Übung zu Betriebssystemtechnik

Aufgabe 2: Systemaufrufe

04. Mai 2023

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

Nachtrag: Sprung in Ring 3 via GCC Inline ASM

```
void switchToUsermode(void *stackpointer, void *kickoff, void * kickoff_parameter) {
  const unsigned ring = 3;      // Ring 3 = User Mode
  asm volatile ("cli\n\t"       // disable Interrupts
                // Prepare Stack
                "push %0\n\t"   // User Data Segment Selector
                "push %1\n\t"   // User Stack Pointer
                "push %2\n\t"   // Flags (previous copy)
                "push %3\n\t"   // User Code Segment Selector
                "push %4\n\t"   // Target Function
                // Swap to user gs
                "swapgs\n\t"
                // Return from (fake) "interrupt" to switch ring
                "iretq\n\t"
                :
                : "n"((GDT::SEGMENT_USER_DATA * sizeof(GDT::SegmentDescriptor)) | ring)
                , "r"(stackpointer)
                , "n"(Core::Interrupt::FLAG_ENABLE | 0x2) // bit 1 should be 1
                , "n"((GDT::SEGMENT_USER_CODE * sizeof(GDT::SegmentDescriptor)) | ring)
                , "r"(kickoff)
                , "D"(kickoff_parameter)   // put in RDI (first parameter)
                : "memory");
}
```

Nachtrag: Segmentselektor

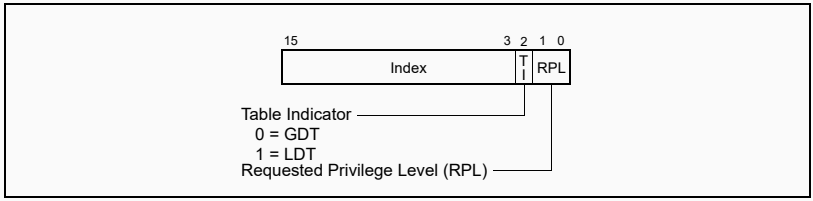


Figure 3-6. Segment Selector

Wiederholung: Aufrufkonvention

Kontextsicherung gemäß Konvention

```
void baz(){
  ...

  func();

  ...
}
```

```
void func(){

  ...

  return;
}
```

Kontextsicherung gemäß Konvention

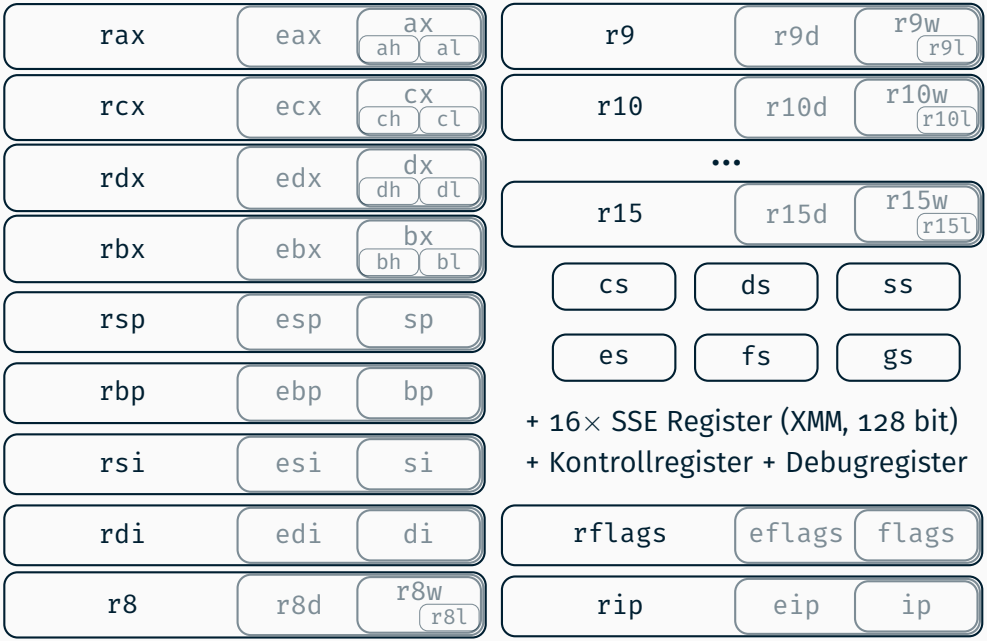
```
void baz(){
  ...

  // flüchtige Register
  // sichern

  func();

  // flüchtige Register
  // wiederherstellen

  ...
}
```

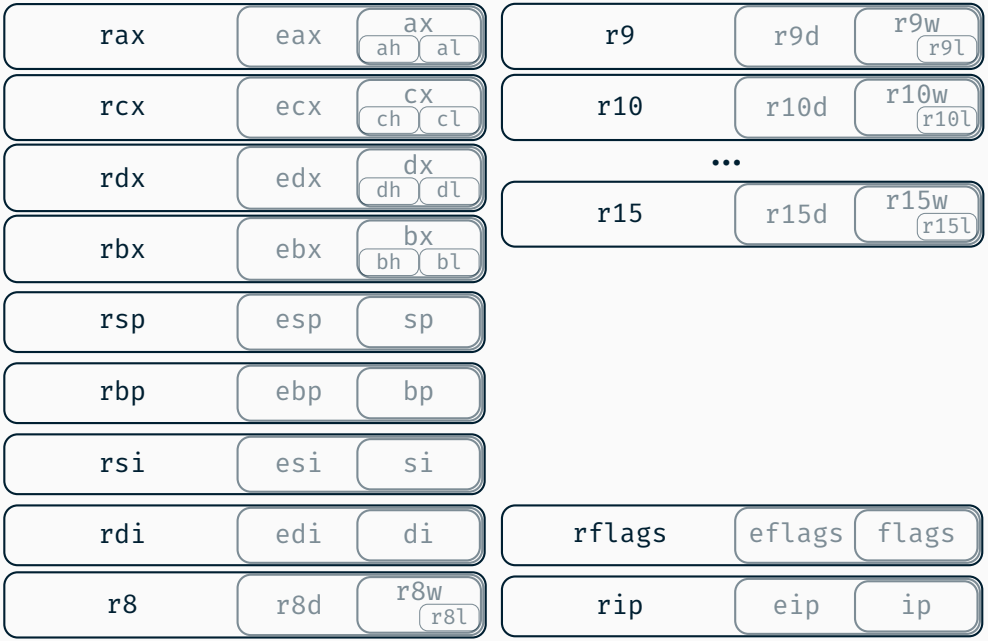
```
void func(){

  ...

  return;
}
```

