name aufgrund der Schutzringe
(*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar

- Segementierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segementierung mit 32 bit Adressbus
- *zusätzlich auch Paging (ab 80386DX)*

Segmentierung im Protected Mode

- Einführung von Deskriptortabellen

  GDT Global Descriptor Table

  LDT Local Descriptor Table

  IDT Interrupt Descriptor Table

Segmentierung im Protected Mode

- Einführung von Deskriptortabellen

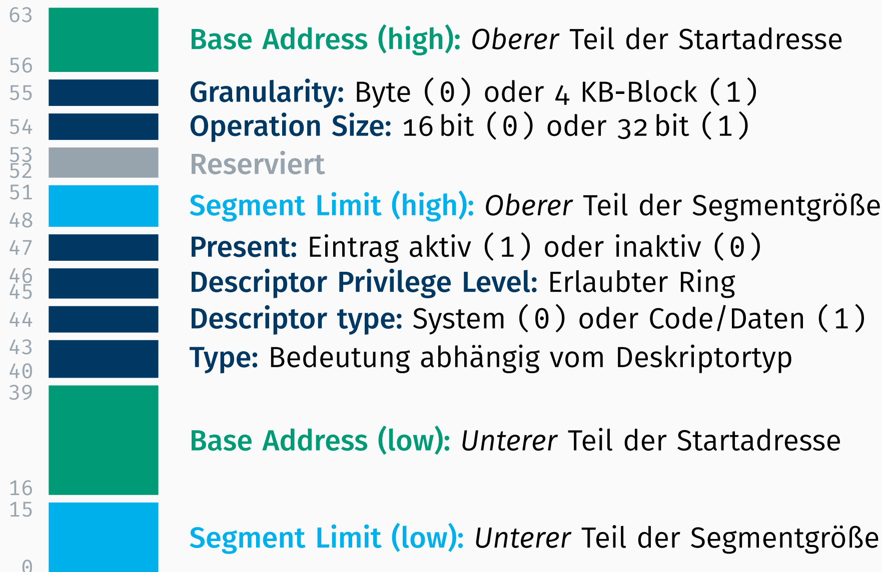
  GDT Global Descriptor Table
  LDT Local Descriptor Table
  IDT Interrupt Descriptor Table

- jeder der Deskriptoren (max. 8 192 pro Tabelle) besteht aus
  - Basisadresse (24 bit bzw. 32 bit)
  - Länge (16 bit bzw. 20 bit)
  - Parameter wie Typ, Berechtigung, Aktiv, ...

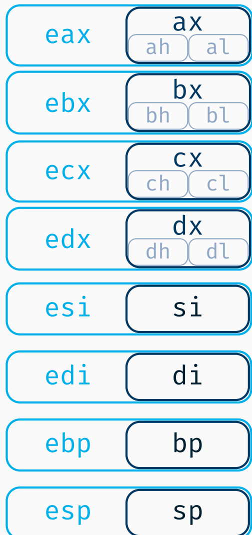
→ Segmentselektoren zeigen nun auf Einträge in GDT/LDT

16 bit Segmentdeskriptoreintrag

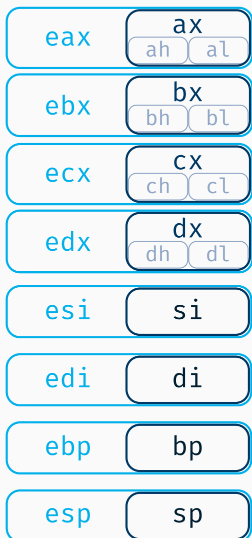
32 bit Segmentdeskriptoreintrag



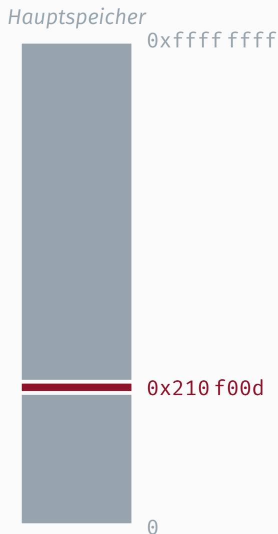
32 bit Protected Mode Register



32 bit Protected Mode Register



Zugriff auf Zieldaten in `0x210 f00d`



Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

- über **GDT**

Hauptspeicher

0xffff ffff

0x210 f00d

0

Global Deskriptor Table

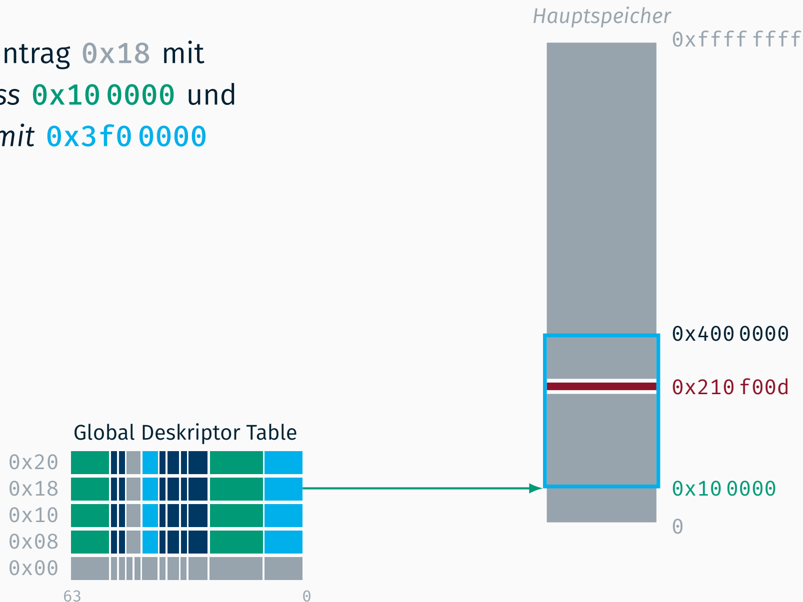
0x20
0x18
0x10
0x08
0x00

63 0

Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

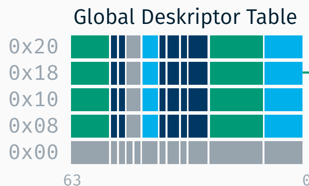
- über **GDT** Eintrag `0x18` mit *Base Address* `0x10 0000` und *Segment Limit* `0x3f0 0000`

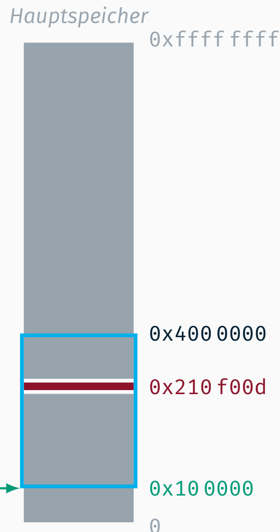


Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

- über **GDT** Eintrag `0x18` mit *Base Address* `0x10 0000` und *Segment Limit* `0x3f0 0000`
- mit Segmentregister `ds`

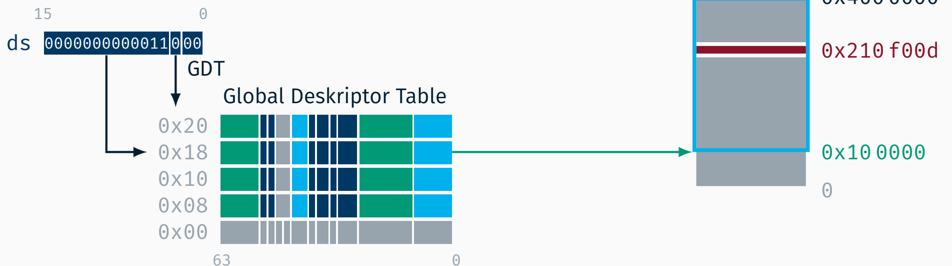




Zugriff auf Zieldaten in `0x210 f00d`

- über **GDT** Eintrag `0x18` mit
  Base Address `0x10 0000` und
  Segment Limit `0x3f0 0000`
- mit Segmentregister **`ds`** mit

```
mov ax, 0x18
mov ds, ax
```



Syntaxunterschiede bei x86-Assembler

Intel:

`mov ax, 0x18`

(Ziel, Quelle)

`objdump -M intel`

Standard bei `nasm` (Netwide Assembler)

AT&T:

`mov $0x18, %ax`

(Quelle, Ziel)

Standard bei `objdump`

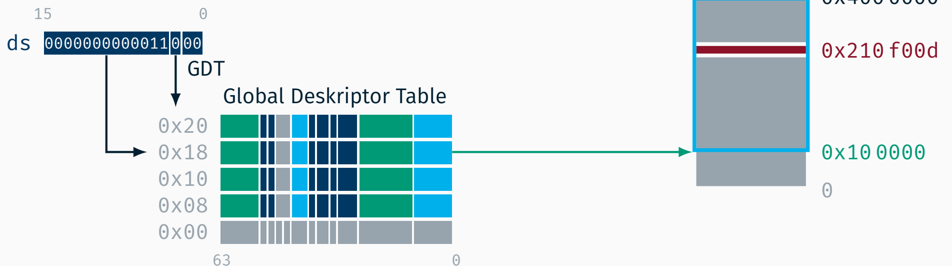
und GCC `inline asm`

Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

- über **GDT** Eintrag `0x18` mit
  Base Address `0x10 0000` und
  Segment Limit `0x3f0 0000`

