Übung zu Betriebssystemtechnik

Aufgabe 1: Privilegientrennung

23. April 2025

Dustin Nguyen, Maximilian Ott & Phillip Raffeck

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

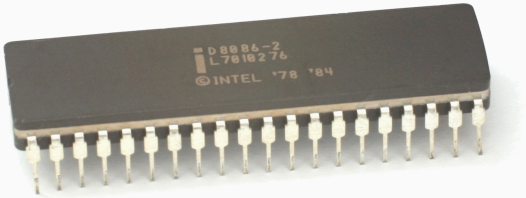
Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

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

Quelle: Wikipedia

Real Mode



Quelle: Wikipedia

8086 16 bit, 1 MB Speicher adressierbar

- Segementierung mit 20 bit Adressbus
- 14 Register

Real Mode Register

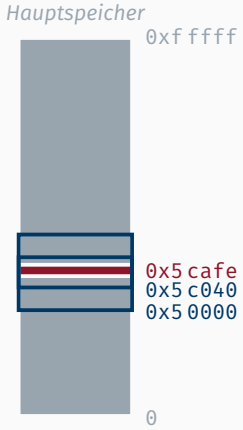
Real Mode Segmentierung

Zugriff auf Zieldaten in `0x5cafe`

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

Alternativ:

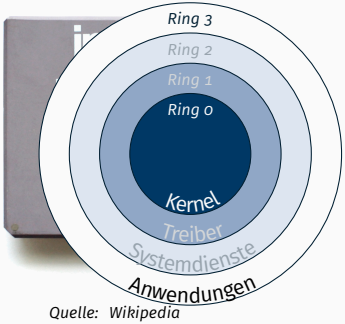
1. Setze **`ds`** auf `0x5c04`
   (Bereich `0x5c040` – `0x6c039`)
2. Zugriff über **`ds:0xabe`**



Protected Mode

Name aufgrund der Schutzringe (*Privilege Level*)

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)*

Segmentierung im Protected Mode

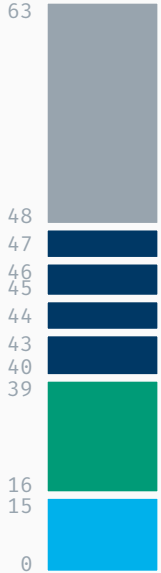
- Einführung von Deskriptortabellen

  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



Reserviert

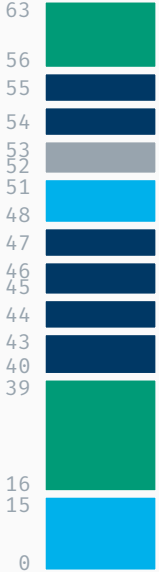
Present: Eintrag aktiv (`1`) oder inaktiv (`0`)
Descriptor Privilege Level: Erlaubter Ring
Descriptor type: System (`0`) oder Code/Daten (`1`)
Type: Bedeutung abhängig vom Deskriptortyp

Base Address: Startadresse (24 bit)

Segment Limit: Segmentgröße (16 bit)

32 bit Segmentdeskriptoreintrag

ISDMv3 3.4.5



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

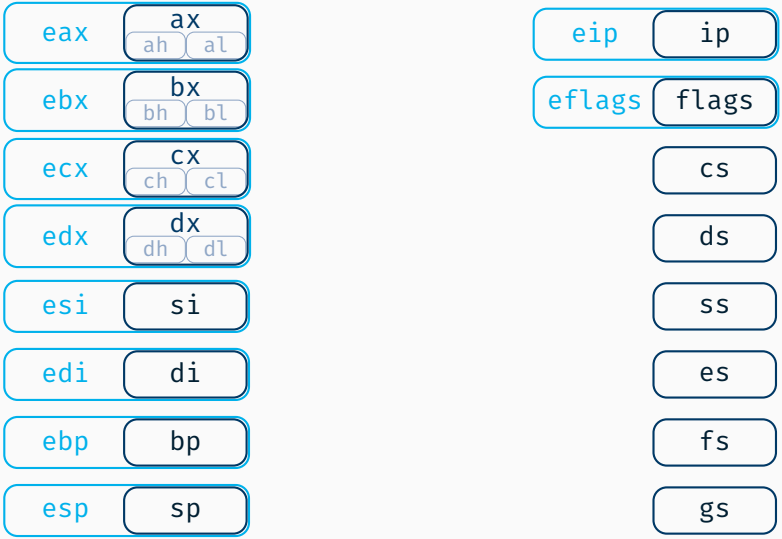
Granularity: Byte (`0`) oder 4 KB-Block (`1`)
Operation Size: 16 bit (`0`) oder 32 bit (`1`)
Reserviert
Segment Limit (high): *Oberer* Teil der Segmentgröße
Present: Eintrag aktiv (`1`) oder inaktiv (`0`)
Descriptor Privilege Level: Erlaubter Ring
Descriptor type: System (`0`) oder Code/Daten (`1`)
Type: Bedeutung abhängig vom Deskriptortyp

