

Übung zu Betriebssystemtechnik

Aufgabe 1: Privilegientrennung

23. April 2025

Dustin Nguyen, Maximilian Ott & Phillip Raffeck

Lehrstuhl für Informatik 4

Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware

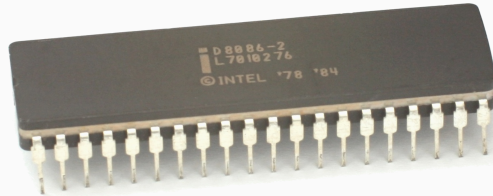


Friedrich-Alexander-Universität
Technische Fakultät

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

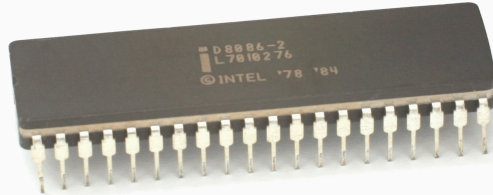


Real Mode



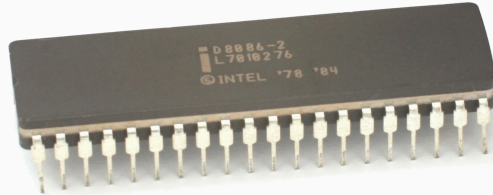
Quelle: Wikipedia

Real Mode



Quelle: Wikipedia

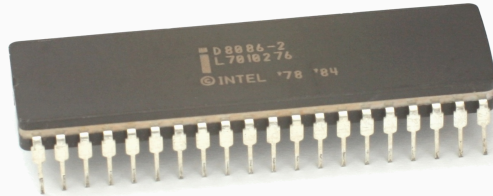
8086 16 bit, 1 MB Speicher adressierbar



Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

- Segmentierung mit 20 bit Adressbus

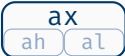


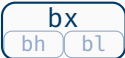
Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

- Segmentierung mit 20 bit Adressbus
- 14 Register

Real Mode Register

Akkumulator 

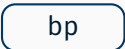
Basisregister 

Zählerregister 

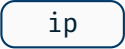
Datenregister 

Quellindex 

Zielindex 

Basiszeiger für
Stapelrahmen 

Stapelzeiger 

Instruktionszeiger 

Statusregister 

Codesegment 

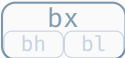
Datensegment 

Stacksegment 

Extrasegment 

Real Mode Register

Akkumulator 

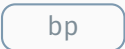
Basisregister 

Zählerregister 

Datenregister 

Quellindex 

Zielindex 

Basiszeiger für
Stapelrahmen 

Stapelzeiger 

Instruktionszeiger 

Statusregister 

Codesegment 

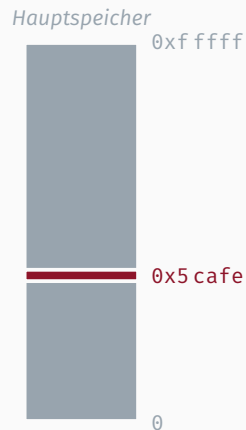
Datensegment 

Stacksegment 

Extrasegment 

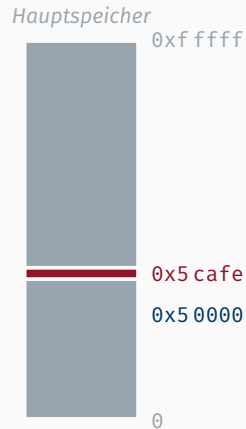


Zugriff auf **Zieldaten** in 0x5 cafe



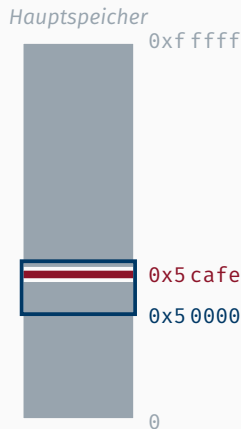
Zugriff auf **Zieldaten** in 0x5 cafe

1. Setze **ds** auf 0x5000



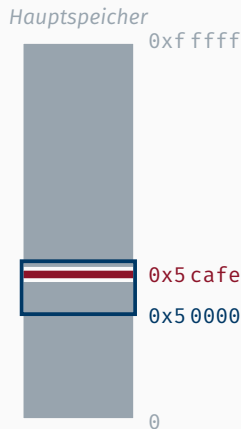
Zugriff auf **Zieldaten** in 0x5 cafe

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



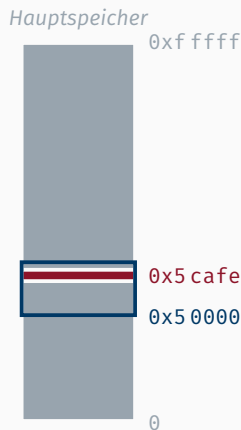
Zugriff auf **Ziel**daten in 0x5 cafe

1. Setze **ds** auf 0x5000
(Bereich 0x5 0000 – 0x5 ffff)
2. Zugriff über **ds:0xcafe**



Zugriff auf **Ziel**daten in 0x5 cafe

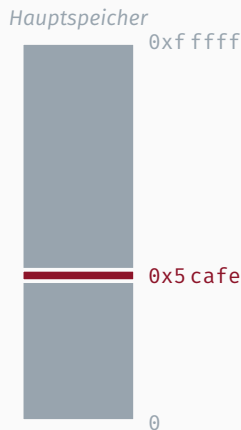
1. Setze **ds** auf 0x5000
(Bereich 0x5 0000 – 0x5 ffff)
2. Zugriff über **ds:0xcafe**
($\text{ds} \times 0x10 + 0xcafe = 0x5\text{ cafe}$)



Zugriff auf **Ziel**daten in 0x5 cafe

1. Setze **ds** auf 0x5000
(Bereich 0x5 0000 – 0x5 ffff)
2. Zugriff über **ds:0xcafe**
($\text{ds} \times 0x10 + 0xcafe = 0x5\text{ cafe}$)

Alternativ:

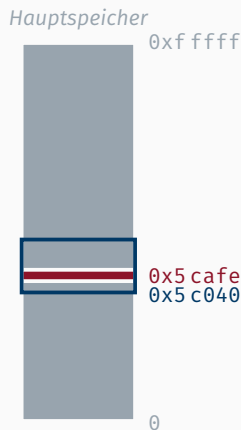


Zugriff auf **Ziel**daten in 0x5 cafe

1. Setze **ds** auf 0x5000
(Bereich 0x5 0000 – 0x5 ffff)
2. Zugriff über **ds:0xcafe**
($\text{ds} \times 0x10 + 0xcafe = 0x5\text{ cafe}$)

Alternativ:

1. Setze **ds** auf 0x5c04
(Bereich 0x5 c040 – 0x6 c039)

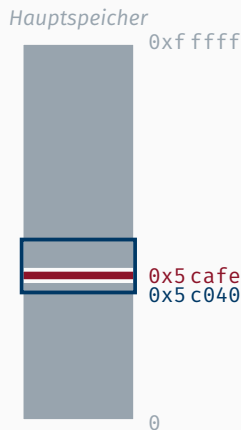


Zugriff auf **Ziel**daten in 0x5 cafe

1. Setze **ds** auf 0x5000
(Bereich 0x5 0000 – 0x5 ffff)
2. Zugriff über **ds:0xcafe**
($\text{ds} \times 0x10 + 0xcafe = 0x5\text{ cafe}$)

Alternativ:

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





Quelle: Wikipedia



Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar



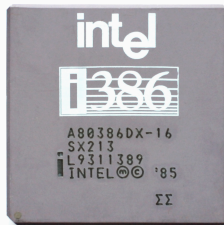
Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus



Quelle: Wikipedia



Quelle: Wikipedia

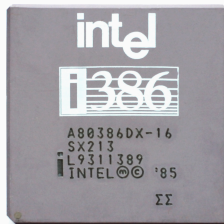
80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar



Quelle: Wikipedia



Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

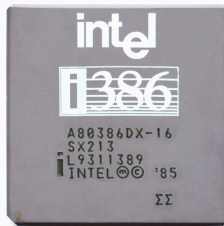
- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

- Segmentierung mit 32 bit Adressbus



Quelle: Wikipedia



Quelle: Wikipedia

80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

Name aufgrund der Schutzringe
(*Privilege Level*)

80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

Name aufgrund der Schutzringe
(*Privilege Level*)



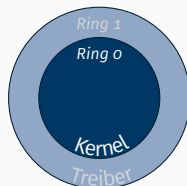
80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

Name aufgrund der Schutzringe
(*Privilege Level*)



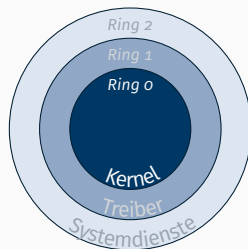
80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

Name aufgrund der Schutzringe
(*Privilege Level*)



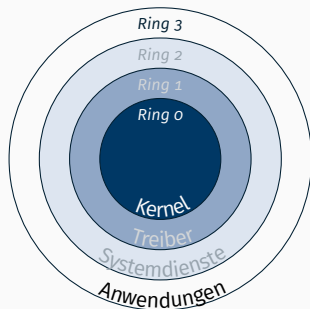
80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

Name aufgrund der Schutzringe
(*Privilege Level*)



80286 16 bit, 16 MB Speicher adressierbar

- Segmentierung mit 24 bit Adressbus

80386 32 bit, 4 GB Speicher adressierbar

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

- Einführung von Deskriptortabellen
 - GDT** Global Descriptor Table
 - LDT** Local Descriptor Table
 - IDT** Interrupt Descriptor Table

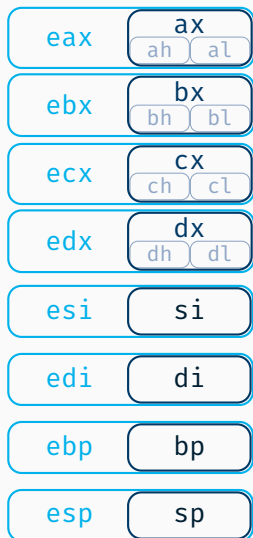
- Einführung von Deskriptortabellen
 - GDT** Global Descriptor Table
 - LDT** Local Descriptor Table
 - IDT** Interrupt Descriptor Table
 - jeder der Deskriptoren (max. 8 192 pro Tabelle) besteht aus
 - Basisadresse (24 bit bzw. 32 bit)
 - Länge (16 bit bzw. 20 bit)
 - Parameter wie Typ, Berechtigung, Aktiv, ...
- Segmentselektoren zeigen nun auf Einträge in GDT/LDT

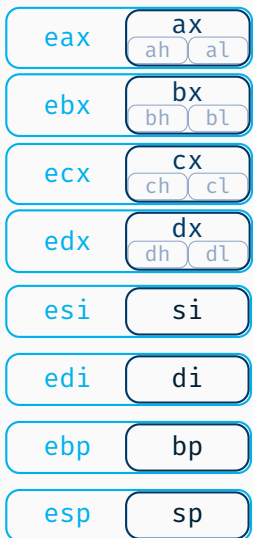
16 bit Segmentdeskriptoreintrag





32 bit Protected Mode Register





Zugriff auf **Zieldaten** in 0x210 f00d

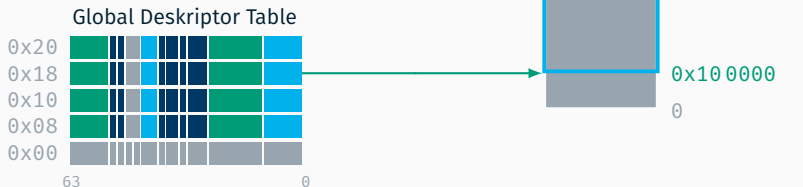


- über GDT



Zugriff auf **Ziel**daten in 0x210 f00d

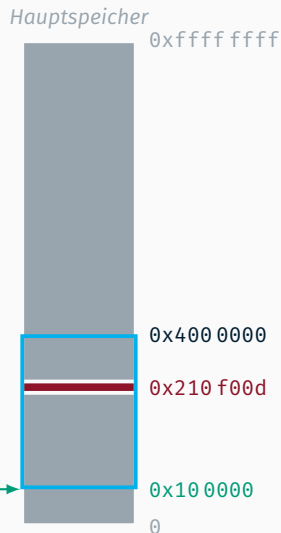
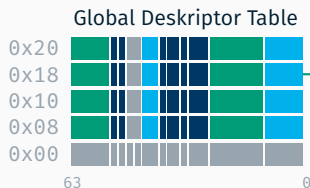
- über **GDT** Eintrag 0x18 mit
Base Address 0x10 0000 und
Segment Limit 0x3f0 0000



Zugriff auf **Ziel**daten in 0x210 f00d

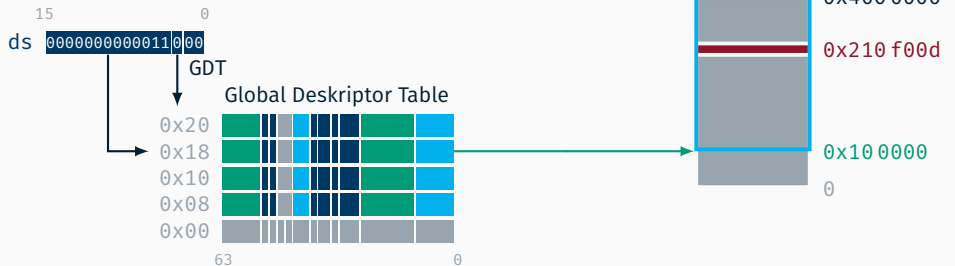
- über **GDT** Eintrag 0x18 mit
Base Address 0x10 0000 und
Segment Limit 0x3f0 0000
- mit Segmentregister **ds**

15 0
ds **Descriptor Index** **T** **PL**



Zugriff auf **Ziel**daten 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`

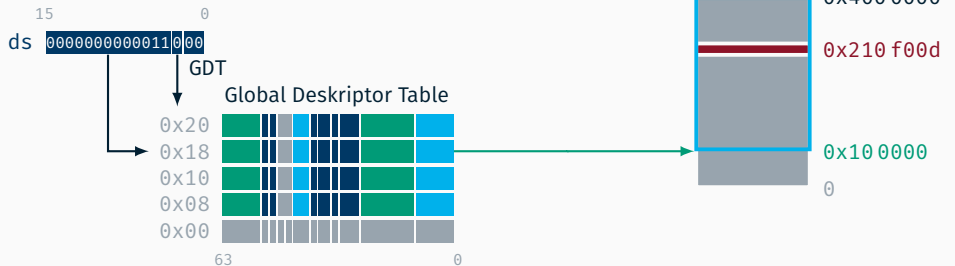
(Quelle, Ziel)