- mit Segmentregister **`ds`** mit
  ```
  mov ax, 0x18
  mov ds, ax
  ```

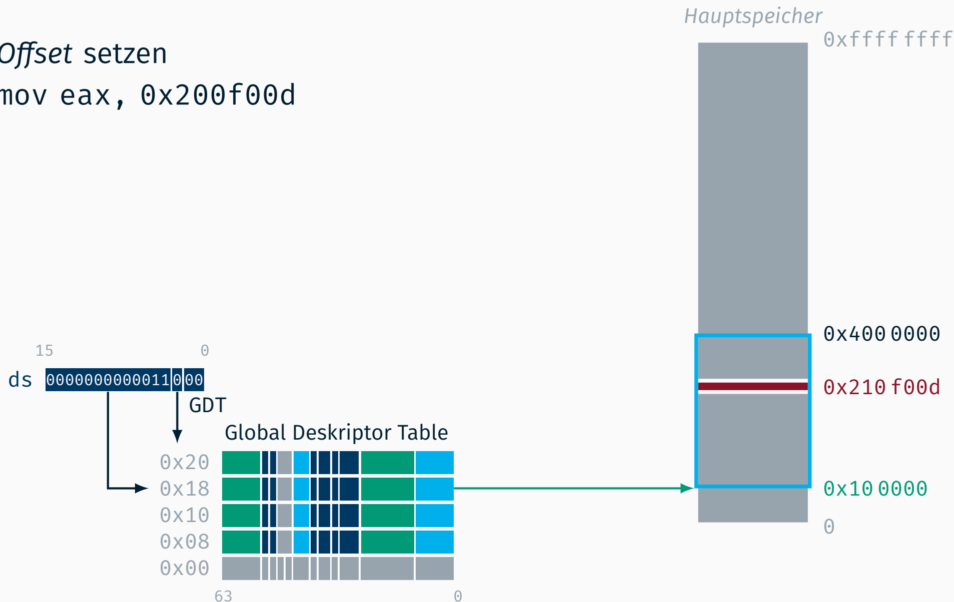


Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

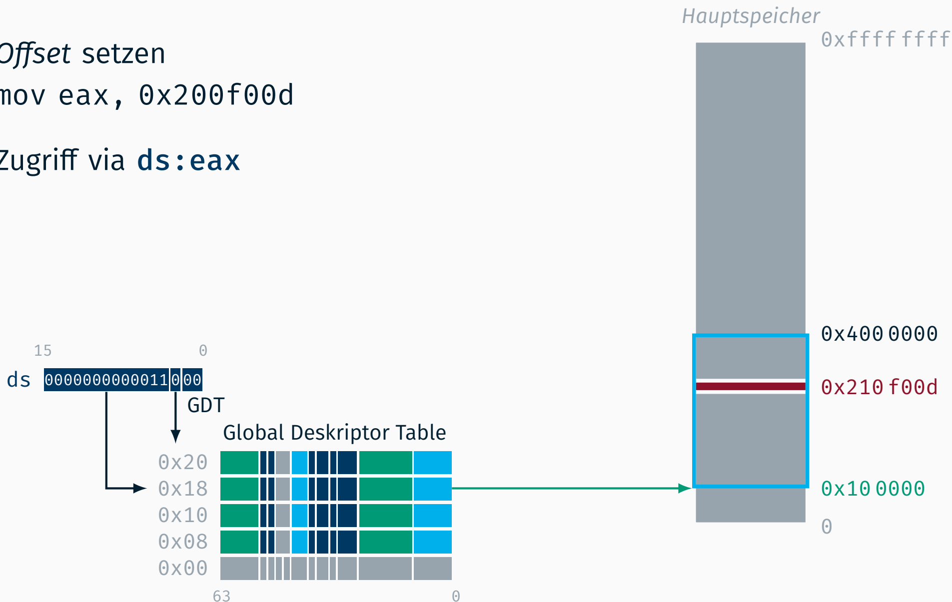
- *Offset* setzen

```
mov eax, 0x200f00d
```



Zugriff auf Zieldaten in `0x210 f00d`

- *Offset* setzen
  `mov eax, 0x200f00d`
- Zugriff via `ds:eax`



Protected Mode Segmentierung (32 bit)

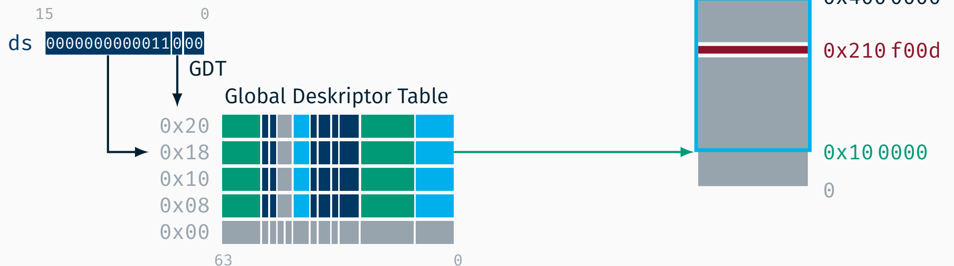
Zugriff auf Zieldaten in `0x210 f00d`

- *Offset* setzen
  `mov eax, 0x200f00d`
- Zugriff via `ds:eax`
  (*Base Address* + *Offset* = Zieldaten)

Protected Mode Segmentierung (32 bit)

Zugriff auf Zieldaten in `0x210 f00d`

- *Offset* setzen
  `mov eax, 0x200f00d`
- Zugriff via `ds:eax`
  (`0x10 0000` + `0x200 f00d` = `0x210 f00d`)



Protected Mode Segmentierung (32 bit) umgehen

Flaches Speichermodell

→ Zugriff über `0x210 f00d`

Hauptspeicher

0xffff ffff

0x210 f00d

0

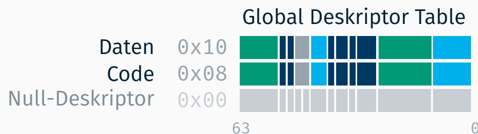
Global Deskriptor Table

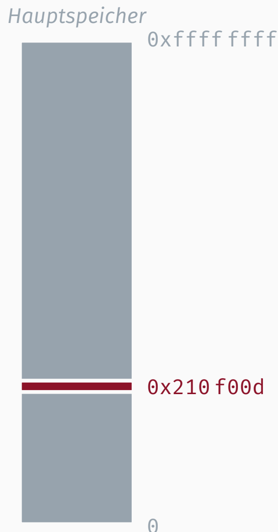
0x20
0x18
0x10
0x08
0x00

63 0

Flaches Speichermodell

→ Zugriff über `0x210 f00d`





Protected Mode Segmentierung (32 bit) umgehen

ISDMv3 3.2.1

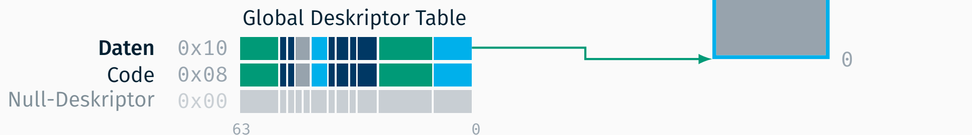
Flaches Speichermodell

```
mov ax, 0x10
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
mov ss, ax
```

Hauptspeicher

0xffff ffff

0x210 f00d



Protected Mode Segmentierung (32 bit) umgehen

Flaches Speichermodell

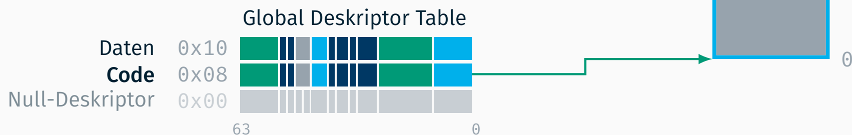
```
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    jmp 0x8:load_cs

load_cs:
    ...
```

Hauptspeicher

0xffff ffff

0x210 f00d

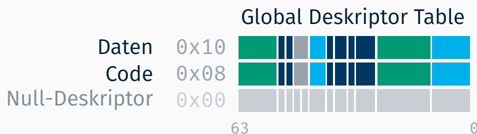
0

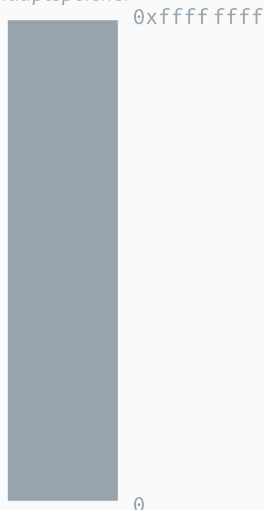


Protected Mode Segmentierung (32 bit) umgehen

Flaches Speichermodell → `startup.asm`

```
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    jmp 0x8:load_cs

load_cs:
    ...
```