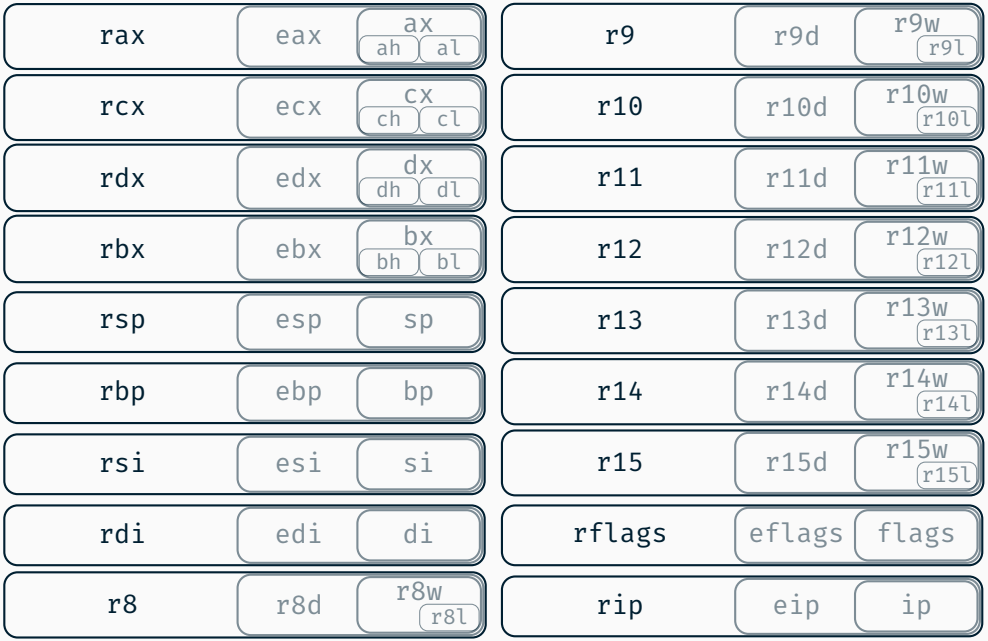
Kontextsicherung gemäß Konvention

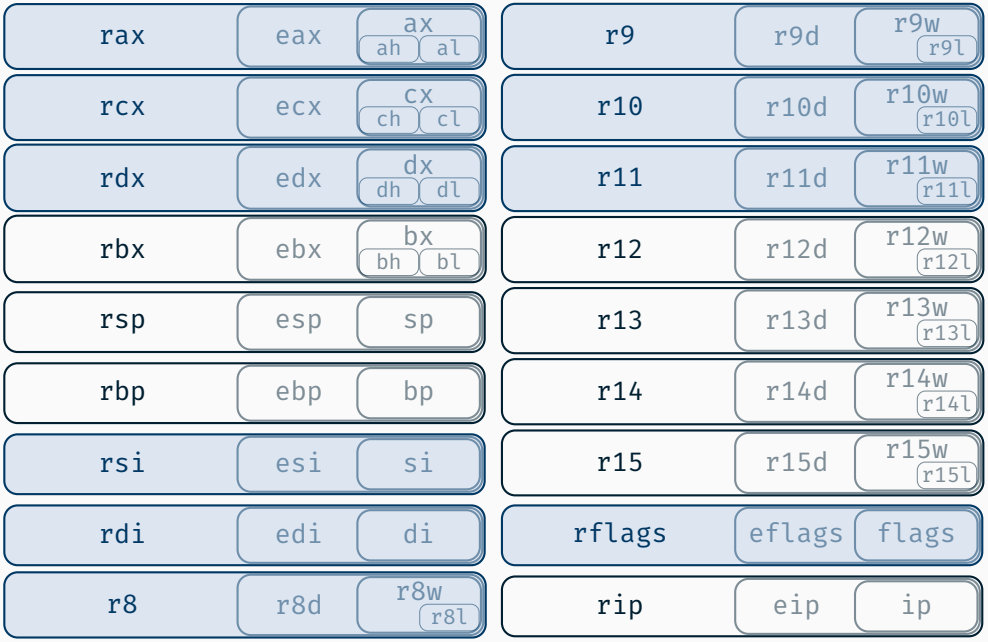
```
void baz(){
  ...

  // flüchtige Register
  // sichern

  func();

  // flüchtige Register
  // wiederherstellen

  ...
}
```

```
void func(){
  // nicht-flüchtige
  // Register
  // sichern

  ...

  // nicht-flüchtige
  // Register
  // wiederherstellen

  return;
}
```

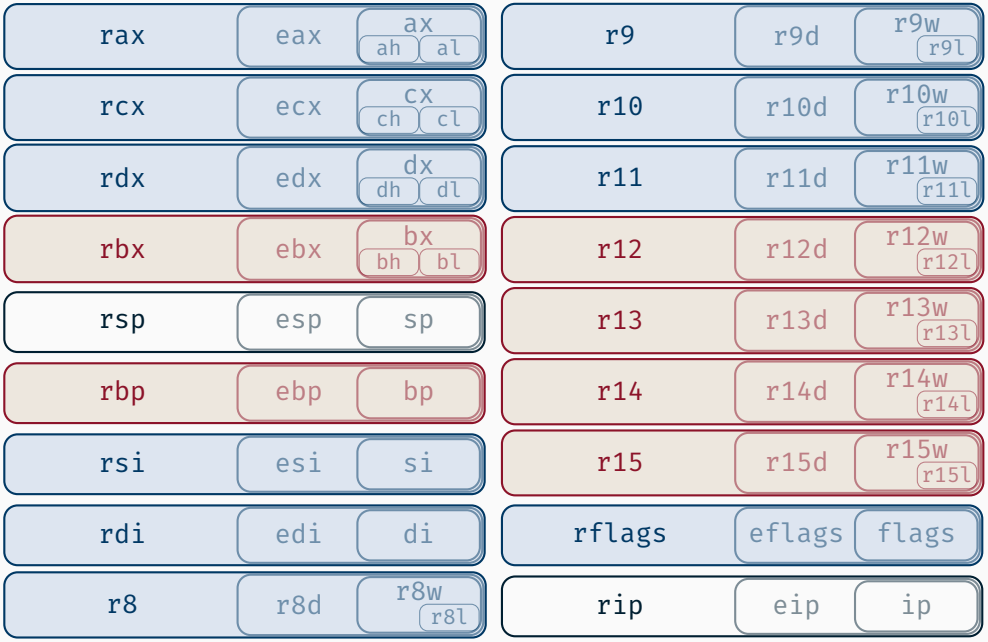

rax
eax
ax
ah
al
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
rdi
edi
di
r8
r8d
r8w
r8l
r9
r9d
r9w
r9l
r10
r10d
r10w
r10l
...
r15
r15d
r15w
r15l
cs
ds
ss
es
fs
gs
+ 16× SSE Register (XMM, 128 bit)
+ Kontrollregister + Debugregister
rflags
eflags
flags
rip
eip
ip


rax
eax
ax
ah
al
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
rdi
edi
di
r8
r8d
r8w
r8l
r9
r9d
r9w
r9l
r10
r10d
r10w
r10l
...
r15
r15d
r15w
r15l
rflags
eflags
flags
rip
eip
ip


rax
eax
ax
ah
al
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
rdi
edi
di
r8
r8d
r8w
r8l
r9
r9d
r9w
r9l
r10
r10d
r10w
r10l
r11
r11d
r11w
r11l
r12
r12d
r12w
r12l
r13
r13d
r13w
r13l
r14
r14d
r14w
r14l
r15
r15d
r15w
r15l
rflags
eflags
flags
rip
eip
ip