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

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

32 bit Protected Mode Register

ISDMv2 3.4.1



Protected Mode Segmentierung (32 bit) umgehen

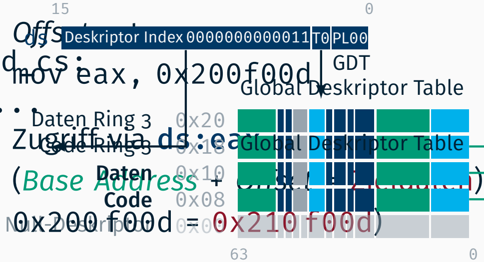
Flaches Speichermodell in startup.asm

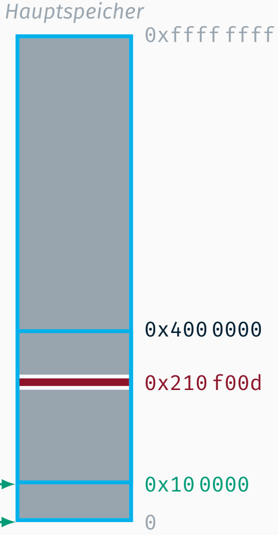
```
mov ax, 0x10
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
mov ss, ax
jmp 0x8:load_cs
load_cs:
...
```

Zugriffsbeispiel: Zieladresse 0x210 f00d

→ Zugriff über 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
  ```
- *Offset* in eax
  ```
  mov eax, 0x200f00d
  ```
- Zugriff via ds:eax
  (*Base Address* + *Offset* = *Zieladresse*) (0x10 0000 + 0x200 f00d = 0x210 f00d)





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 über 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
```

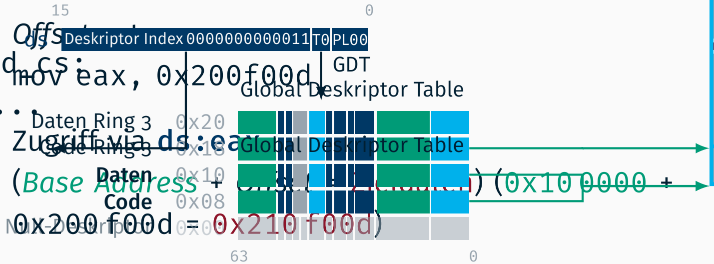
- *Offset* in eax

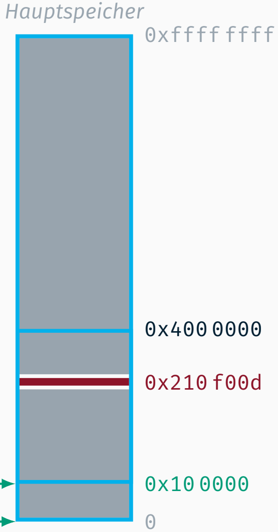
```
mov eax, 0x200f00d
```

- Zugriff via ds:eax

(*Base Address* + *Offset* = Zieladresse) (0x10 0000 + 0x200 f00d = 0x210 f00d)

```
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)

Flaches Speichermodell → `longmode.asm`:

```
lgdt [gdt.long_mode_pointer]
jmp start
```

GDT-Register **gdtr**

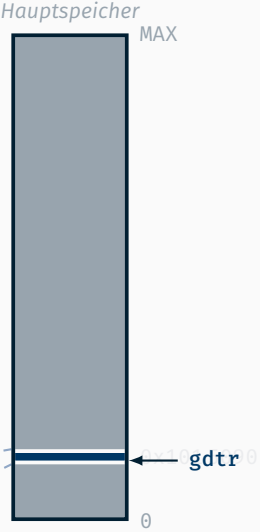
Quelle: Wikipedia

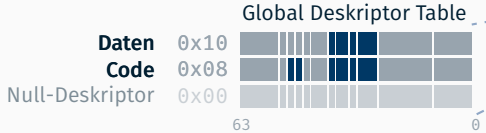
Basis: Startadresse der GDT (bei uns GDT:

Limit: Länge der GDT in Bytes

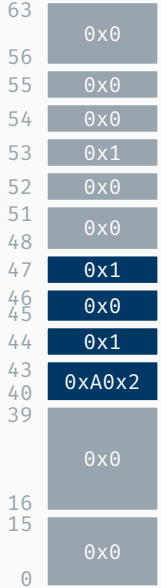
Instruktionen:

- **lgdt** in Register laden
- **sgdt** aus Register lesen





64 bit GDT Datensegmentdeskriptoreintrag ISDMv3 3.4.5.1 & AAPMv2 4.8.2



Base Address (high): Bits 24 – 31 der Startadresse Base Address (high)

Granularity: Byte (0) oder 4 KB-Block (1) Granularity

Operation Size: Longmode / 16 bit (0) oder 32 bit (1) Operation Size

Longmode: Kompatibilitäts- (0) oder 64 bit-Modus (1) Longmode

Available: zur freien Verwendung

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

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

Descriptor Privilege Level: Erlaubter Ring

Descriptor type: System (0) oder Code/Daten (1)

Type: Bedeutung abhängig vom Deskriptortyp Code (Bit 43 = 1) lesbar (Bit 41 = 1) Daten (Bit 43 = 0) schreibbar (Bit 41 = 1)

Base Address (low): Bits 0 – 23 der Startadresse Base Address (low)

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

64 bit GDT (Flaches Speichermodell)

Flaches Speichermodell → `longmode.asm`:

```
lgdt [gdt.long_mode_pointer]
jmp start
```

GDT-Register **gdtr**

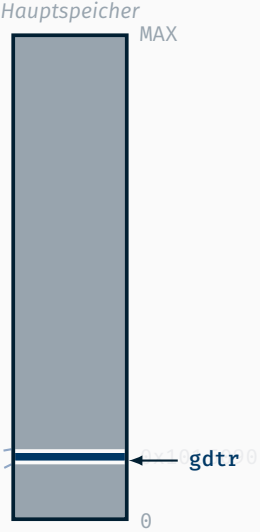
Quelle: Wikipedia

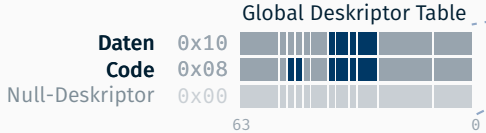
Basis: Startadresse der GDT (bei uns GDT:

Limit: Länge der GDT in Bytes

Instruktionen:

- **lgdt** in Register laden
- **sgdt** aus Register lesen





64 bit GDT (Privilegienebenen und Unterbrechungen wechseln)

ISDMv3 6.14

Ausführung von **priviligierten Befehlen** wie

- Laden von GDT/LDT/IDT (lgdt/lldt/lidt) und Taskregister (ltr)
- Ändern (mo... 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)

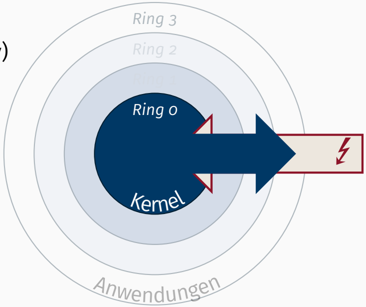
in Ring 3 verboten (führt zu **General Protection Fault**)

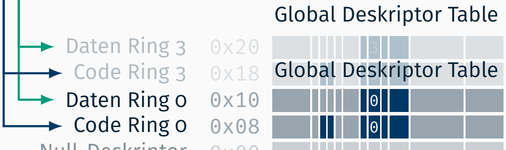
(bei Unterbrechungen und I/O Port jedoch abhängig von I/O Privilege Level)

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

Kernelstack ***bei Unterbrechung in Ring 0/3***

	...	
rsp + 32	ss: 0000000000001000000000000100 0 0011	Datensegment
rsp + 24	rsp	Top-of-User-Stackpointer
rsp + 16	rflags	Status (Unterbrechungen aktiv)
rsp + 8	cs: 0000000000001000000000000011 0 0011	Codesegment
rsp →	rip	Rücksprungadresse





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?

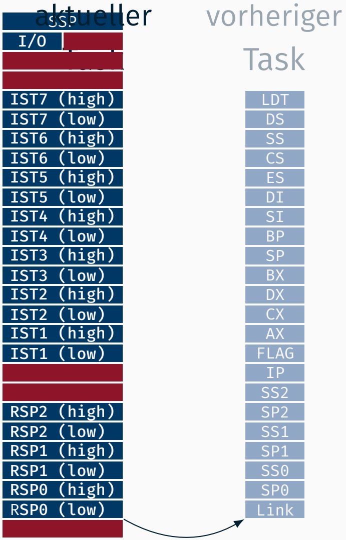
Task State Segment

Task State Segment (64 bit)

- Ab 386er (auch für 32-bit Protected Mode) Hardwareunterstützung für Hardware Tasks
- Der Zustand wird beim Wechsel im Task State Segment (TSS) gesichert
→ einfachere Betriebssysteme, Task Switching 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 Betriebssystemen mit Paging])
- Nicht portabel (auf andere Architekturen)
→ Verwendung von softwarebasierten Kontextwechsel in modernen BS

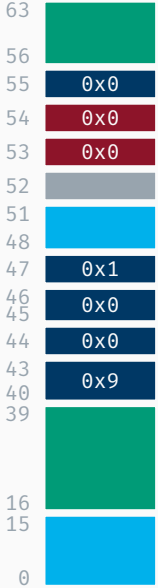
- Task-Register zeigt (über Deskriptor in GDT) auf aktuelles TSS
  - laden/ändern mittels ltr Instruktion (nicht notwendig für StuBS)
  - Segment Selektor in GDT als Parameter
- Steuerung von I/O Berechtigungen
- Stackpointer für jeden (privilegierten) Ring (auch nicht notwendig für StuBS)