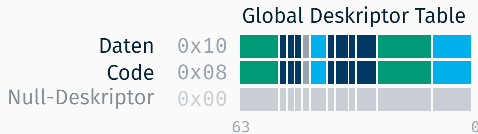




64 bit GDT

Flaches Speichermodell

Quelle: Wikipedia

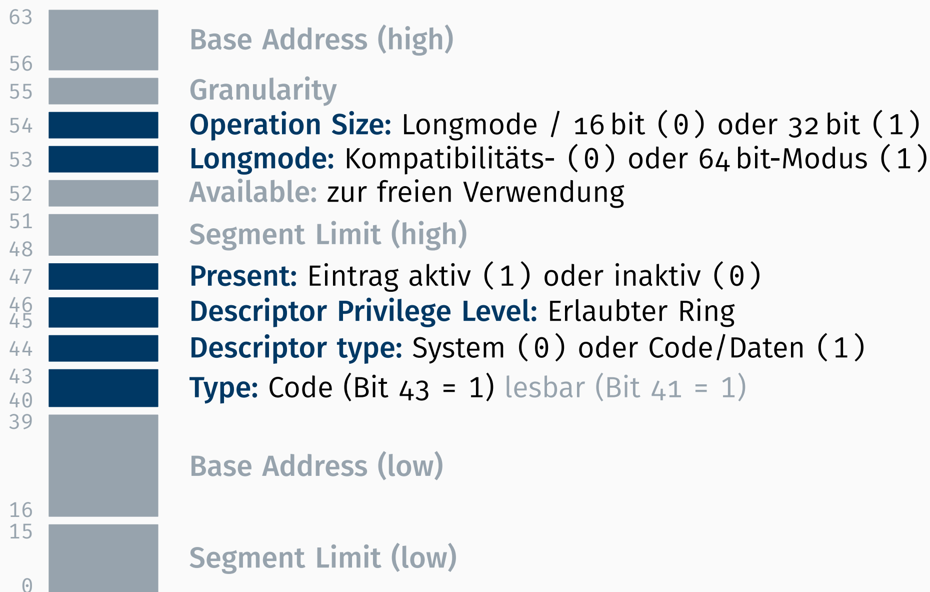


64 bit Segmentdeskriptoreintrag

ISDMv3 3.4.5 & AAPMv2 4.8

64 bit GDT Codesegmentdeskriptoreintrag

ISDMv3 3.4.5.1 & AAPMv2 4.8.1

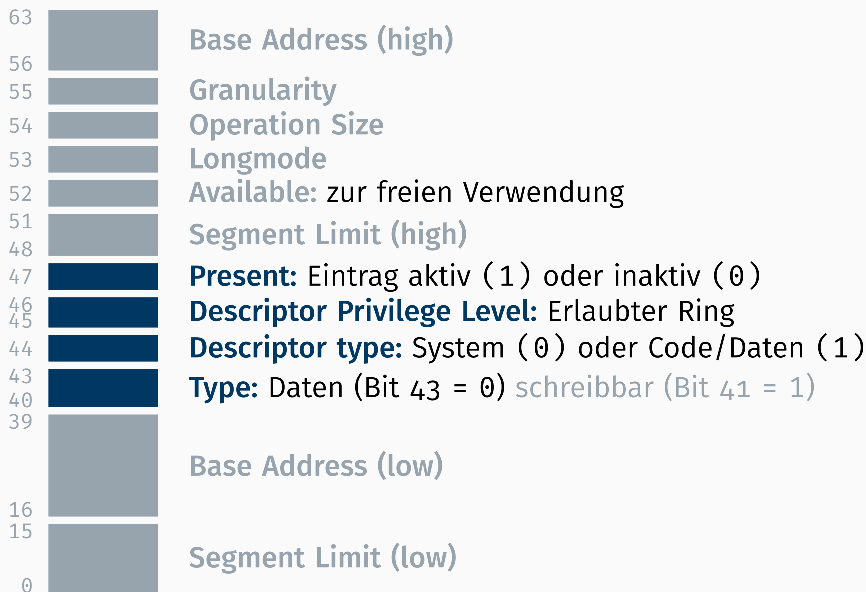


64 bit GDT Codesegmentdeskriptoreintrag

ISDMv3 3.4.5.1 & AAPMv2 4.8.1

Bits	Wert	Feld
63–56	0x0	Base Address (high)
55	0x0	Granularity
54	0x0	**Operation Size:** Longmode / 16 bit (0) oder 32 bit (1)
53	0x1	**Longmode:** Kompatibilitäts- (0) oder 64 bit-Modus (1)
52	0x0	Available: zur freien Verwendung
51–48	0x0	Segment Limit (high)
47	0x1	**Present:** Eintrag aktiv (1) oder inaktiv (0)
46–45	0x0	**Descriptor Privilege Level:** Erlaubter Ring
44	0x0	**Descriptor type:** System (0) oder Code/Daten (1)
43–40	0xA	**Type:** Code (Bit 43 = 1) lesbar (Bit 41 = 1)
39–16	0x0	Base Address (low)
15–0	0x0	Segment Limit (low)

64 bit GDT Datensegmentdeskriptoreintrag ISDMv3 3.4.5.1 & AAPMv2 4.8.2

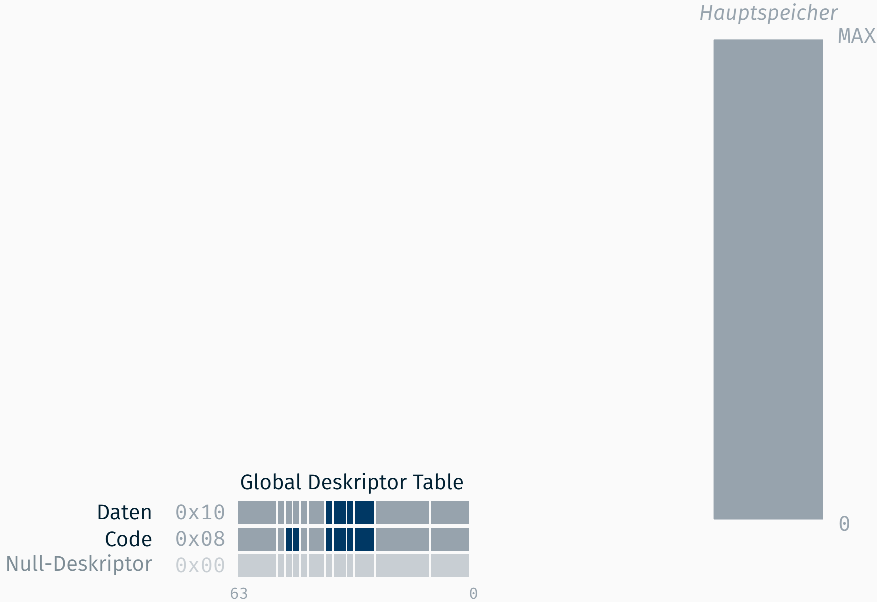


64 bit GDT Datensegmentdeskriptoreintrag

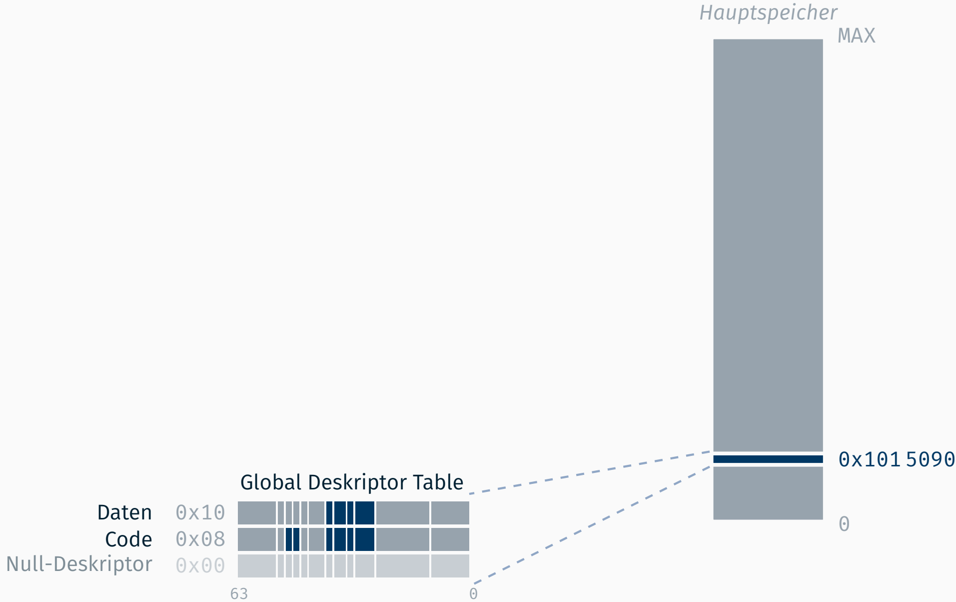
ISDMv3 3.4.5.1 & AAPMv2 4.8.2

Bits	Wert	Feld
63–56	0x0	Base Address (high)
55	0x0	Granularity
54	0x0	Operation Size
53	0x1	Longmode
52	0x0	Available: zur freien Verwendung
51–48	0x0	Segment Limit (high)
47	0x1	**Present:** Eintrag aktiv (1) oder inaktiv (0)
46–45	0x0	**Descriptor Privilege Level:** Erlaubter Ring
44	0x0	**Descriptor type:** System (0) oder Code/Daten (1)
43–40	0x2	**Type:** Daten (Bit 43 = 0) schreibbar (Bit 41 = 1)
39–16	0x0	Base Address (low)
15–0	0x0	Segment Limit (low)

64 bit GDT (Flaches Speichermodell)