Standard bei objdump

und GCC inline asm

Zugriff auf **Ziel**daten in 0x210 f00d

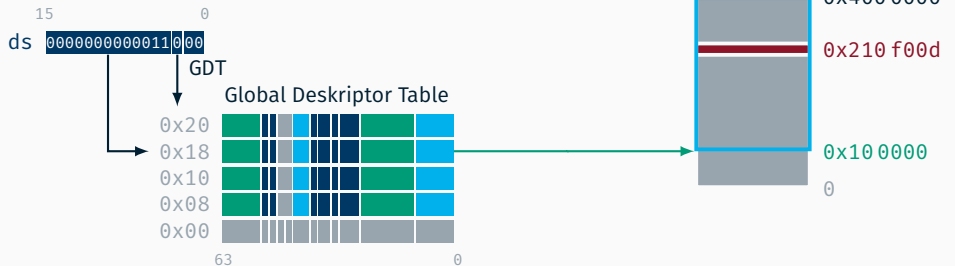
- über **GDT** Eintrag **0x18** mit
Base Address **0x10 0000** und
Segment Limit **0x3f0 0000**
- mit Segmentregister **ds** mit
`mov ax, 0x18`
`mov ds, ax`



Zugriff auf **Ziel**daten in 0x210 f00d

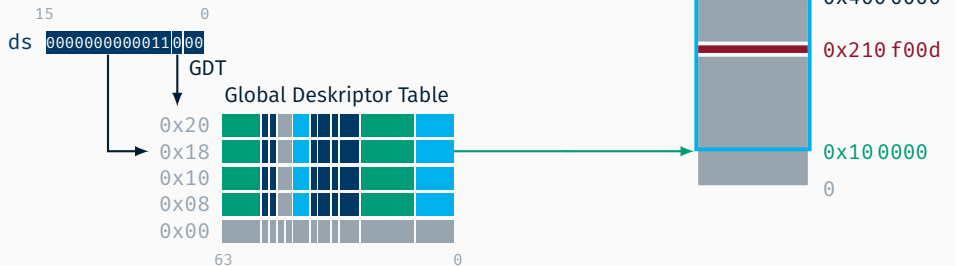
- Offset setzen

```
mov eax, 0x200f00d
```