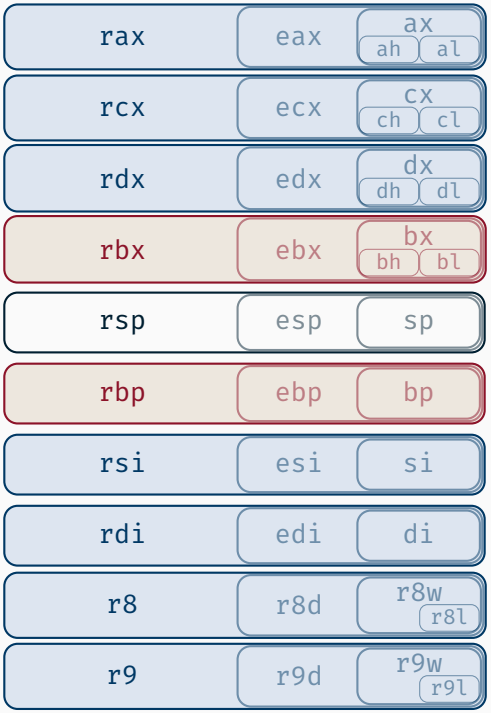


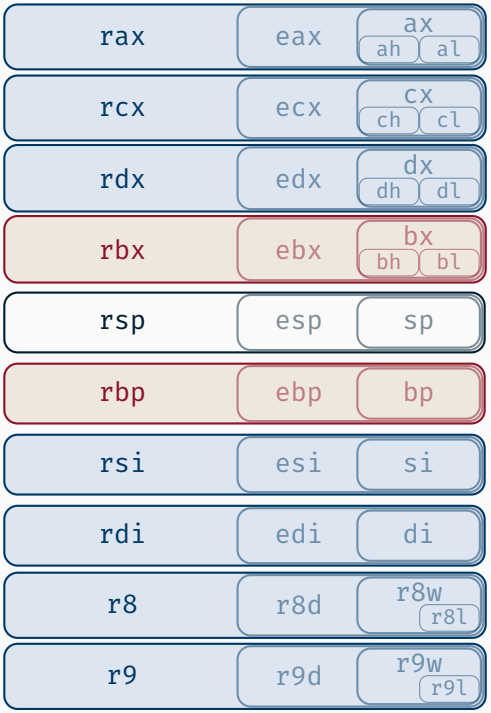
flüchtige (*scratch* / *caller-save*) Register



flüchtige (*scratch* / *caller-save*) Register nicht-flüchtig (*non-scratch* / *callee-save*)

Wieso nicht gleich die **flüchtigen Register** beim Funktionsaufruf nutzen?

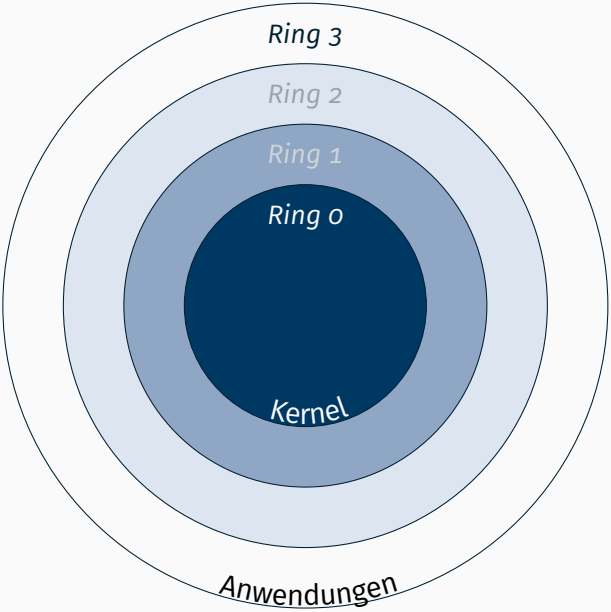

rax
eax
ax
ah
al
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
rdi
edi
di
r8
r8d
r8w
r8l
r9
r9d
r9w
r9l


rax
eax
ax
ah
al
Rückgabewert
rcx
ecx
cx
ch
cl
4. Parameter
rdx
edx
dx
dh
dl
3. Parameter
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
2. Parameter
rdi
edi
di
1. Parameter
r8
r8d
r8w
r8l
5. Parameter
r9
r9d
r9w
r9l
6. Parameter

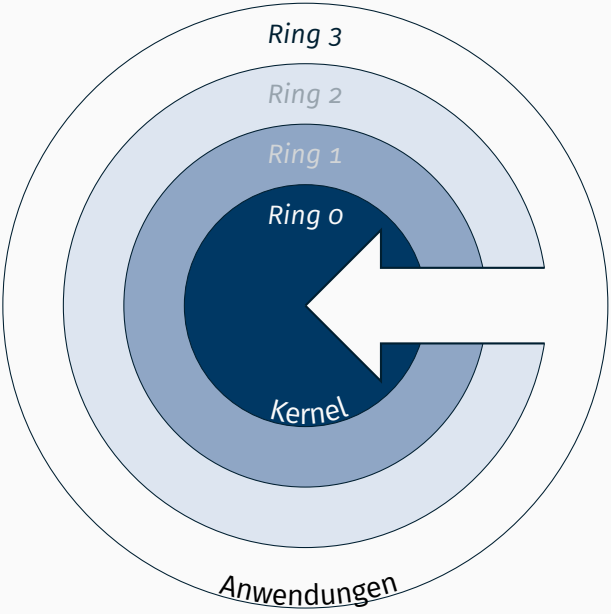
Aufgabe 2

Anwendungen (Ring 3) sollen von StuBSmi (Ring 0) bereitgestellte Funktionalitäten nutzen können

Anwendung muss Kontrolle an Kernel übergeben

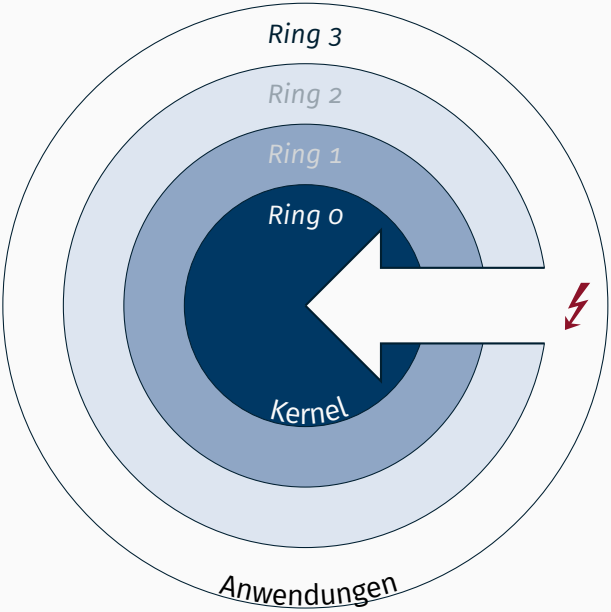


Anwendung muss Kontrolle an Kernel übergeben



Anwendung muss Kontrolle an Kernel übergeben

➔ durch Auslösen eines Softwareinterrupts



Interruptbasierte Systemaufrufe

Bitte beachten

Zur Vereinfachung wird bei der Übung mit **Unterbrechungen** / **Interrupts** analog zum **Intel Manual** eine Obermenge bezeichnet, welche

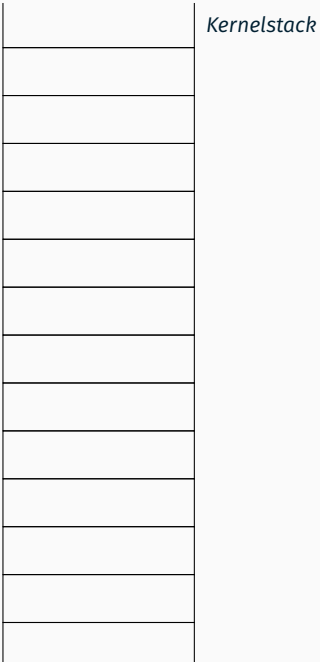
- asynchrone Unterbrechungen (z.B. durch Geräte),
- synchone Ausnahmen (durch Exceptions) sowie den
- Unterbrechungsbefehl (durch Softwareinstruktion `int`)

einschließt.