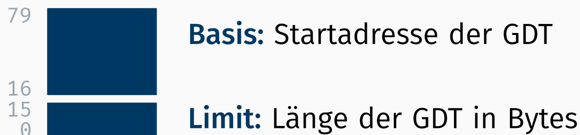


64 bit GDT (Flaches Speichermodell)



64 bit GDT (Flaches Speichermodell)

GDT-Register `gdtr`



Hauptspeicher

MAX

`gdtr`

0

Global Deskriptor Table

Daten	0x10	
Code	0x08	
Null-Deskriptor	0x00	

63 0

64 bit GDT (Flaches Speichermodell)

GDT-Register **gdtr**

Bits	Wert
79–16	0x101 5090
15–0	0x17

Basis: Startadresse der GDT

(bei uns `GDT::long_mode` aus `machine/gdt.cc`)

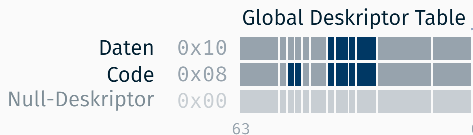
Limit: Länge der GDT in Bytes

Hauptspeicher

MAX

gdtr

0



64 bit GDT (Flaches Speichermodell)

GDT-Register **`gdtr`**

Bits	Wert
79–16	0x101 5090
15–0	0x17

Basis: Startadresse der GDT
(bei uns `GDT::long_mode` aus `machine/gdt.cc`)

Limit: Länge der GDT in Bytes

Instruktionen:

`lgdt` in Register laden

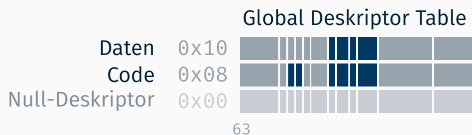
`sgdt` aus Register lesen

Hauptspeicher

MAX

`gdtr` 0x101 5090

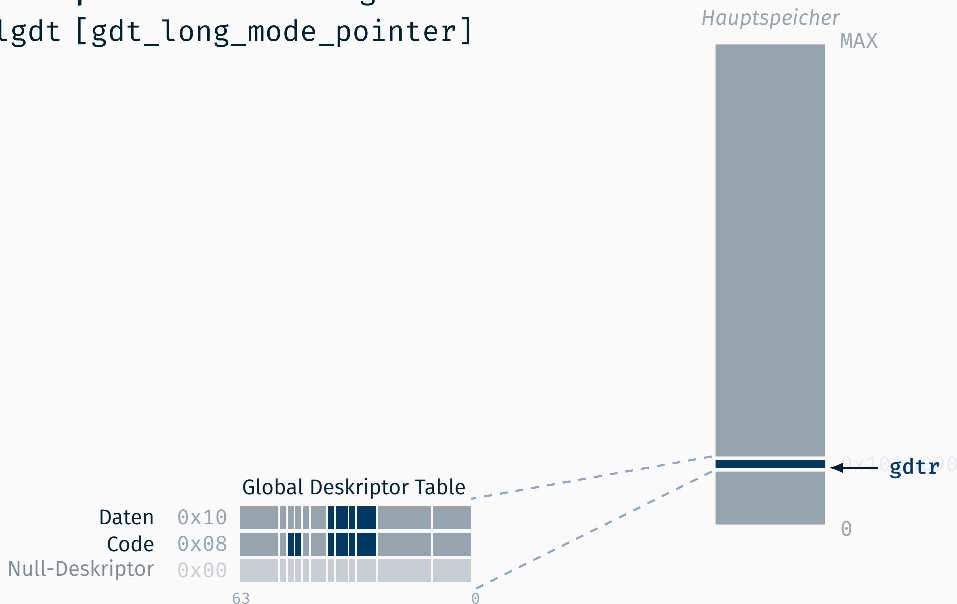
0



64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → `longmode.asm`:

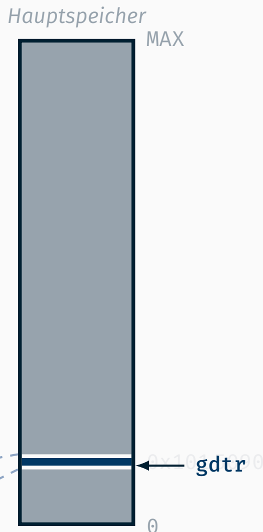
```
lgdt [gdt_long_mode_pointer]
```

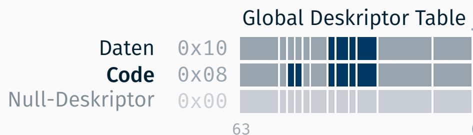


64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → `longmode.asm`:

```
    lgdt [gdt_long_mode_pointer]
    jmp 0x8:long_mode_start

long_mode_start:
```

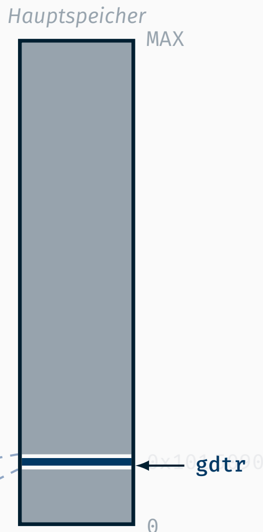


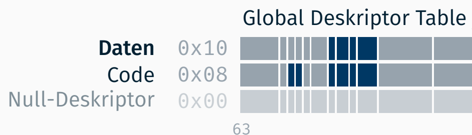


64 bit GDT (Flaches Speichermodell)

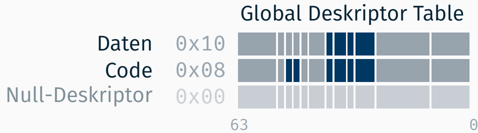
Flaches Speichermodell → `longmode.asm`:

```
   lgdt [gdt_long_mode_pointer]
   jmp 0x8:long_mode_start

long_mode_start:
   mov ax, 0x10  ; oder 0x0
   mov ss, ax
   mov ds, ax
   mov es, ax
   mov fs, ax
   mov gs, ax
```

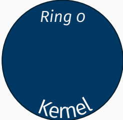


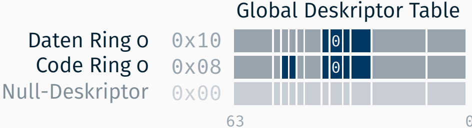


64 bit GDT



64 bit GDT (Privilegienebenen)





64 bit GDT (Privilegienebenen)

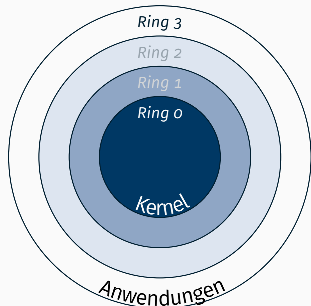
ISDMv3 5.9

Ausführung von **priviligierten Befehlen** wie

- Laden von GDT/LDT/IDT (`lgdt`/`lldt`/`lidt`) und Taskregister (`ltr`)
- Ändern (`mov`) von Kontroll- (`cr0-3`) und Debugregister (`dr0-7`)
- Konfigurieren von Model-Specific Register (`rdmsr`/`writemsr`)
- Invalidieren von Cache (`invd`/`wbinvd`) und TLB (`invlpg`)
- Sperren/Erlauben von Unterbrechungen (`cli`/`sti`)
- Zugriff auf I/O Ports (`in`/`out`)
- Stoppen des Kerns (`hlt`)

in Ring 3 verboten (führt zu **General Protection Fault**)

(bei Unterbrechungen und I/O Port jedoch abhängig von I/O Privilege Level)



Global Deskriptor Table

Deskriptor	Offset	Privilegienebene
Daten Ring 3	`0x20`	3
Code Ring 3	`0x18`	3
Daten Ring 0	`0x10`	0
Code Ring 0	`0x08`	0
Null-Deskriptor	`0x00`	

63 … 0

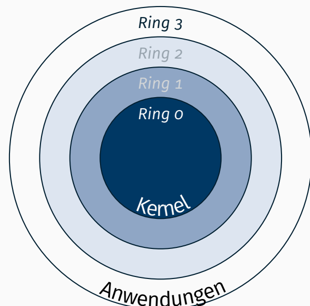
64 bit GDT (Privilegienebenen und Unterbrechungen)

Ausführung von **priviligierten Befehlen** wie

- Laden von GDT/LDT/IDT (`lgdt`/`lldt`/`lidt`) und Taskregister (`ltr`)
- Ändern (`mov`) von Kontroll- (`cr0-3`) und Debugregister (`dr0-7`)
- Konfigurieren von Model-Specific Register (`rdmsr`/`writemsr`)
- Invalidieren von Cache (`invd`/`wbinvd`) und TLB (`invlpg`)
- Sperren/Erlauben von Unterbrechungen (`cli`/`sti`)
- Zugriff auf I/O Ports (`in`/`out`)
- Stoppen des Kerns (`hlt`)

in Ring 3 verboten (führt zu **General Protection Fault**)

(bei Unterbrechungen und I/O Port jedoch abhängig von I/O Privilege Level)



Global Deskriptor Table

Daten Ring 3	0x20		3	
Code Ring 3	0x18		3	
Daten Ring 0	0x10		0	
Code Ring 0	0x08		0	
Null-Deskriptor	0x00			

63 … 0

64 bit GDT (Privilegienebenen und Unterbrechungen)