- Verlinkung bei (Hardware Task) Umschaltung (via call / jmp zu Task oder Interrupt)

- Im Long Mode gibt es weiterhin das TSS, aber keine Hardware Tasks
  - Steuerung von I/O Berechtigungen
  - Stackpointer für jeden (privilegierten) Ring (RSP0 für Ring 0 interessant in StuBS)
  - zusätzlich noch *Interrupt Stack Table*
- ein TSS pro Kern (unabhängig von Tasks)
  - für Stackpointer und ggf. I/O Berechtigung



64 bit TSS Deskriptoreintrag

ISDMv3 7.2.3



Base Address (high): *Bits 24 – 31* der Startadresse

Granularity: Byte (0) oder 4 KB-Block (1)

Reserviert

Reserviert

Available: zur freien Verwendung

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

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

Descriptor Privilege Level: Erlaubter Ring (bei uns: 0)

Descriptor type: System (0) oder Code/Daten (1)

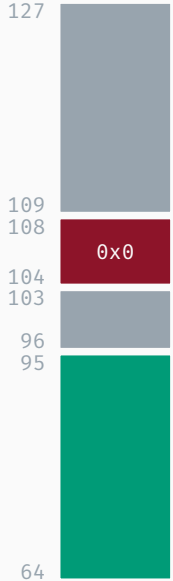
Type 0x9 für TSS; *Busy Flag* (Bit 41) wird gesetzt wenn aktiv

Base Address (low): Bits 0 – 23 der Startadresse

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

64 bit TSS Deskriptoreintrag (Fortsetzung)

ISDMv3 7.2.3



Reserviert (wird ignoriert)

Reserviert: muss `0` sein

Reserviert (wird ignoriert)

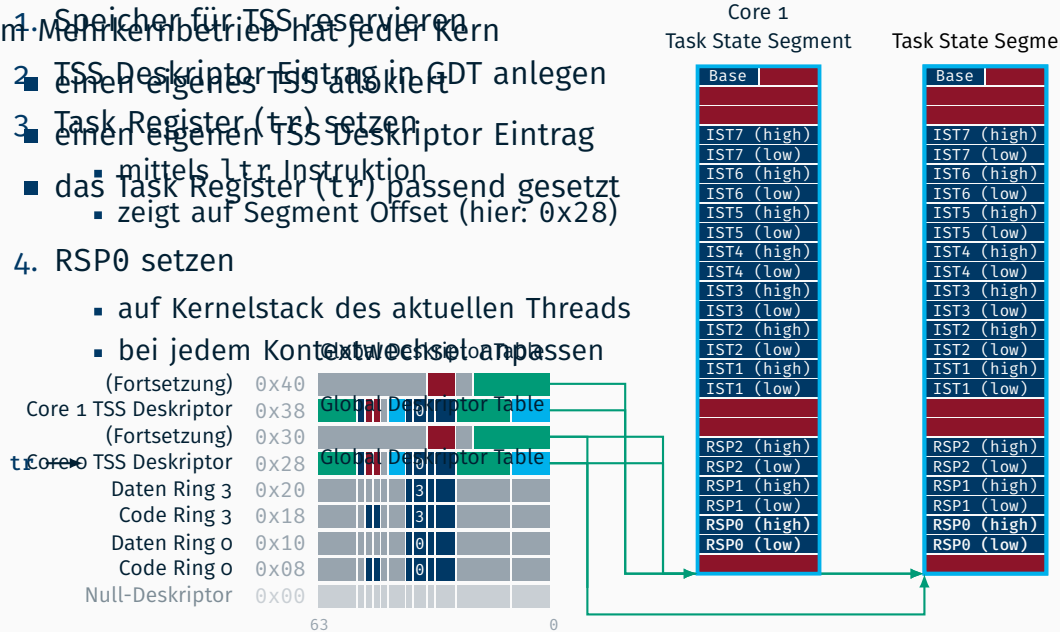
Base Address: *Bits 32 – 63* der Startadresse

TSS Deskriptoreintrag in GDT bei MPStuBSmi

Im Mehrkernbetrieb hat jeder Kern

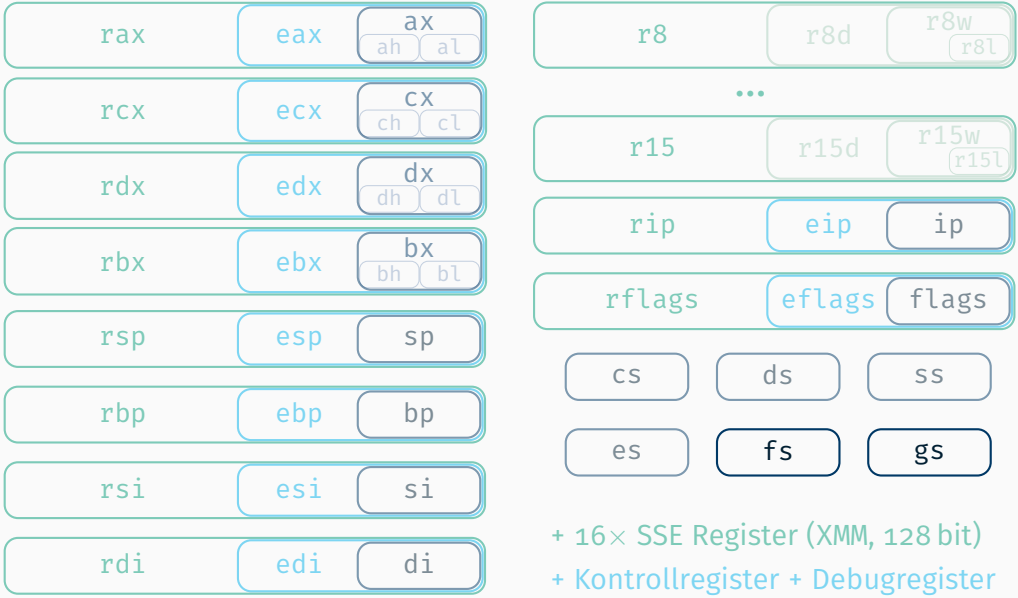
- einen eigenen TSS allokiert
- einen eigenen TSS Deskriptor Eintrag
- das Task Register (`tr`) passend gesetzt
  - zeigt auf Segment Offset (hier: `0x28`)

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



Core Local Storage (7.5 ECTS)

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;
}
```

```
$ 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)

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)
  - 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
- 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`

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

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?