In der Vorlesung (und dem Glossar) findet jedoch eine genauer Unterscheidung statt.

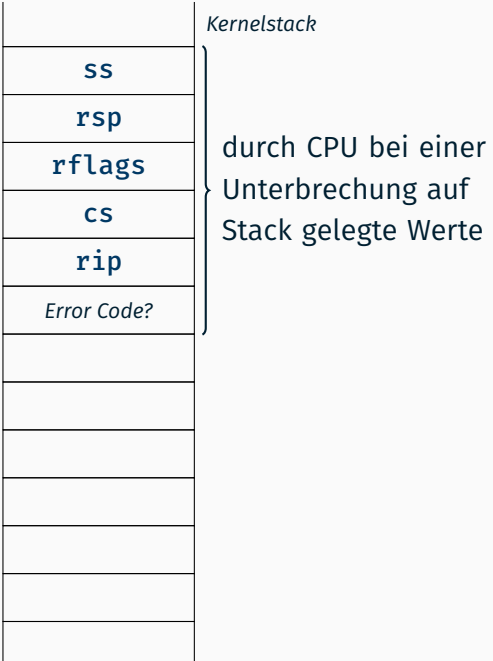
Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack



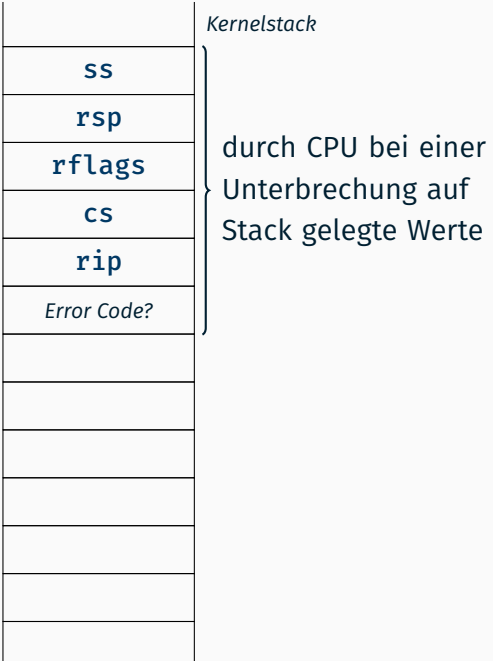
Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack



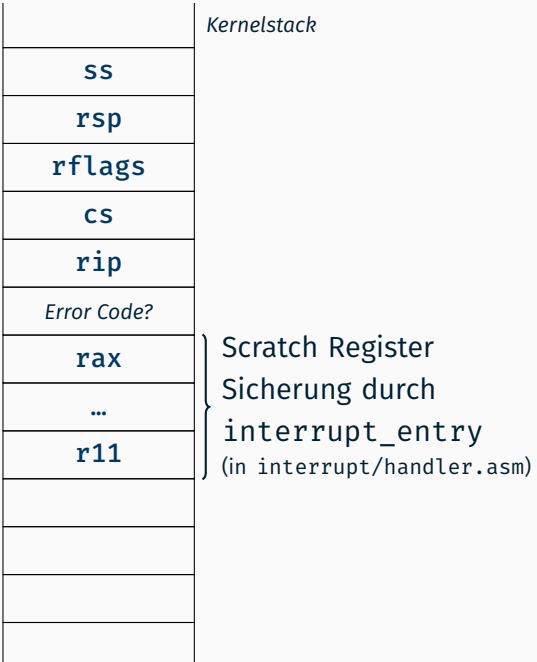
Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack



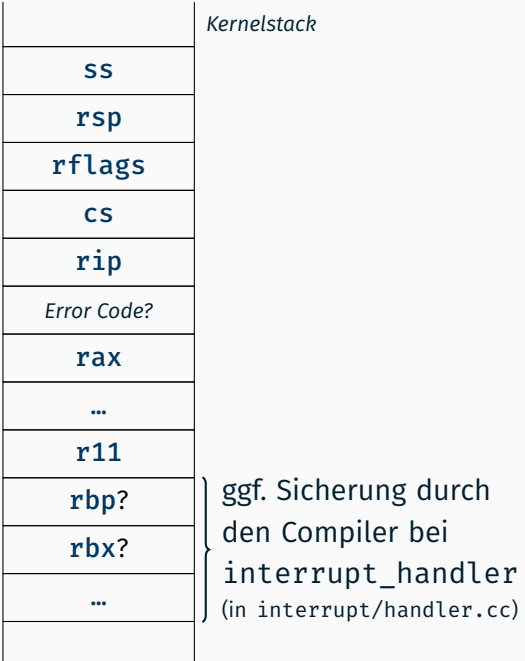
Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register



Registersicherung bei Unterbrechungen

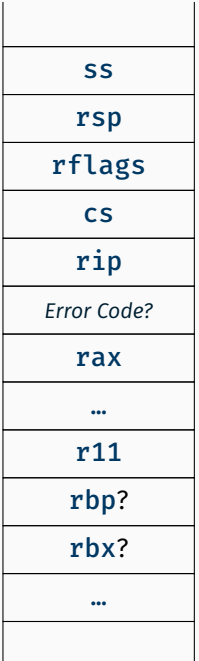
- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register



Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register

  Weitere Details siehe BS Übungsvideos
  - *Interrupts und Traps auf x86*
  - *Aufrufkonvention*

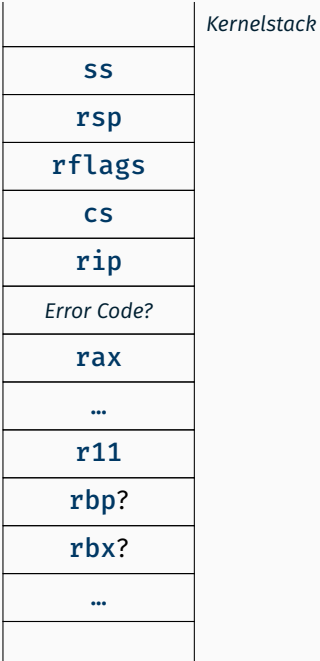


Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register

  Weitere Details siehe BS Übungsvideos
  - *Interrupts und Traps auf x86*
  - *Aufrufkonvention*

➔ Bei synchronen Unterbrechungen (Systemaufruf) aber gar nicht nötig

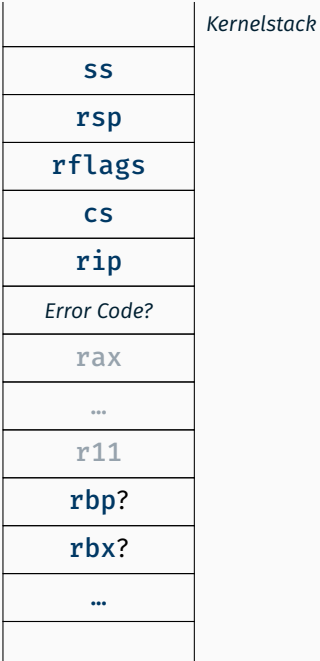


Registersicherung bei Unterbrechungen

- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register

  Weitere Details siehe BS Übungsvideos
  - *Interrupts und Traps auf x86*
  - *Aufrufkonvention*

➔ Bei synchronen Unterbrechungen (Systemaufruf) aber gar nicht nötig
  - Register direkt für die Parameterübergabe verwenden (ohne Stack)