Ausführung von **priviligierten Befehlen** wie

- Laden von GDT/LDT/IDT (`lgdt/lldt/lidt`) und Taskregister (`ltr`)
- Ändern (`mov`) von Kontroll- (`cr0-3`) und Debugregister (`dr0-7`)
- Konfigurieren von Model-Specific Register (`rdmsr/writemsr`)
- Invalidieren von Cache (`invd/wbinvd`) und TLB (`invlpg`)
- Sperren/Erlauben von Unterbrechungen (`cli/sti`)
- Zugriff auf I/O Ports (`in/out`)
- Stoppen des Kerns (`hlt`)

in Ring 3 verboten (führt zu **General Protection Fault**)

(bei Unterbrechungen und I/O Port jedoch abhängig von I/O Privilege Level)

64 bit GDT (Privilegienebenen und Unterbrechungen)

Kernelstack	...	*bei Unterbrechung in Ring 0*
rsp + 32	ss	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	cs	vorheriges Codesegment
rsp →	rip	Rücksprungadresse

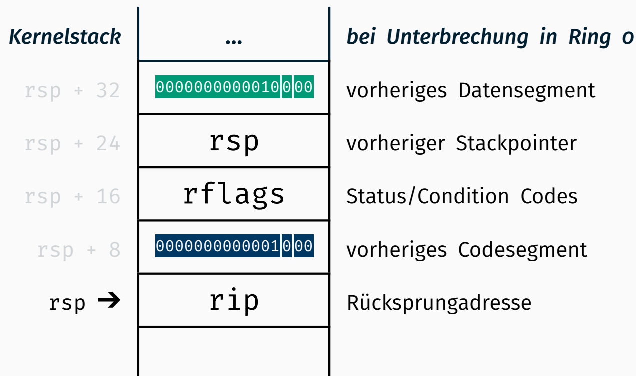
Ring 3
Ring 2
Ring 0
Kernel
Anwendungen

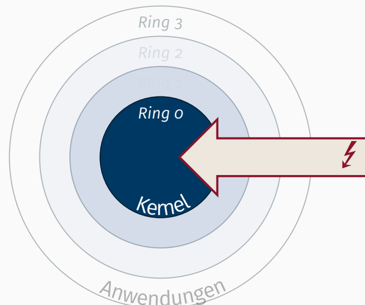
Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

64 bit GDT (Privilegienebenen und Unterbrechungen)





Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 0

64 bit GDT (Privilegienebenen und Unterbrechungen)

Kernelstack	...	*bei Unterbrechung in Ring 0*
rsp + 32	0000000000010 0 00	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	0000000000001 0 00	vorheriges Codesegment
rsp →	rip	Rücksprungadresse

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

64 bit GDT (Privilegienebenen und Unterbrechungen)

Kernelstack	...	*bei Unterbrechung in Ring 3*
rsp + 32	ss	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	cs	vorheriges Codesegment
rsp →	rip	Rücksprungadresse

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

64 bit GDT (Privilegienebenen und Unterbrechungen)

Kernelstack	...	*bei Unterbrechung in Ring 3*
rsp + 32	0000000000100 0 11	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	0000000000011 0 11	vorheriges Codesegment
rsp →	rip	Rücksprungadresse

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 0

64 bit GDT (Privilegienebenen und Unterbrechungen)

Kernelstack	...	*bei Unterbrechung in Ring 3*
rsp + 32	0000000000100 0 11	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	0000000000011 0 11	vorheriges Codesegment
rsp →	rip	Rücksprungadresse

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

64 bit GDT (Privilegienebenen wechseln)

Initialer Wechsel in Ring 3 durch präparierten Kernelstack und `iretq`

rsp + 32	0000000000100 0 11	Zieldatensegment
rsp + 24	rsp	Top-of-Userstack
rsp + 16	rflags	Status
rsp + 8	0000000000011 0 11	Zielcodesegment
rsp →	rip	Zielfunktion

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Global Deskriptor Table

Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 0

64 bit GDT (Privilegienebenen wechseln)

Initialer Wechsel in Ring 3 durch präparierten Kernelstack und `iretq`

Adresse	Stackinhalt	Bedeutung
rsp + 32	0000000000100 0 11	Zieldatensegment
rsp + 24	rsp	Top-of-Userstack
rsp + 16	rflags	Status (Unterbrechungen aktiv)
rsp + 8	0000000000011 0 11	Zielcodesegment
rsp →	rip	Zielfunktion

Global Deskriptor Table

Deskriptor	Offset	DPL
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

64 bit GDT (Privilegienebenen wechseln)

Initialer Wechsel in Ring 3 durch präparierten Kernelstack und `iretq`

Adresse	Stack	Beschreibung
rsp + 32	0000000000100 0 11	Zieldatensegment
rsp + 24	rsp	Top-of-**Userstack**
rsp + 16	rflags	Status (Unterbrechungen aktiv)
rsp + 8	0000000000011 0 11	Zielcodesegment
rsp →	rip	Zielfunktion

Global Deskriptor Table

Eintrag	Offset	DPL
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Anwendungen

Den aktuellen Ring finden

And some things that should not have been forgotten were lost.
[...] the ring passed out of all knowledge.

(Galadriel)

Den aktuellen Ring finden

And some things that should not have been forgotten were lost.
[...] the ring passed out of all knowledge.

(Galadriel)

```
inline unsigned current_ring() {
    // read code segment register
    unsigned cs;
    asm volatile ("mov %%cs, %0\n\t" : "=r"(cs));
    return cs & 0b11;  // last bits define the ring
}
```

Wenn im Userspace (Ring 3) ein Interrupt kommt: Woher bekommen wir (wieder) den Kernelstack?

Task State Segment

Task State Segment (16 bit)

- Ab 286er (16 bit Protected Mode):
  Hardware Tasks

Task State Segment (16 bit)

- Ab 286er (16 bit Protected Mode):
  Hardware Tasks
- Der Zustand wird beim Wechsel im *Task State Segment (TSS)* gesichert

LDT
DS
SS
CS
ES
DI
SI
BP
SP
BX
DX
CX
AX
FLAG
IP
SS2
SP2
SS1
SP1
SS0
SP0
Link

Task State Segment (16 bit)

- Ab 286er (16 bit Protected Mode):
  Hardware Tasks
- Der Zustand wird beim Wechsel im *Task State Segment (TSS)* gesichert
- *Task Register* zeigt (über eine Indirektion in GDT) auf aktuelles TSS
  - Laden/Ändern mittels `ltr`-Instruktion
  - Segment Selektor (in GDT) als Parameter

LDT
DS
SS
CS
ES
DI
SI
BP
SP
BX
DX
CX
AX
FLAG
IP
SS2
SP2
SS1
SP1
SS0
SP0
Link

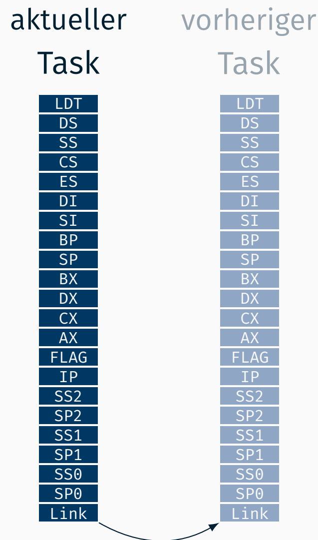
Task State Segment (16 bit)

- Ab 286er (16 bit Protected Mode):
  Hardware Tasks
- Der Zustand wird beim Wechsel im *Task State Segment (TSS)* gesichert
- *Task Register* zeigt (über eine Indirektion in GDT) auf aktuelles TSS
  - Laden/Ändern mittels `ltr`-Instruktion
  - Segment Selektor (in GDT) als Parameter
- Stackpointer für jeden (privilegierten) Ring

TSS
LDT
DS
SS
CS
ES
DI
SI
BP
SP
BX
DX
CX
AX
FLAG
IP
SS2
SP2
SS1
SP1
SS0
SP0
Link

Task State Segment (16 bit)

- Ab 286er (16 bit Protected Mode):
  Hardware Tasks
- Der Zustand wird beim Wechsel im *Task State Segment (TSS)* gesichert
- *Task Register* zeigt (über eine Indirektion in GDT) auf aktuelles TSS
  - Laden/Ändern mittels `ltr`-Instruktion
  - Segment Selektor (in GDT) als Parameter
- Stackpointer für jeden (privilegierten) Ring
- Verlinkung bei (Hardware Task) Umschaltung (via `call` / `jmp` zu Task oder Interrupt)



Task State Segment (32 bit)

- Ab 386er auch für 32 bit Protected Mode
  - mit Unterstützung für Paging
  - Steuerung von I/O Berechtigungen

→ einfaches Multitasking in Hardware!

SSP	
I/O	T
	LDT
	GS
	FS
	DS
	SS
	CS
	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3	
	SS2
ESP2	
	SS1
ESP1	
	SS0
ESP0	
	Link

Task State Segment (32 bit)

- Ab 386er auch für 32 bit Protected Mode
  - mit Unterstützung für Paging
  - Steuerung von I/O Berechtigungen

➔ einfaches Multitasking in Hardware!