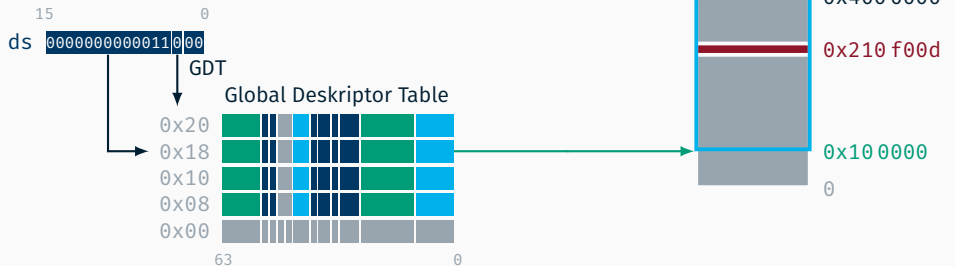
Zugriff auf **Ziel**daten in 0x210 f00d

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



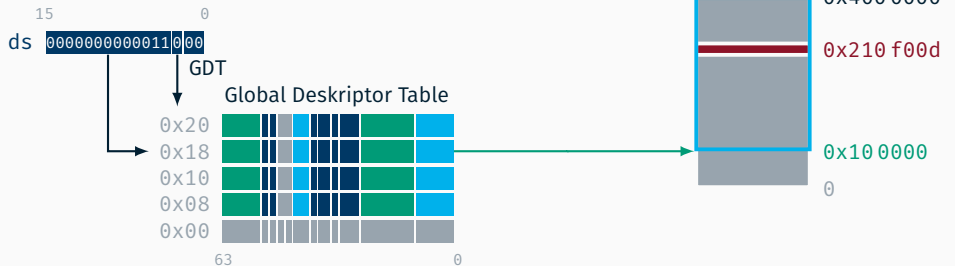
Zugriff auf **Zieldaten** in 0x210 f00d

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



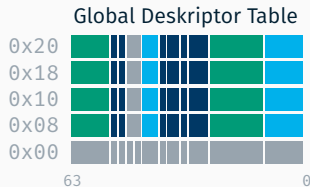
Zugriff auf **Ziel**daten in 0x210 f00d

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



Flaches Speichermodell

→ Zugriff über 0x210 f00d

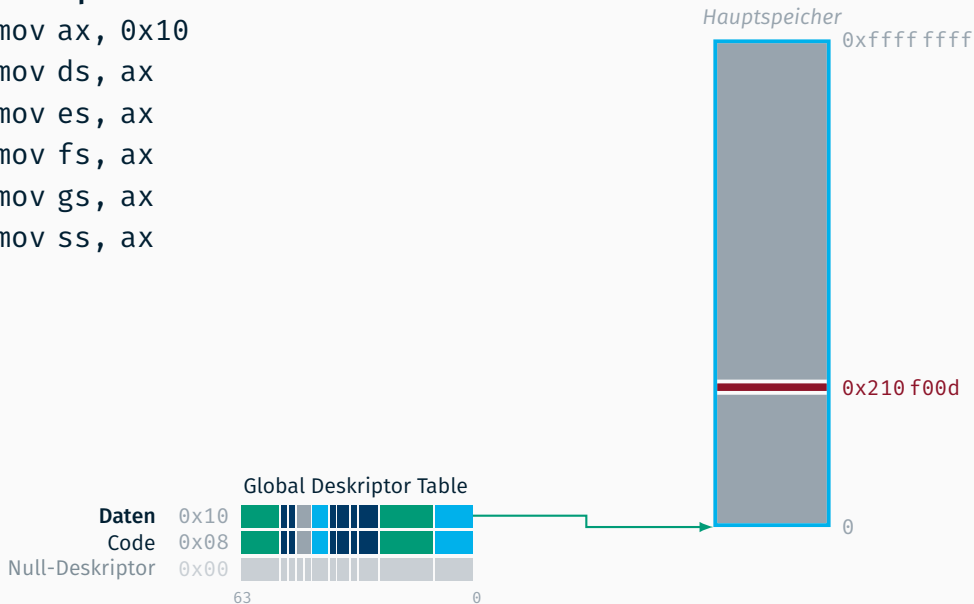


→ Zugriff über 0x210 f00d

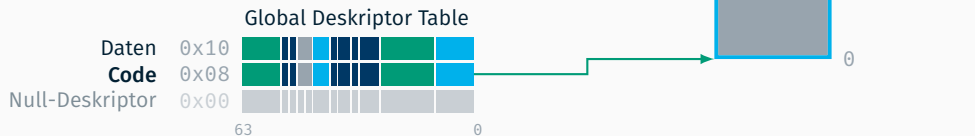


Flaches Speichermodell

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



• • •



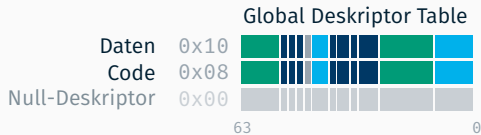
G

64 bit GDT

Flaches Speichermodell



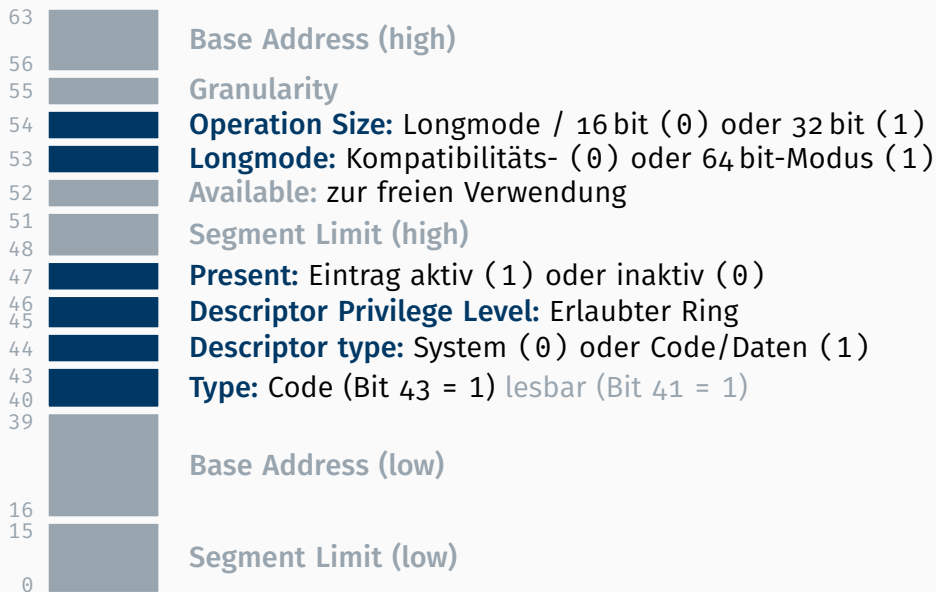
Quelle: Wikipedia



Hauptspeicher

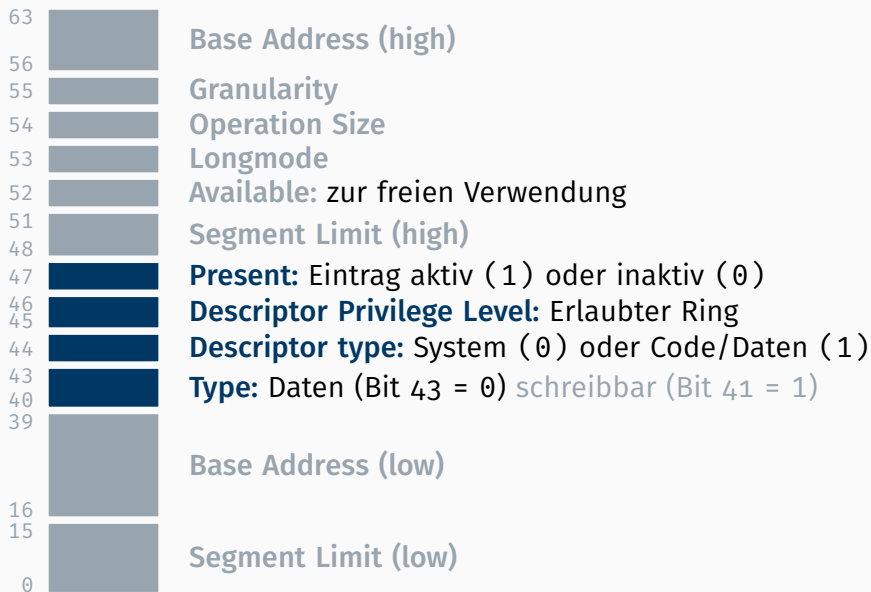






63	0x0	Base Address (high)
56		
55	0x0	Granularity
54	0x0	Operation Size: Longmode / 16 bit (0) oder 32 bit (1)
53	0x1	Longmode: Kompatibilitäts- (0) oder 64 bit-Modus (1)
52	0x0	Available: zur freien Verwendung
51		
48	0x0	Segment Limit (high)
47	0x1	Present: Eintrag aktiv (1) oder inaktiv (0)
46		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

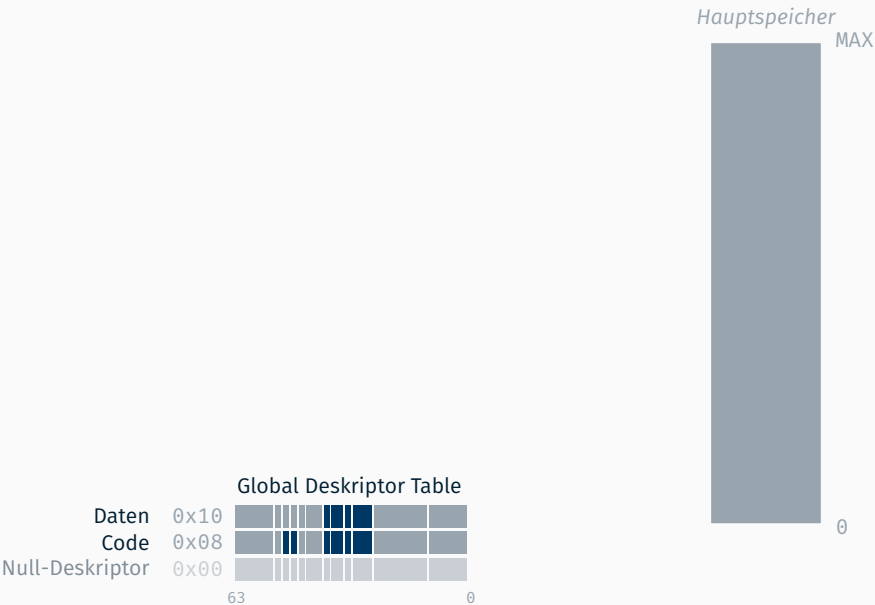
64 bit GDT Datensegmentdeskriptoreintrag ISDMv3 3.4.5.1 & AAPMv2 4.8.2



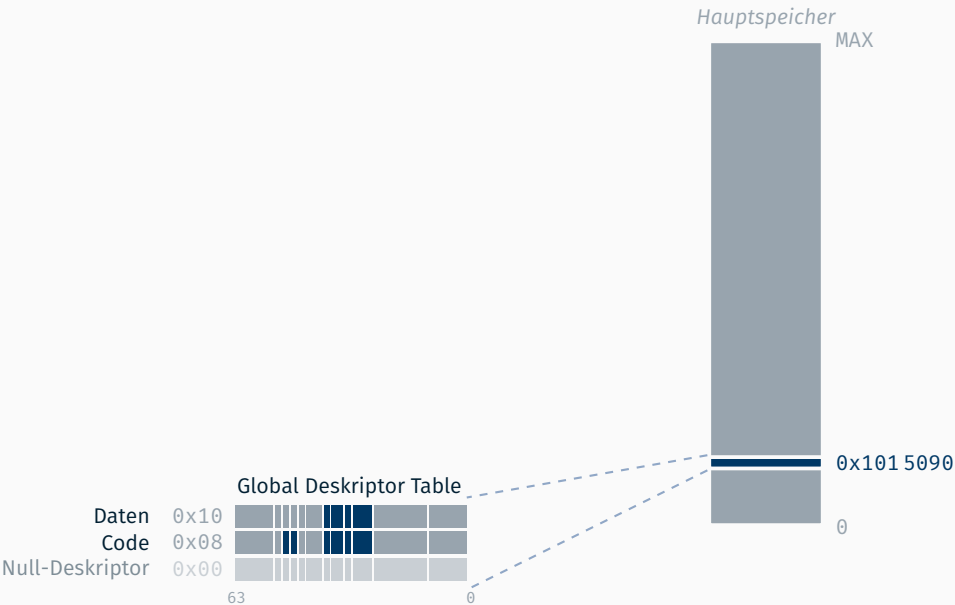
64 bit GDT Datensegmentdeskriptoreintrag ISDMv3 3.4.5.1 & AAPMv2 4.8.2