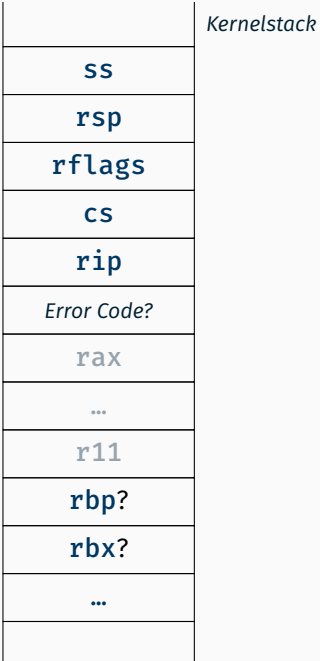


Registersicherung bei Unterbrechungen

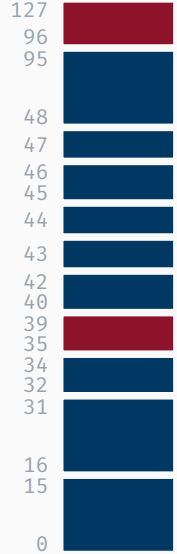
- CPU sichert bei einem Interrupt wichtige Register auf dem Stack
- Bei einer (ggf. asynchronen) Unterbrechung sichert unsere Behandlung in **StuBS** alle Register

  Weitere Details siehe BS Übungsvideos
  - *Interrupts und Traps auf x86*
  - *Aufrufkonvention*

→ Bei synchronen Unterbrechungen (Systemaufruf) aber gar nicht nötig
  - Register direkt für die Parameterübergabe verwenden (ohne Stack)
  - eigene Systemaufrufbehandlung (→ `syscall_entry`) sinnvoll



Interrupt Deskriptor



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsadresse für die Interruptbehandlung

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

Descriptor Privilege Level

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32/64-bit (`1`)

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

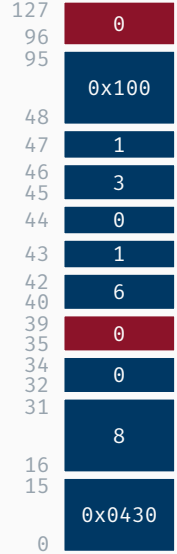
Unused – muss `0` sein

Interrupt Stack Table

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

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

Interrupt Deskriptor für `0x80` / Syscall



Unused – muss `0` sein

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

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

Descriptor Privilege Level: muss nun Ring 3 sein

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32/64-bit (`1`)

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

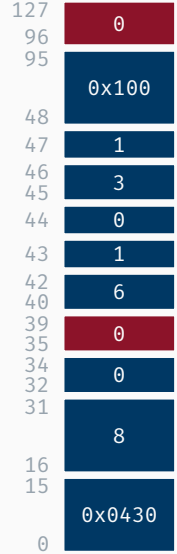
Unused – muss `0` sein

Interrupt Stack Table

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

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

Interrupt Deskriptor für `0x80` / Syscall



Unused – muss `0` sein

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

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

Descriptor Privilege Level: muss nun Ring 3 sein

Storage Segment: `0` für Interrupt und Traps

Mode: 16-bit (`0`) oder 32/64-bit (`1`)

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

Unused – muss `0` sein

Interrupt Stack Table

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

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

```
IDT::handle(0x80, syscall_entry, GATE_INT, GATE_SIZE_32, DPL_USER);
```

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)
  - zusätzliche Operationen zum Stopfen von Informationslecks sind OK

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)
  - zusätzliche Operationen zum Stopfen von Informationslecks sind OK
- Synchronisierung der Systemaufrufe im Kernel

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)
  - zusätzliche Operationen zum Stopfen von Informationslecks sind OK
- Synchronisierung der Systemaufrufe im Kernel
  - → Ausführung auf der Epilogebene

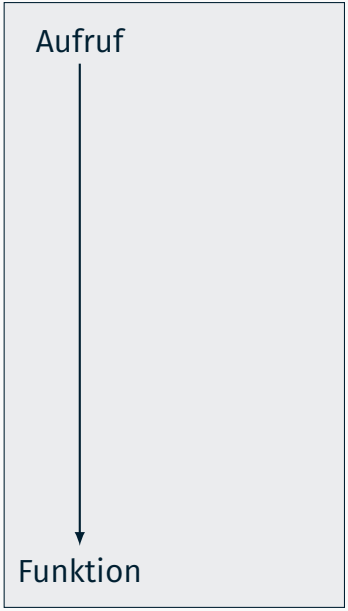
Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)
  - zusätzliche Operationen zum Stopfen von Informationslecks sind OK
- Synchronisierung der Systemaufrufe im Kernel
  - → Ausführung auf der Epilogebene
- Möglichst modulares und flexibles Design

Anforderungen an unsere Systemaufrufe

- Unterstützung von bis zu fünf Parametern
  - → nach System V ABI werden die ersten sechs Parameter in (general purpose) Register übergeben (`rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`)
- Keine Beschränkung der Anzahl der möglichen unterschiedlichen Systemaufrufe in **StuBSmi** durch die Interruptvektoren
  - Unterstützung für mehr als 256 unterschiedliche Syscalls
  - → extra Parameter für Systemaufrufnummer
- Vermeidung unnötiger Kopieroperationen (insb. auf Stack)
  - zusätzliche Operationen zum Stopfen von Informationslecks sind OK
- Synchronisierung der Systemaufrufe im Kernel
  - → Ausführung auf der Epilogebene
- Möglichst modulares und flexibles Design
  - → Vermeidung von Codeduplikation

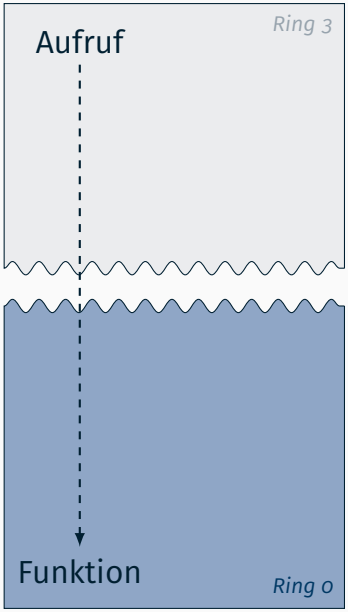
Ablauf bei Funktionsaufruf `foo`




```
// Aufruf
int r = foo(0x42);
```

```
int foo(int bar) {
    // ...
}
```

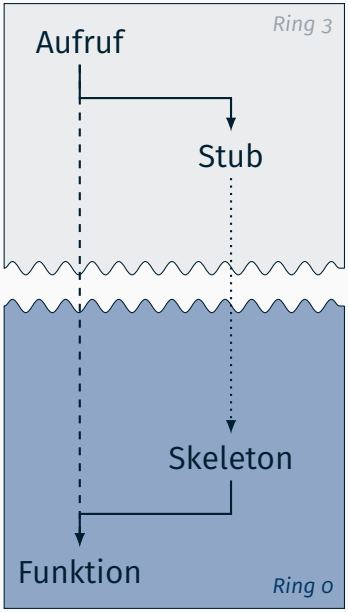
Ablauf bei Systemaufruf `foo`




```
// Aufruf
int r = foo(0x42);
```

```
int foo(int bar) {
    // ...
}
```