- Performance ist jedoch ein Problem
  (es wird immer der komplette CPU Zustand gewechselt
  ➔ Segmente werden neu geprüft [teuer], selbst wenn sie sich nicht ändern [= Normalfall bei Betriebssysteme mit Paging])
- Nicht portabel (auf andere Architekturen)

SSP	
I/O	T
	LDT
	GS
	FS
	DS
	SS
	CS
	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3	
	SS2
ESP2	
	SS1
ESP1	
	SS0
ESP0	
	Link

Task State Segment (32 bit)

ISDMv3 7.2.1

- Ab 386er auch für 32 bit Protected Mode
  - mit Unterstützung für Paging
  - Steuerung von I/O Berechtigungen

→ einfaches Multitasking in Hardware!

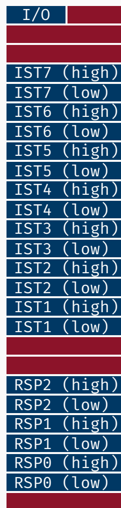
- Performance ist jedoch ein Problem
  (es wird immer der komplette CPU Zustand gewechselt
  → Segmente werden neu geprüft [teuer], selbst wenn sie sich nicht ändern [= Normalfall bei Betriebssysteme mit Paging])
- Nicht portabel (auf andere Architekturen)

→ Verwendung von softwarebasierten Kontextwechsel in modernen BS
  - ein TSS pro Kern (unabhängig von Tasks)
  - für Stackpointer und ggf. I/O Berechtigung

SSP	
I/O	T
	LDT
	GS
	FS
	DS
	SS
	CS
	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3	
	SS2
ESP2	
	SS1
ESP1	
	SS0
ESP0	
	Link

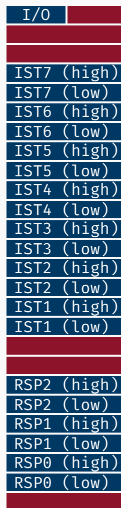
Task State Segment (64 bit)

- Im Long Mode gibt es immer noch das TSS



Task State Segment (64 bit)

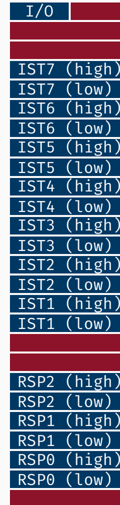
- Im Long Mode gibt es immer noch das TSS, aber kein Unterstützung für Hardware Tasks



Task State Segment (64 bit)

- Im Long Mode gibt es immer noch das TSS, aber kein Unterstützung für Hardware Tasks

→ nur noch die für moderne Betriebssysteme relevanten Features

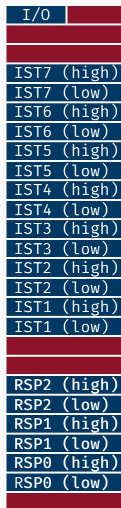


Task State Segment (64 bit)

- Im Long Mode gibt es immer noch das TSS, aber kein Unterstützung für Hardware Tasks

→ nur noch die für moderne Betriebssysteme relevanten Features

  - Stackpointer für jeden (privilegierten) Ring (`RSP0` *für Ring 0 interessant in STUBS*)

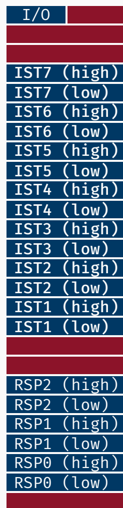


Task State Segment (64 bit)

- Im Long Mode gibt es immer noch das TSS, aber kein Unterstützung für Hardware Tasks

➔ nur noch die für moderne Betriebssysteme relevanten Features

  - Stackpointer für jeden (privilegierten) Ring (RSP0 *für Ring 0 interessant in* StuBS)
  - zusätzlich noch *Interrupt Stack Table* *(nicht notwendig für* StuBS)



Task State Segment (64 bit)

- Im Long Mode gibt es immer noch das TSS, aber kein Unterstützung für Hardware Tasks

➔ nur noch die für moderne Betriebssysteme relevanten Features
  - Stackpointer für jeden (privilegierten) Ring (`RSP0` *für Ring 0 interessant in* STUBS)
  - zusätzlich noch *Interrupt Stack Table* *(nicht notwendig für* STUBS)
  - Steuerung von I/O Berechtigungen *(auch nicht notwendig für* STUBS)

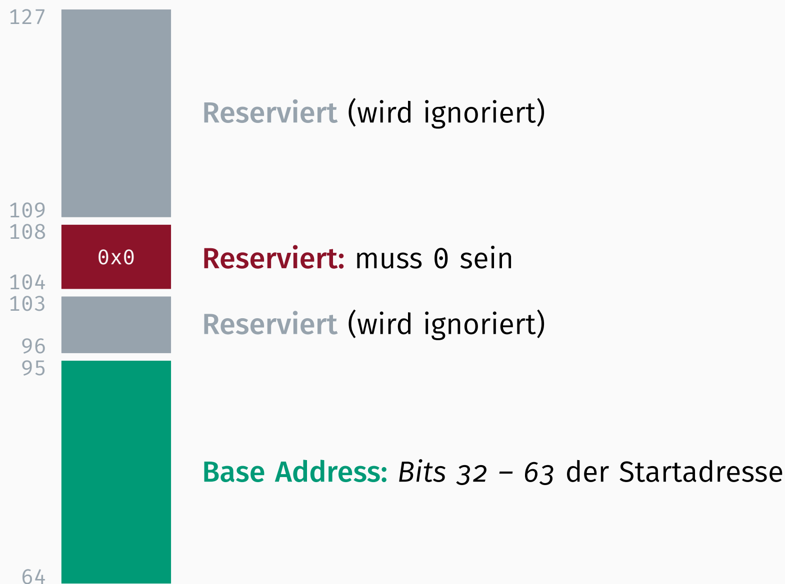
I/O	
IST7 (high)	
IST7 (low)	
IST6 (high)	
IST6 (low)	
IST5 (high)	
IST5 (low)	
IST4 (high)	
IST4 (low)	
IST3 (high)	
IST3 (low)	
IST2 (high)	
IST2 (low)	
IST1 (high)	
IST1 (low)	
RSP2 (high)	
RSP2 (low)	
RSP1 (high)	
RSP1 (low)	
RSP0 (high)	
RSP0 (low)	

64 bit TSS Deskriptoreintrag

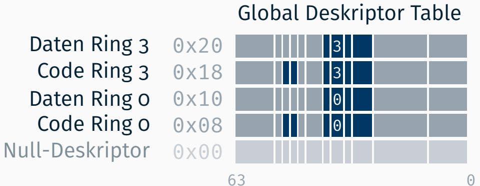
ISDMv3 7.2.3

Bits	Wert	Feld
63–56		**Base Address (high):** *Bits 24 – 31* der Startadresse
55	`0x0`	**Granularity:** Byte (`0`) oder 4 KB-Block (`1`)
54	`0x0`	**Reserviert**
53	`0x0`	**Reserviert**
52		**Available:** zur freien Verwendung
51–48		**Segment Limit (high):** *Oberer* Teil der Segmentgröße
47	`0x1`	**Present:** Eintrag aktiv (`1`) oder inaktiv (`0`)
46–45	`0x0`	**Descriptor Privilege Level:** Erlaubter Ring (bei uns: 0)
44	`0x0`	**Descriptor type:** System (`0`) oder Code/Daten (`1`)
43–40	`0x9`	**Type** `0x9` für TSS; *Busy Flag* (Bit 41) wird gesetzt wenn aktiv
39–16		**Base Address (low):** Bits 0 – 23 der Startadresse
15–0		**Segment Limit (low):** *Unterer* Teil der Segmentgröße

64 bit TSS Deskriptoreintrag (Fortsetzung)

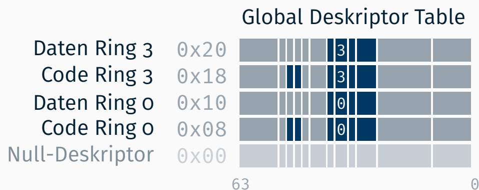


TSS Deskriptoreintrag in GDT



TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren



Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen

Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

Global Deskriptor Table

(Fortsetzung)	0x30	
TSS Deskriptor	0x28	0
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 0

TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen

Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

Global Deskriptor Table

	Eintrag	Offset
	(Fortsetzung)	0x30
tr →	TSS Deskriptor	0x28
	Daten Ring 3	0x20
	Code Ring 3	0x18
	Daten Ring 0	0x10
	Code Ring 0	0x08
	Null-Deskriptor	0x00

63 … 0

TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion

Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

Global Deskriptor Table

(Fortsetzung)	0x30	
`tr` → TSS Deskriptor	0x28	0
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 0

TSS Deskriptoreintrag in GDT

ISDMv3 7.2.4

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion
   - zeigt auf Segment Offset (hier: `0x28`)

Task State Segment

Task State Segment
Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

Global Deskriptor Table

	Eintrag	Offset	DPL
	(Fortsetzung)	0x30	
tr →	TSS Deskriptor	0x28	0
	Daten Ring 3	0x20	3
	Code Ring 3	0x18	3
	Daten Ring 0	0x10	0
	Code Ring 0	0x08	0
	Null-Deskriptor	0x00	

63 … 0

TSS Deskriptoreintrag in GDT

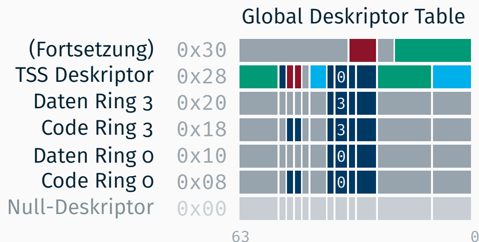
1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion
   - zeigt auf Segment Offset (hier: `0x28`)
4. `RSP0` setzen