63	0x0	Base Address (high)
56		
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	0x1	Descriptor type: System (0) oder Code/Daten (1)
43		
40	0x2	Type: Daten (Bit 43 = 0) schreibbar (Bit 41 = 1)
39		
	0x0	Base Address (low)
16		
15		
	0x0	Segment Limit (low)
0		

64 bit GDT (Flaches Speichermodell)

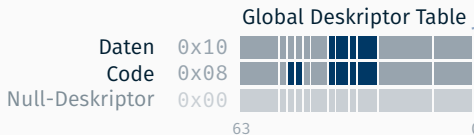
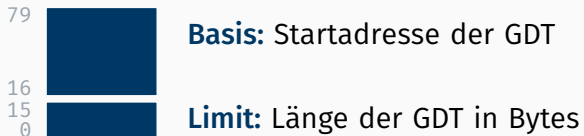


64 bit GDT (Flaches Speichermodell)



64 bit GDT (Flaches Speichermodell)

GDT-Register **gdtr**



Hauptspeicher

MAX

0x00000000 **gdtr** 0

64 bit GDT (Flaches Speichermodell)

GDT-Register `gdt r`

79

0x1015090

Basis: Startadresse der GDT

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

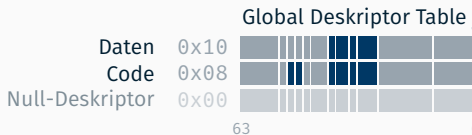
16

15

0

0x17

Limit: Länge der GDT in Bytes



Hauptspeicher

MAX

0x1015090
← `gdt r`

0

64 bit GDT (Flaches Speichermodell)

GDT-Register `gdt r`

79

0x1015090

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

16

15

0

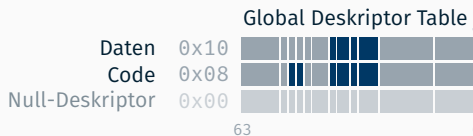
0x17

Limit: Länge der GDT in Bytes

Instruktionen:

`lgdt` in Register laden

`sgdt` aus Register lesen



Hauptspeicher

MAX

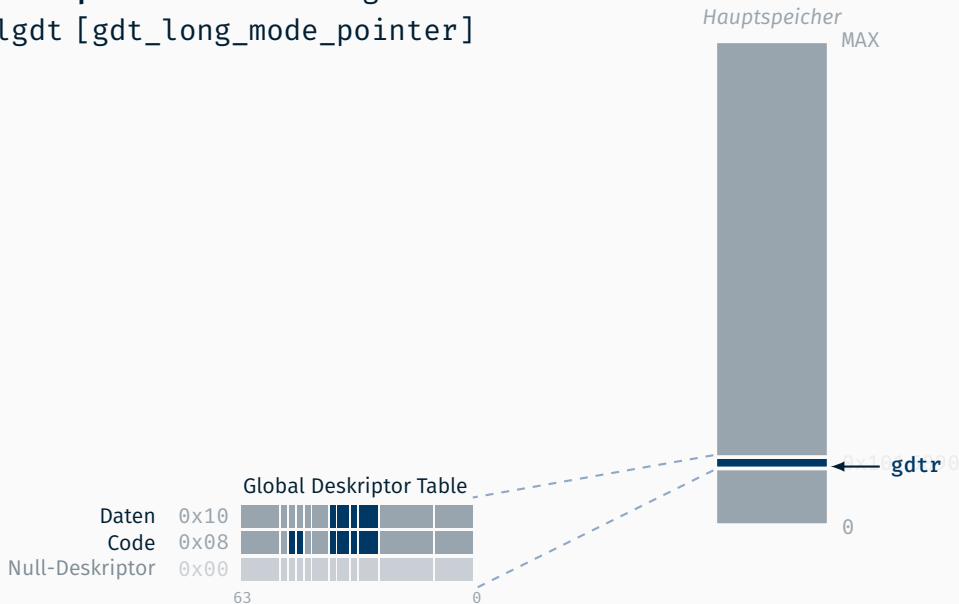
0x1015090
← `gdt r`

0

64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → longmode.asm:

```
lgdt [gdt_long_mode_pointer]
```

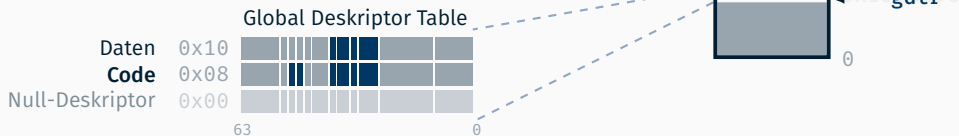


64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → longmode.asm:

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

```
long_mode_start:
```



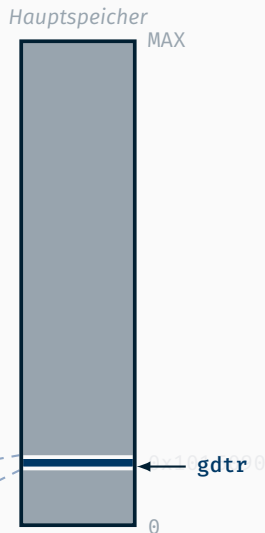
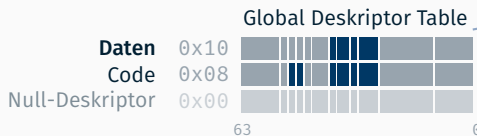
64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → longmode.asm:

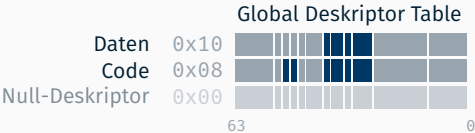
```
lgdt [gdt_long_mode_pointer]  
jmp 0x8:long_mode_start
```

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)



Daten Ring o **0x10**

Code Ring o **0x08**

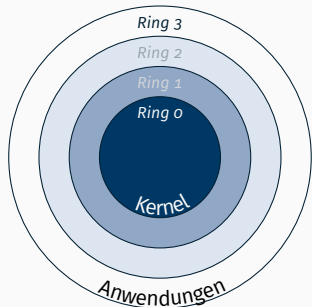
Null-Descriptor **0x00**

63 0

Global Descriptor Table

- 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 Prozessorkerns (hlt)

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

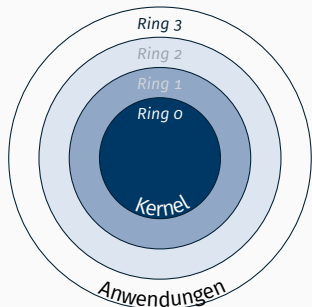


Daten Ring 3	0x20
Code Ring 3	0x18
Daten Ring 0	0x10
Code Ring 0	0x08
Null-Deskriptor	0x00

63

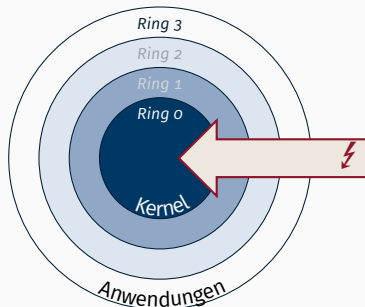
- Laden von GDT/LDT/IDT (lgdt/lldt/lidt) und Taskregister (ltr)
- Ändern (mov) von Kontroll- (cr0-3) und Debugregister (dr0-7)
- Konfigurieren von Model-Specific Register (rdmsr/writemsr)
- Invalidieren von Cache (invd/wbinvd) und TLB (invlpg)
- Sperren/Erlauben von Unterbrechungen (cli/sti)
- Zugriff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

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

[illegible]