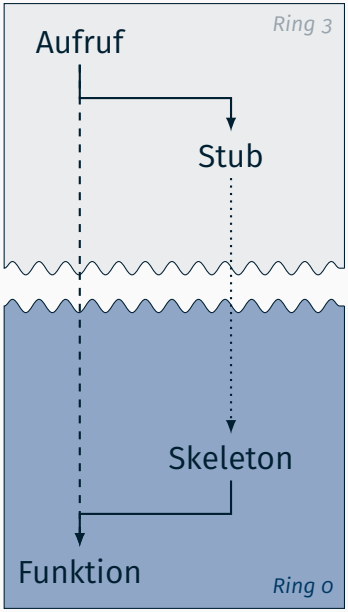
Ablauf bei Systemaufruf `foo`




```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

```
int Skeleton::foo(int bar) {
    // ...
}
```

Ablauf bei Systemaufruf `foo`

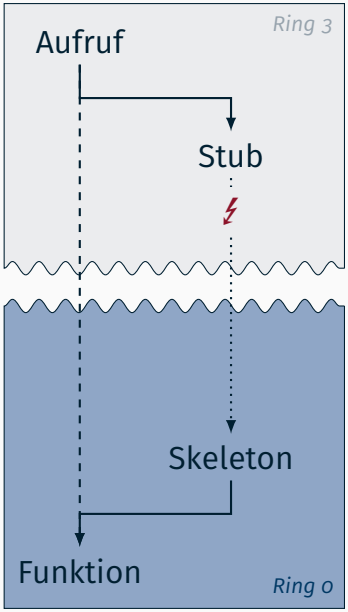



```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

1. *Stub* setzt Systemaufrufnummer

```
int Skeleton::foo(int bar) {
    // ...
}
```

Ablauf bei Systemaufruf `foo`

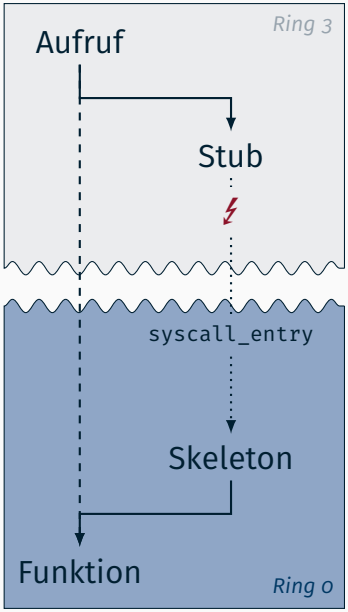



```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

1. *Stub* setzt Systemaufrufnummer
2. *Stub* löst Softwareinterrupt aus

```
int Skeleton::foo(int bar) {
    // ...
}
```

Ablauf bei Systemaufruf `foo`

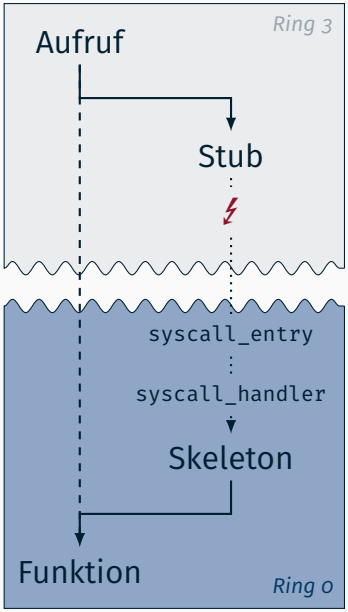



```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

1. *Stub* setzt Systemaufrufnummer
2. *Stub* löst Softwareinterrupt aus
3. Einsprung auf Ring 0 in `syscall_entry`

```
int Skeleton::foo(int bar) {
    // ...
}
```

Ablauf bei Systemaufruf `foo`

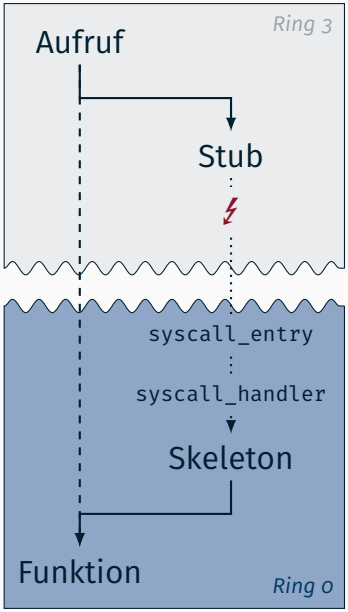



```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

1. *Stub* setzt Systemaufrufnummer
2. *Stub* löst Softwareinterrupt aus
3. Einsprung auf Ring 0 in `syscall_entry`
4. Aufruf der Hochsprachenbehandlung `syscall_handler`

```
int Skeleton::foo(int bar) {
    // ...
}
```

Ablauf bei Systemaufruf `foo`




```
// Aufruf in Userspace App
int r = sys_foo(0x42);
```

1. *Stub* setzt Systemaufrufnummer
2. *Stub* löst Softwareinterrupt aus
3. Einsprung auf Ring 0 in `syscall_entry`
4. Aufruf der Hochsprachenbehandlung `syscall_handler`
5. Auswahl des zugehörigen *Skeleton* mittels `switch case` auf Systemaufrufnummer

```
int Skeleton::foo(int bar) {
    // ...
}
```

Zu implementierende Systemaufrufe

```
size_t write(int fd, const void *buf, size_t len);
size_t read(int fd, void *buf, size_t len);
void sleep(int ms);
int sem_init(int semid, int value);
void sem_destroy(int semid);
void sem_wait(int semid);
void sem_signal(int semid);
```

mit eigener (sinnvoller) Semantik

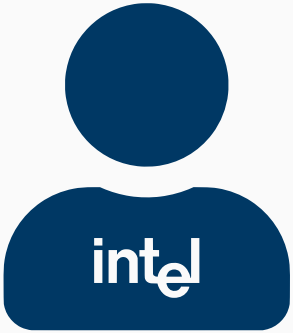
Zu implementierende Systemaufrufe