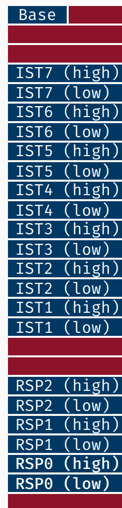
TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion
   - zeigt auf Segment Offset (hier: `0x28`)
4. `RSP0` setzen
   - auf Kernelstack

TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion
   - zeigt auf Segment Offset (hier: `0x28`)
4. `RSP0` setzen
   - auf Kernelstack des aktuellen Threads





TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren
2. TSS Deskriptor Eintrag in GDT anlegen
3. Task Register (`tr`) setzen
   - mittels `ltr` Instruktion
   - zeigt auf Segment Offset (hier: `0x28`)
4. `RSP0` setzen
   - auf Kernelstack des aktuellen Threads
   - bei jedem Kontextwechsel anpassen

Global Deskriptor Table

(Fortsetzung) 0x30
TSS Deskriptor 0x28 (0)
Daten Ring 3 0x20 (3)
Code Ring 3 0x18 (3)
Daten Ring 0 0x10 (0)
Code Ring 0 0x08 (0)
Null-Deskriptor 0x00

63 0

Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

TSS Deskriptoreintrag in GDT bei MPStuBSmI

Im Mehrkernbetrieb hat jeder Kern

Global Deskriptor Table

(Fortsetzung)	0x30	
TSS Deskriptor	0x28	0
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Deskriptor	0x00	

63 … 0

Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

TSS Deskriptoreintrag in GDT bei MPStuBSmi

Im Mehrkernbetrieb hat jeder Kern

- einen eigenes TSS allokiert

TSS Deskriptoreintrag in GDT bei MPStubsmi

Im Mehrkernbetrieb hat jeder Kern

- einen eigenes TSS allokiert
- einen eigenen TSS Deskriptor Eintrag

Global Deskriptor Table

(Fortsetzung) 0x40
Core 1 TSS Deskriptor 0x38
(Fortsetzung) 0x30
Core 0 TSS Deskriptor 0x28
Daten Ring 3 0x20
Code Ring 3 0x18
Daten Ring 0 0x10
Code Ring 0 0x08
Null-Deskriptor 0x00

63 0

Core 1
Task State Segment

Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

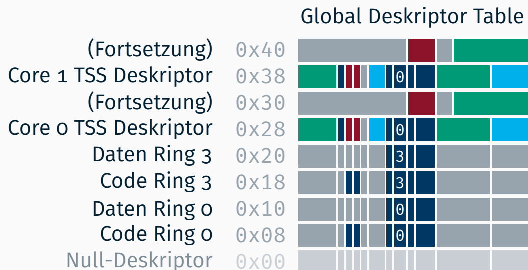
Core 0
Task State Segment

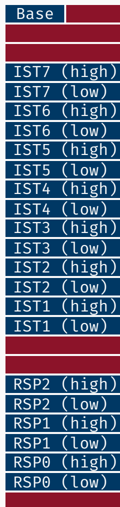
Base
IST7 (high)
IST7 (low)
IST6 (high)
IST6 (low)
IST5 (high)
IST5 (low)
IST4 (high)
IST4 (low)
IST3 (high)
IST3 (low)
IST2 (high)
IST2 (low)
IST1 (high)
IST1 (low)
RSP2 (high)
RSP2 (low)
RSP1 (high)
RSP1 (low)
RSP0 (high)
RSP0 (low)

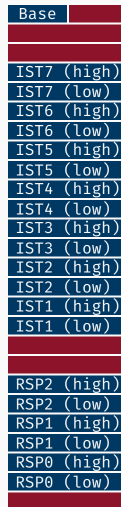
TSS Deskriptoreintrag in GDT bei MPStuBSmi

Im Mehrkernbetrieb hat jeder Kern

- einen eigenes TSS allokiert
- einen eigenen TSS Deskriptor Eintrag
- das Task Register (`tr`) passend gesetzt

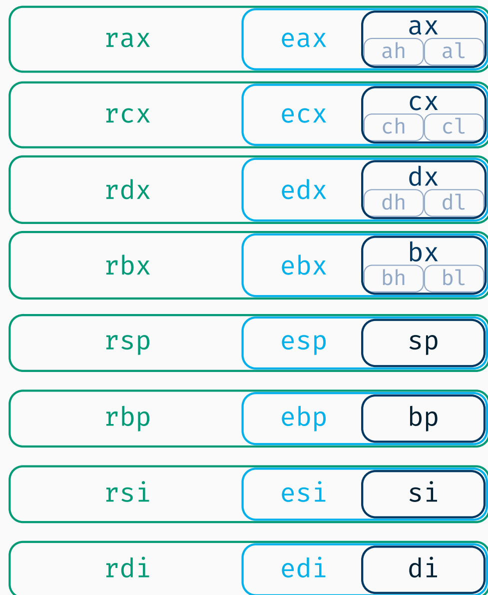






Core Local Storage (7.5 ECTS)

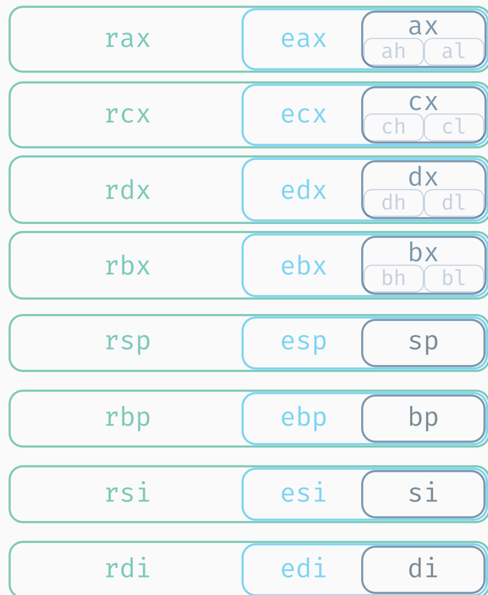
Long Mode Register



+ 16× SSE Register (XMM, 128 bit)

+ Kontrollregister + Debugregister

Long Mode Register



+ 16× SSE Register (XMM, 128 bit)

+ Kontrollregister + Debugregister

```c
// gcc -O0 example.c -pthread
#include <stdio.h>
#include <pthread.h>

long foo = 1;
__thread long bar = 1;

static void * action(void * tid) {
  for (int i = 0; i < 3; i++)
    printf("Thread %p: foo = %ld, bar = %ld\n",
           tid, ++foo, ++bar);
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, action, (void*)1);
  pthread_create(&t2, NULL, action, (void*)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("foo = %ld, bar = %ld\n", foo, bar);
  return 0;
}
```

```c
// gcc -O0 example.c -pthread
#include <stdio.h>
#include <pthread.h>

long foo = 1;
__thread long bar = 1;

static void * action(void * tid) {
  for (int i = 0; i < 3; i++)
    printf("Thread %p: foo = %ld, bar = %ld\n",
           tid, ++foo, ++bar);
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, action, (void*)1);
  pthread_create(&t2, NULL, action, (void*)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("foo = %ld, bar = %ld\n", foo, bar);
  return 0;
}
```

```
$ gcc -O0 example.c -pthread
$ ./a.out
Thread 0x1: foo = 2, bar = 2
Thread 0x2: foo = 3, bar = 2
Thread 0x1: foo = 4, bar = 3
Thread 0x1: foo = 5, bar = 4
Thread 0x2: foo = 6, bar = 3
Thread 0x2: foo = 7, bar = 4
foo = 7, bar = 1
```

(Wettlaufsituation → auch andere Ausgaben möglich)

```c
// gcc -O0 example.c -pthread
#include <stdio.h>
#include <pthread.h>

long foo = 1;
__thread long bar = 1;

static void * action(void * tid) {
  for (int i = 0; i < 3; i++)
    printf("Thread %p: foo = %ld, bar = %ld\n",
           tid, ++foo, ++bar);
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, action, (void*)1);
  pthread_create(&t2, NULL, action, (void*)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("foo = %ld, bar = %ld\n", foo, bar);
  return 0;
}
```

```
; objdump -d a.out -M intel

; ...

action:
; ...

; ++foo
mov rax, [rip+0x2e39]  ; 4010
add rax, 0x1
mov [rip+0x2e2e], rax  ; 4010

; ++bar
mov rax, fs:0xfffffffffffffff8
add rax, 0x1
mov fs:0xfffffffffffffff8, rax
```

```c
// gcc -O0 example.c -pthread
#include <stdio.h>
#include <pthread.h>

long foo = 1;
__thread long bar = 1;

static void * action(void * tid) {
  for (int i = 0; i < 3; i++)
    printf("Thread %p: foo = %ld, bar = %ld\n",
           tid, ++foo, ++bar);
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, action, (void*)1);
  pthread_create(&t2, NULL, action, (void*)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("foo = %ld, bar = %ld\n", foo, bar);
  return 0;
}
```

```asm
; objdump -d a.out -M intel

; ...

action:
; ...

; ++foo
mov rax, [rip+0x2e39]  ; 4010
add rax, 0x1
mov [rip+0x2e2e], rax  ; 4010

; ++bar
mov rax, fs:-8
add rax, 0x1
mov fs:-8, rax
```