- 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 Prozessorkerns (hlt)

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



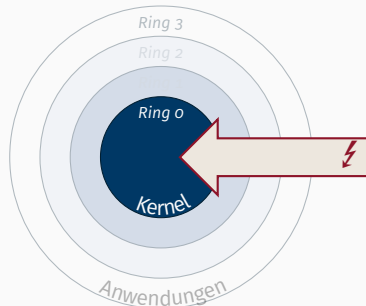
Daten Ring 3	0x20
Code Ring 3	0x18
Daten Ring 0	0x10
Code Ring 0	0x08
Null-Deskriptor	0x00

63

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

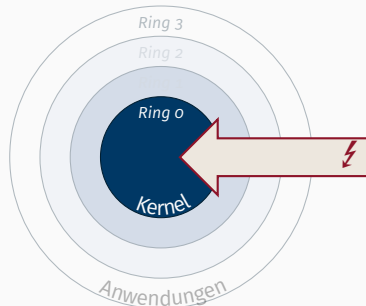
63

Kernelstack	...	bei Unterbrechung in Ring 0
rsp + 32	0000000000001000	vorheriges Datensegment
rsp + 24	rsp	vorheriger Stackpointer
rsp + 16	rflags	Status/Condition Codes
rsp + 8	0000000000000100	vorheriges Codesegment
rsp →	rip	Rücksprungadresse



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

63 0

63

The diagram illustrates the Linux kernel ring architecture as a series of concentric circles. At the center is the **Kernel**. Surrounding it are four rings, labeled from the center outwards: **Ring 0**, **Ring 1**, **Ring 2**, and **Ring 3**. The outermost layer is labeled **Anwendungen** (Applications). A large red arrow points from the **Anwendungen** layer towards the **Kernel**, indicating the flow of data or control. A small red lightning bolt symbol is located on the arrow.

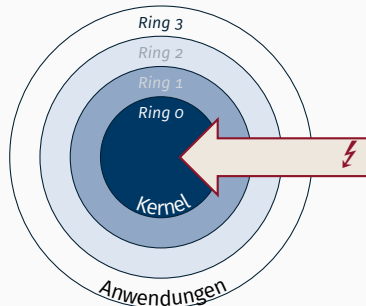
Daten Ring 3	0x20
Code Ring 3	0x18
Daten Ring 0	0x10
Code Ring 0	0x08
Null-Descriptor	0x00

63

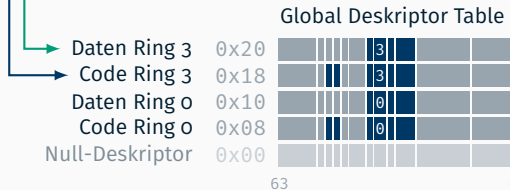
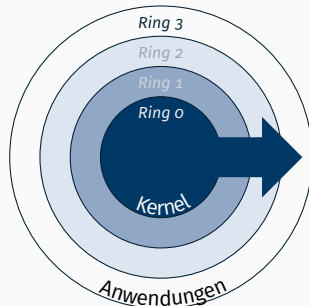
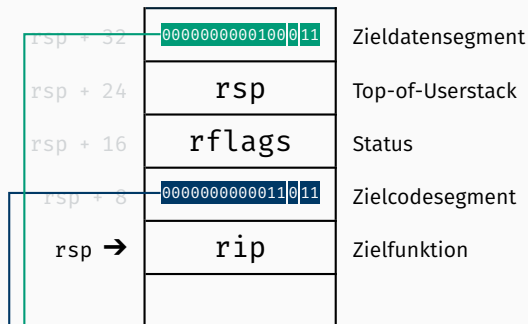
The diagram illustrates the Linux kernel's ring architecture. It consists of four concentric rings, labeled Ring 0, Ring 1, Ring 2, and Ring 3, moving from the center outwards. The innermost region is the **kernel**, and the outermost region is labeled **Anwendungen** (Applications). A large red arrow points from the applications layer towards the kernel, indicating the flow of data or control. A small red lightning bolt symbol is located on the arrow, near the applications layer.

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

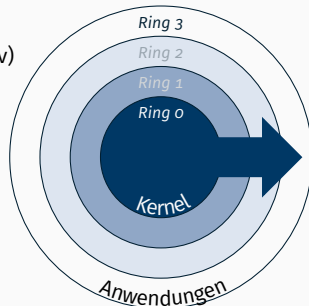
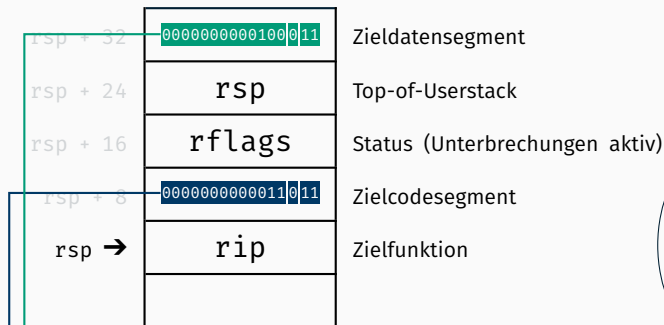
63

63

Initialer Wechsel in Ring 3 durch präparierten Kernelstack und iretq

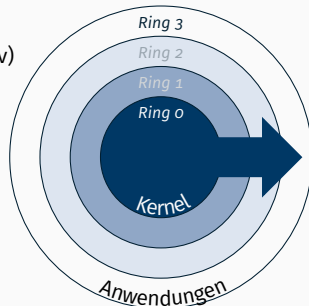
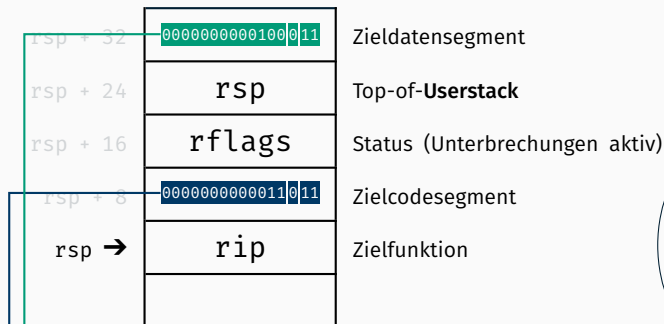


Initialer Wechsel in Ring 3 durch präparierten Kernelstack und iretq



			Global Descriptor Table																					
→	Daten Ring 3	0x20														3								
→	Code Ring 3	0x18														3								
	Daten Ring 0	0x10														0								
	Code Ring 0	0x08														0								
	Null-Descriptor	0x00																						

Initialer Wechsel in Ring 3 durch präparierten Kernelstack und iretq



			Global Descriptor Table																					
→	Daten Ring 3	0x20														3								
→	Code Ring 3	0x18														3								
	Daten Ring 0	0x10														0								
	Code Ring 0	0x08														0								
	Null-Descriptor	0x00																						

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 ("movl %%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

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

- 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

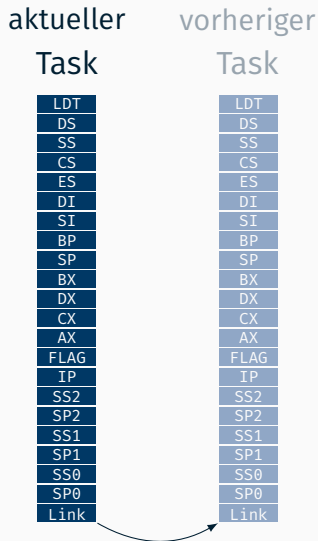
- Ab 286er (16 bit Protected Mode):
Hardware Tasks
- Der Zustand wird beim Wechsel im *Task State Segment (TSS)* gesichert
- *Task Register* zeigt (über eine Indirektion in GDT) auf aktuelles TSS
 - Laden/Ändern mittels `ltr`-Instruktion
 - Segment Selektor (in GDT) als Parameter