```
size_t write(int fd, const void *buf, size_t len);
size_t read(int fd, void *buf, size_t len);
void sleep(int ms);
int sem_init(int semid, int value);
void sem_destroy(int semid);
void sem_wait(int semid);
void sem_signal(int semid);
```

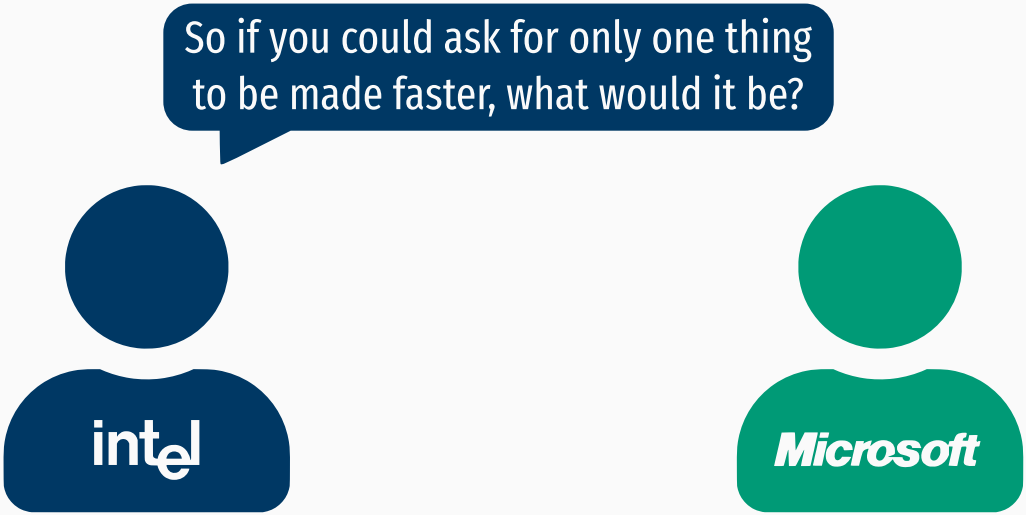
mit eigener (sinnvoller) Semantik

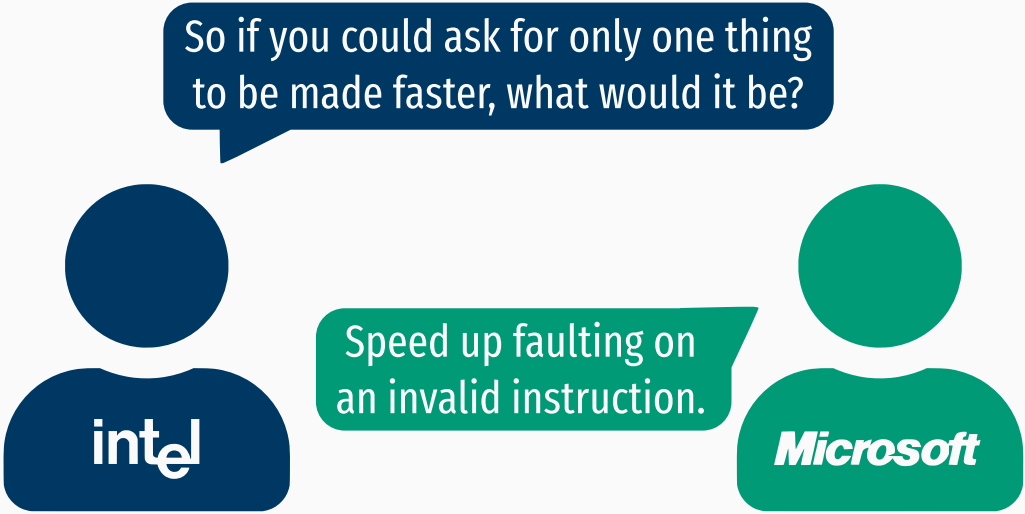
Implementierungstipps

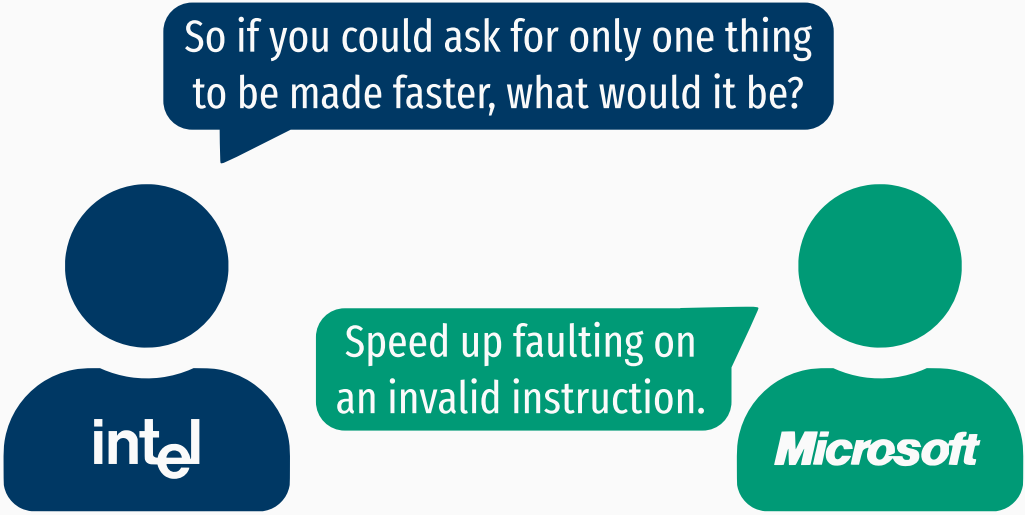
- `write` abhängig von `fd` an `kout` und DBG
- `write` einfach in angepassten `OutputStream` integrierbar
- Semaphoren (via `semid`) z.B. auf 64 limitiert (→ statisches Array)


intel


Microsoft


So if you could ask for only one thing to be made faster, what would it be?
intel
Microsoft


So if you could ask for only one thing to be made faster, what would it be?
intel
Speed up faulting on an invalid instruction.
Microsoft



➔ Windows/386 used an invalid instruction as its syscall trap!

(Quelle: Raymond Chan in Microsoft DevBlogs)

Schnelle Systemaufrufe

Spezielle Systemaufrufinstruktionen

Intel `sysenter/sysexit`

AMD `syscall/sysret`

Spezielle Systemaufrufinstruktionen

Intel `sysenter/sysexit`
AMD `syscall/sysret`

Gemeinsamkeiten: Konfiguration über *Model-specific Register (MSR)*, spezielles Layout der *Global Descriptor Table (GDT)* notwendig.

Spezielle Systemaufrufinstruktionen

Intel `sysenter/sysexit`
AMD `syscall/sysret`

Gemeinsamkeiten: Konfiguration über *Model-specific Register (MSR)*, spezielles Layout der *Global Descriptor Table (GDT)* notwendig.

Unterschiede bei der Kompatibilität:

	Intel-CPUs	**AMD-CPUs**
32 bit	`sysenter`	`sysenter` & `syscall`
64 bit	`sysenter` & `syscall`	`syscall`

Spezielle Systemaufrufinstruktionen

Intel `sysenter/sysexit`
AMD `syscall/sysret`

Gemeinsamkeiten: Konfiguration über *Model-specific Register (MSR)*, spezielles Layout der *Global Descriptor Table (GDT)* notwendig.

Unterschiede bei der Kompatibilität:

	Intel-CPUs	**AMD-CPUs**
32 bit	`sysenter`	`sysenter` & `syscall`
64 bit	`sysenter` & `syscall`	`syscall`

→ Wir wollen `syscall/sysret` für unser 64 bit **StuBSmi** verwenden

Instruktion `syscall` erlauben

ISDMv3 2.2.1, ISDMv4 & AAPMv2 3.1.7

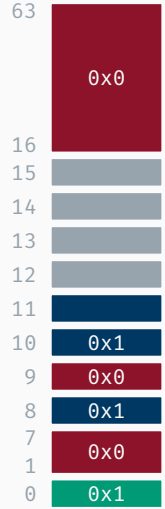
`MSR_EFER (0xc000 0080)`

- Reserviert
- Translation Cache Extension
- Fast `fxsave/fxrstor`
- Long Mode Segment Limit Enable
- Secure Virtual Machine Enable
- No-Execute Enable
- Long Mode Active
- Reserviert
- Long Mode Enable
- Reserviert
- System Call Extensions

Instruktion `syscall` erlauben

ISDMv3 2.2.1, ISDMv4 & AAPMv2 3.1.7

`MSR_EFER (0xc000 0080)`



Reserviert, muss `0` sein

Translation Cache Extension

Fast `fxsave/fxrstor`

Long Mode Segment Limit Enable

Secure Virtual Machine Enable

No-Execute Enable (Verwendung in Aufgabe 3)

Long Mode Active → gesetzt

Reserviert, muss `0` sein

Long Mode Enable → gesetzt (in `boot/longmode.asm`)

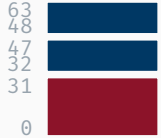
Reserviert, muss `0` sein

System Call Extensions aktiviert `syscall/sysret`

Einfacher Zugriff → `machine/core_msr.h`

Instruktion `syscall` steuern

MSR_STAR (0xc000 0081)

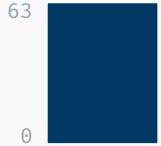


Segment für `sysret` (mit letzten 2 Bits für Ring 3 gesetzt)

Segment für `syscall`

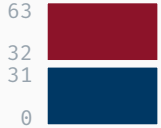
Reserviert (Einsprungsadresse bei 32 bit)

MSR_LSTAR (0xc000 0082)



Einsprungsadresse bei `syscall`

MSR_SFMASK (0xc000 0084)



Reserviert

Löschmaske für `rflags`

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter **`rip`** nach **`rcx`** sichern
  - **`rip`** auf Inhalt von **MSR_LSTAR** setzen

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter **`rip`** nach **`rcx`** sichern
  - **`rip`** auf Inhalt von **MSR_LSTAR** setzen
- Statusregister
  - alte **`rflags`** nach **`r11`** sichern
  - Bits gemäß **`mask`** aus **MSR_SFMASK** löschen:
    `rflags &= ~mask`

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter `rip` nach `rcx` sichern
  - `rip` auf Inhalt von **MSR_LSTAR** setzen
- Statusregister
  - alte `rflags` nach `r11` sichern
  - Bits gemäß `mask` aus **MSR_SFMASK** löschen:
    `rflags &= ~mask`
- Segmentregister
  - `segment` für `syscall` aus **MSR_STAR** (Bits 32 – 47)
  - `cs` auf `segment` setzen
  - `ss` auf `segment + 8` setzen

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter **`rip`** nach **`rcx`** sichern