Thread Local Storage (TLS)

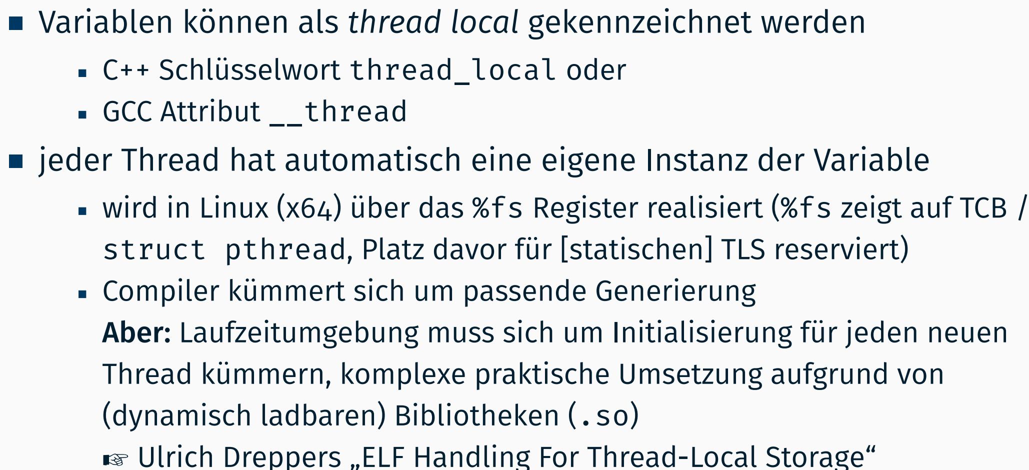
- Variablen können als *thread local* gekennzeichnet werden
  - C++ Schlüsselwort `thread_local` oder
  - GCC Attribut `__thread`
- jeder Thread hat automatisch eine eigene Instanz der Variable

Thread Local Storage (TLS)

- Variablen können als *thread local* gekennzeichnet werden
  - C++ Schlüsselwort `thread_local` oder
  - GCC Attribut `__thread`
- jeder Thread hat automatisch eine eigene Instanz der Variable
  - wird in Linux (x64) über das `%fs` Register realisiert (`%fs` zeigt auf TCB / `struct pthread`, Platz davor für [statischen] TLS reserviert)

Thread Local Storage (TLS)

GCC Doc



- Variablen können als *thread local* gekennzeichnet werden
  - C++ Schlüsselwort `thread_local` oder
  - GCC Attribut `__thread`
- jeder Thread hat automatisch eine eigene Instanz der Variable
  - wird in Linux (x64) über das `%fs` Register realisiert (`%fs` zeigt auf TCB / `struct pthread`, Platz davor für [statischen] TLS reserviert)
  - Compiler kümmert sich um passende Generierung
    Aber: Laufzeitumgebung muss sich um Initialisierung für jeden neuen Thread kümmern, komplexe praktische Umsetzung aufgrund von (dynamisch ladbaren) Bibliotheken (`.so`)
    ☞ Ulrich Dreppers „ELF Handling For Thread-Local Storage“

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)

```
namespace Core {
  namespace Local {
    struct cache_aligned Storage {
      unsigned id;  // Core ID
      // ... später mehr
    };
  }
}
```

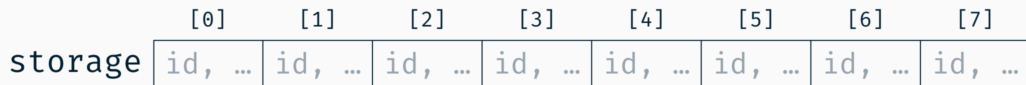
Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)

```
namespace Core {
  namespace Local {
    struct cache_aligned Storage {
      unsigned id;  // Core ID
      // ... später mehr
    } storage[Core::MAX];
  }
}
```

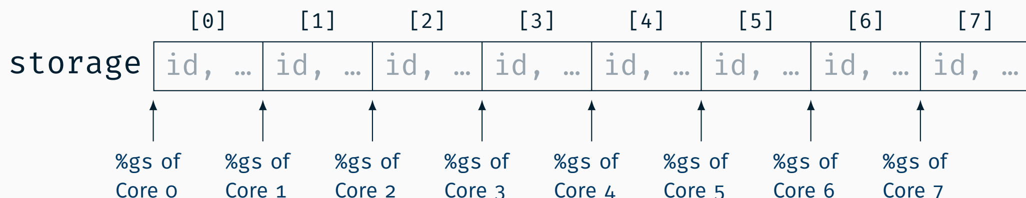
Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)



Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das `%gs` Register eines jeden Kerns zeigt auf den jeweiligen Eintrag



Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das `%gs` Register eines jeden Kerns zeigt auf den jeweiligen Eintrag

```
// Beispiel für Core 2:
//    %gs -> &(storage[2]);
uintptr_t gs = reinterpret_cast<uintptr_t>(storage + 2);
Core::MSR<MSR_GS_BASE>::write(gs);
Core::MSR<MSR_SHADOW_GS_BASE>::write(0);
```

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das %gs Register eines jeden Kerns zeigt auf den jeweiligen Eintrag
    - setzen mittels MSR_GS_BASE sowie MSR_SHADOW_GS_BASE

```
// Beispiel für Core 2:
//    %gs -> &(storage[2]);
uintptr_t gs = reinterpret_cast<uintptr_t>(storage + 2);
Core::MSR<MSR_GS_BASE>::write(gs);
Core::MSR<MSR_SHADOW_GS_BASE>::write(0);
```

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das %gs Register eines jeden Kerns zeigt auf den jeweiligen Eintrag
  - setzen mittels MSR_GS_BASE sowie MSR_SHADOW_GS_BASE
  - (nur) bei Ringwechsel swapgs ausführen

```
interrupt_entry_%1:
  ; ...
  ; Prüfen ob von Ring 3 -> 0
  swapgs
  ; ... interrupt_handler ...
  ; ggf wieder swapgs
  iretq
```

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das `%gs` Register eines jeden Kerns zeigt auf den jeweiligen Eintrag
  - setzen mittels MSR_GS_BASE sowie MSR_SHADOW_GS_BASE
  - (nur) bei Ringwechsel `swapgs` ausführen

```
interrupt_entry_%1:
  ; ...
  ; Prüfen ob von Ring 3 -> 0 (Tipp: altes %cs liegt auf Stack)
  swapgs
  ; ... interrupt_handler ...
  ; ggf wieder swapgs
  iretq
```

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das `%gs` Register eines jeden Kerns zeigt auf den jeweiligen Eintrag
  - setzen mittels MSR_GS_BASE sowie MSR_SHADOW_GS_BASE
  - (nur) bei Ringwechsel `swapgs` ausführen
- Zugriff mittels `%gs:OFFSET`

```
inline unsigned getID() {
  unsigned id;
  asm volatile("mov %%gs:0x0, %0"
               : "=r"(id));

  return id;
}
```

Core Local Storage

- im Gegensatz zu TLS händisch, ohne Compiler-Magie
- Verwendung einer Struktur (`Core::Local::Storage`)
- jeder Kern hat eine eigene Instanz der Struktur (→ Array)
- das `%gs` Register eines jeden Kerns zeigt auf den jeweiligen Eintrag
  - setzen mittels MSR_GS_BASE sowie MSR_SHADOW_GS_BASE
  - (nur) bei Ringwechsel `swapgs` ausführen
- Zugriff mittels `%gs:OFFSET`

```
inline unsigned getID() {
  unsigned id;
  asm volatile("mov %%gs:%c1, %0"
               : "=r"(id)
               : "n"(__builtin_offsetof(struct Storage, id)));
  return id;
}
```

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch `Core::Local::getID()`:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register
  (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle
  (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch `Core::Local::getID()`:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Aufgabe: Benchmark der Varianten

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch `Core::Local::getID()`:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Aufgabe: Benchmark der Varianten mittels Timestamp Counter (TSC)

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch `Core::Local::getID()`:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Aufgabe: Benchmark der Varianten mittels Timestamp Counter (TSC) auf Testrechner und in KVM!

Core Local Storage: Einsatz

Abrufen der Kern ID via `Core::getID()`:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch `Core::Local::getID()`:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Aufgabe: Benchmark der Varianten mittels Timestamp Counter (TSC) auf Testrechner und in KVM! (Wieso gibt es da signifikante Unterschiede?)

Core Local Storage: Einsatz

Abrufen der Kern ID via **`Core::getID()`**:

- liest 8 bit **LAPIC ID** aus Identification Register (Memory mapped an `0xfee00020`)
- Nachschlagen der **Core ID** in Lookup-Tabelle (wird beim Systemstart durch `Core::init()` angelegt)

Ersatz durch **`Core::Local::getID()`**:

- Abruf im passenden Index des `Storage`-Arrays mittels `%gs`

Aufgabe: Benchmark der Varianten mittels Timestamp Counter (TSC) auf Testrechner und in KVM! (Wieso gibt es da signifikante Unterschiede?)

(Ein weiteres & wichtigeres Einsatzgebiet von Core Local Storage folgt in Aufgabe 2)

Fragen?

Nächste Woche am Dienstag das Assembler-Seminar?