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

- 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

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

- 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)



- 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	

- 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

- 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

- Im Long Mode gibt es immer noch das TSS

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)	

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

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)	

- Im Long Mode gibt es immer noch das TSS, aber keine Unterstützung für Hardware Tasks
- nur noch die für moderne Betriebssysteme relevanten Features

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)	

- Im Long Mode gibt es immer noch das TSS, aber keine Unterstützung für Hardware Tasks
- nur noch die für moderne Betriebssysteme relevanten Features
 - Stackpointer für jeden (privilegierten) Ring (RSP0 für Ring 0 interessant in 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)	

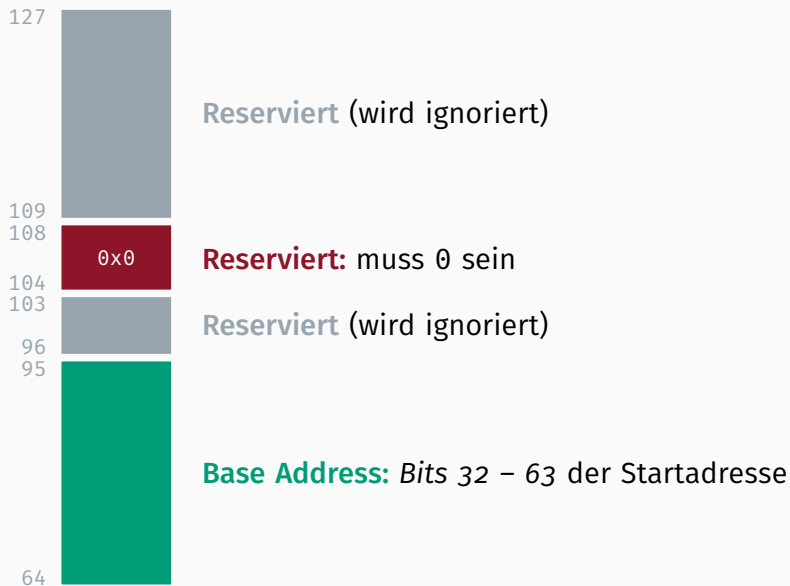
- Im Long Mode gibt es immer noch das TSS, aber keine Unterstützung für Hardware Tasks
- nur noch die für moderne Betriebssysteme relevanten Features
 - Stackpointer für jeden (privilegierten) Ring (RSP0 für Ring 0 interessant in STuBS)
 - zusätzlich noch *Interrupt Stack Table* (nicht notwendig für STuBS)

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)	

- Im Long Mode gibt es immer noch das TSS, aber keine 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)	





TSS Deskriptoreintrag in GDT

Global Descriptor Table

Descriptor	Size	Value
Daten Ring 3	0x20	3
Code Ring 3	0x18	3
Daten Ring 0	0x10	0
Code Ring 0	0x08	0
Null-Descriptor	0x00	0

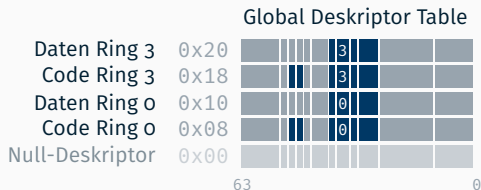
63 0

63

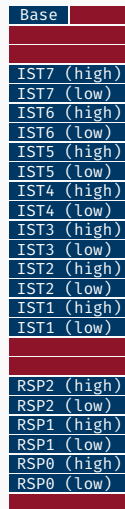
0

TSS Deskriptoreintrag in GDT

1. Speicher für TSS reservieren



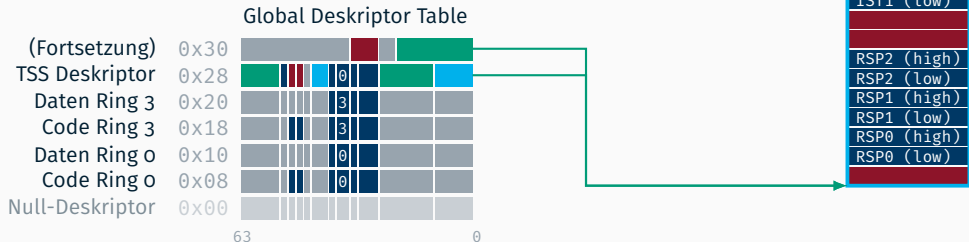
Task State Segment



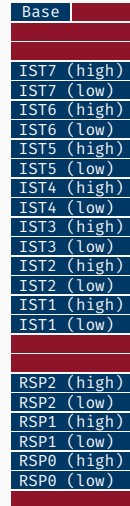
TSS Deskriptoreintrag in GDT

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

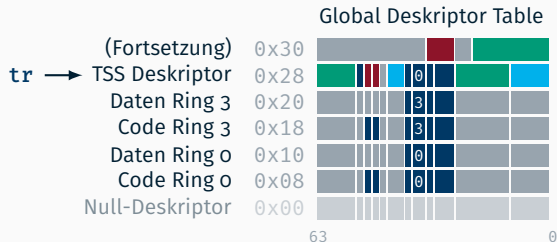
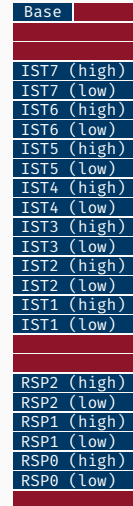
Task State Segment



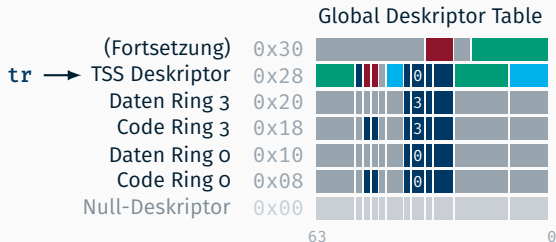
- ## Task State Segment



- ## Task State Segment



- ## Task State Segment



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

Global Deskriptor Table

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

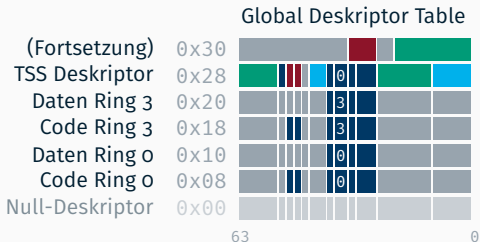
63
0

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

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



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
3. Task Register (tr) setzen
 - mittels `ltr` Instruktion
 - zeigt auf Segment Offset (hier: `0x28`)
4. `RSP0` setzen
 - auf Kernelstack des aktuellen Threads

Global Deskriptor Table

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

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

63

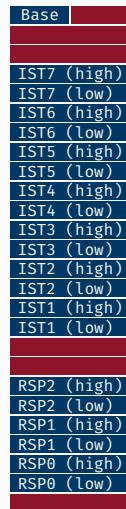
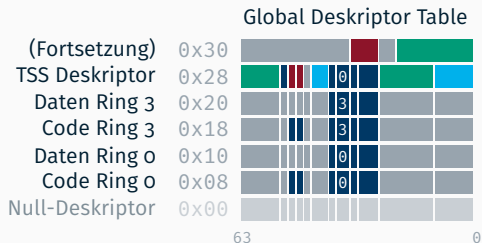
Task State Segment

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

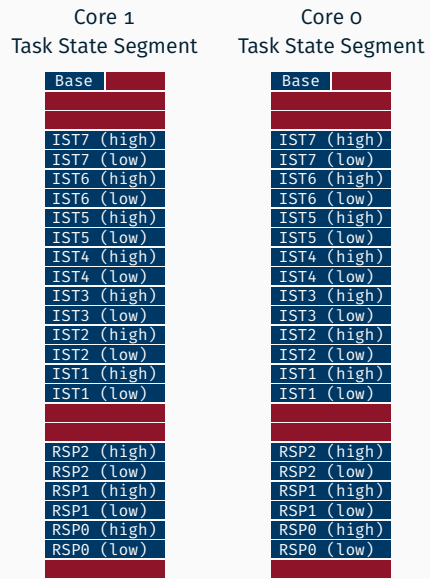
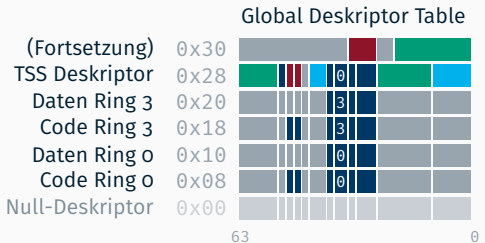
Task State Segment



TSS Deskriptoreintrag in GDT bei MPStUBSML

Im Mehrkernbetrieb hat jeder Kern

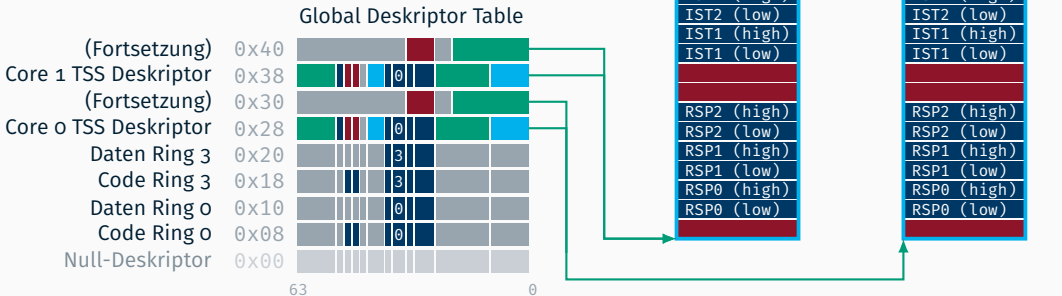
- einen eigenes TSS allokiert



TSS Deskriptoreintrag in GDT bei MPSTuBSMI

Im Mehrkernbetrieb hat jeder Kern

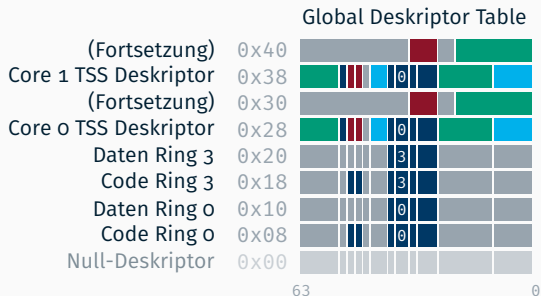
- einen eigenes TSS allokiert
- einen eigenen TSS Deskriptor Eintrag



TSS Deskriptoreintrag in GDT bei MPStUBSML

Im Mehrkernbetrieb hat jeder Kern

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



Core 1
Task State Segment

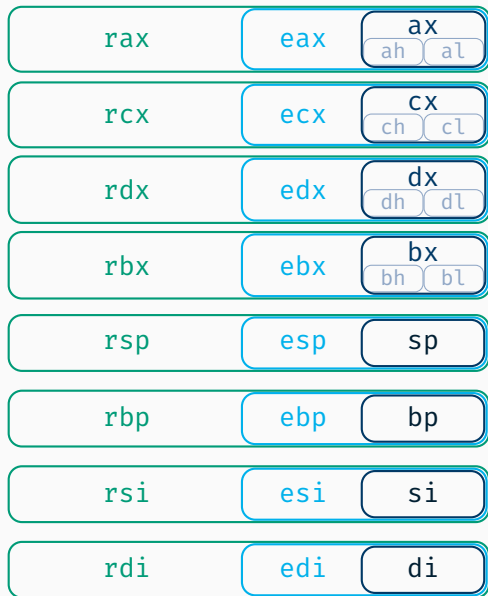
Base	
IST7 (high)	
IST7 (low)	
IST6 (high)	
IST6 (low)	
IST5 (high)	
IST5 (low)	
IST4 (high)	
IST4 (low)	
IST3 (high)	
IST3 (low)	
IST2 (high)	
IST2 (low)	
IST1 (high)	
IST1 (low)	
RSP2 (high)	
RSP2 (low)	
RSP1 (high)	
RSP1 (low)	
RSP0 (high)	
RSP0 (low)	

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)	

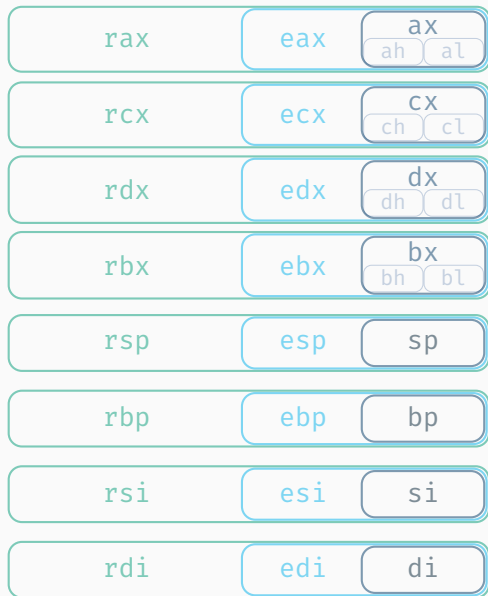
Core Local Storage (7.5 ECTS)

Long Mode Register



+ 16× SSE Register (XMM, 128 bit)

+ Kontrollregister + Debugregister



+ 16× SSE Register (XMM, 128 bit)

+ Kontrollregister + Debugregister

```
// 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
#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)

```
// 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:0xfffffffffffffffff8
add rax, 0x1
mov fs:0xfffffffffffffffff8, rax
```

```
// 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:-8
add rax, 0x1
mov fs:-8, rax
```

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

- 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)

- 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  
        };  
    }  
}
```

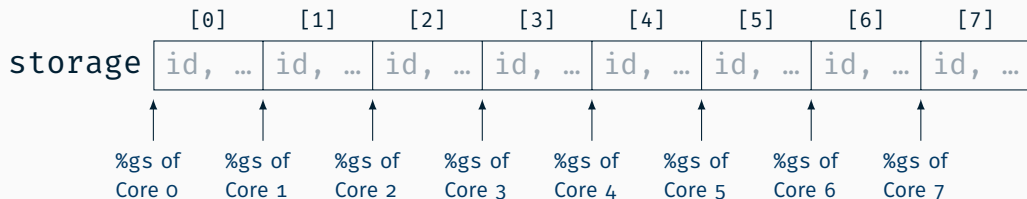
Core Local Storage

- 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)
- 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);
```

- 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);
```

- 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
```

- 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("movl %%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("movl %%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 0xfe00020)
- 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 0xfe00020)
- 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 0xfe00020)
- 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 0xfe00020)
- 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 0xf0000000)
- 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 `0xffe00020`)
- 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 0xf0000000)
- 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?