  - **`rip`** auf Inhalt von **MSR_LSTAR** setzen
- Statusregister
  - alte **`rflags`** nach **`r11`** sichern
  - Bits gemäß **`mask`** aus **MSR_SFMASK** löschen:
    `rflags &= ~mask`
- Segmentregister
  - **`segment`** für `syscall` aus **MSR_STAR** (Bits 32 – 47)
  - **`cs`** auf **`segment`** setzen
  - **`ss`** auf **`segment + 8`** setzen
  - ➔ Ring 0 Datensegment (für **`ss`**) muss direkt über dem Ring 0 Codesegment (**`cs`**) liegen!

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter **`rip`** nach **`rcx`** sichern
  - **`rip`** auf Inhalt von **MSR_LSTAR** setzen
- Statusregister
  - alte **`rflags`** nach **`r11`** sichern
  - Bits gemäß **`mask`** aus **MSR_SFMASK** löschen:
    `rflags &= ~mask`
- Segmentregister
  - **`segment`** für `syscall` aus **MSR_STAR** (Bits 32 – 47)
  - **`cs`** auf **`segment`** setzen
  - **`ss`** auf **`segment + 8`** setzen
  - ➔ Ring 0 Datensegment (für **`ss`**) muss direkt über dem Ring 0 Codesegment (**`cs`**) liegen!
- Stapelzeiger (für Ring 0)

Interner Ablauf von `syscall`

- Instruktionszeiger
  - alter **`rip`** nach **`rcx`** sichern
  - **`rip`** auf Inhalt von **MSR_LSTAR** setzen
- Statusregister
  - alte **`rflags`** nach **`r11`** sichern
  - Bits gemäß **`mask`** aus **MSR_SFMASK** löschen:
    `rflags &= ~mask`
- Segmentregister
  - **`segment`** für `syscall` aus **MSR_STAR** (Bits 32 – 47)
  - **`cs`** auf **`segment`** setzen
  - **`ss`** auf **`segment + 8`** setzen
  - → Ring 0 Datensegment (für **`ss`**) muss direkt über dem Ring 0 Codesegment (**`cs`**) liegen!
- Stapelzeiger (für Ring 0) wird nicht (automatisch) gesetzt!

Interner Ablauf von `sysret`

- Instruktionszeiger
  - vorherigen **`rip`** aus **`rcx`** wiederherstellen

Interner Ablauf von `sysret`

- Instruktionszeiger
  - vorherigen `rip` aus `rcx` wiederherstellen
- Statusregister
  - vorherige `rflags` aus `r11` wiederherstellen

Interner Ablauf von `sysret`

- Instruktionszeiger
  - vorherigen **`rip`** aus **`rcx`** wiederherstellen
- Statusregister
  - vorherige **`rflags`** aus **`r11`** wiederherstellen
- Segmentregister
  - **`segment`** für `sysret` aus **`MSR_STAR`** (Bits 48 – 63)
  - **`cs`** bei Wechsel auf Ring 0 im 32 bit Modus auf **`segment`**, sonst (im *long mode* bleibend) auf **`segment + 16`** setzen
  - **`ss`** auf **`segment + 8`** setzen (unabhängig vom Modus)

Interner Ablauf von `sysret`

- Instruktionszeiger
  - vorherigen **`rip`** aus **`rcx`** wiederherstellen
- Statusregister
  - vorherige **`rflags`** aus **`r11`** wiederherstellen
- Segmentregister
  - **`segment`** für `sysret` aus **`MSR_STAR`** (Bits 48 – 63)
  - **`cs`** bei Wechsel auf Ring 0 im 32 bit Modus auf **`segment`**, sonst (im *long mode* bleibend) auf **`segment + 16`** setzen
  - **`ss`** auf **`segment + 8`** setzen (unabhängig vom Modus)

Modus abhängig von Instruktion: **`sysret`** [Opcode `0f 07`] für 32 bit, **`o64 sysret`** (NASM) bzw. **`sysretq`** (GNU) [→ Prefix 48] für 64 bit.

Interner Ablauf von `sysret`

- Instruktionszeiger
  - vorherigen **`rip`** aus **`rcx`** wiederherstellen
- Statusregister
  - vorherige **`rflags`** aus **`r11`** wiederherstellen
- Segmentregister
  - **`segment`** für `sysret` aus **`MSR_STAR`** (Bits 48 – 63)
  - **`cs`** bei Wechsel auf Ring 0 im 32 bit Modus auf **`segment`**, sonst (im *long mode* bleibend) auf **`segment + 16`** setzen
  - **`ss`** auf **`segment + 8`** setzen (unabhängig vom Modus)

  → Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen!

Modus abhängig von Instruktion: **`sysret`** [Opcode `0f 07`] für 32 bit, **`o64 sysret`** (NASM) bzw. **`sysretq`** (GNU) [→ Prefix 48] für 64 bit.

GDT Anpassung für `syscall/sysret`

- `syscall`: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen

GDT Anpassung für `syscall/sysret`

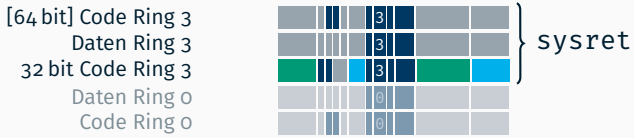
- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen

GDT Anpassung für `syscall/sysret`

- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen
- **`sysret`**: Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen

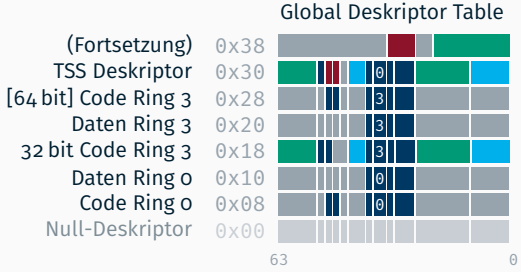
GDT Anpassung für `syscall/sysret`

- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen
- **`sysret`**: Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen



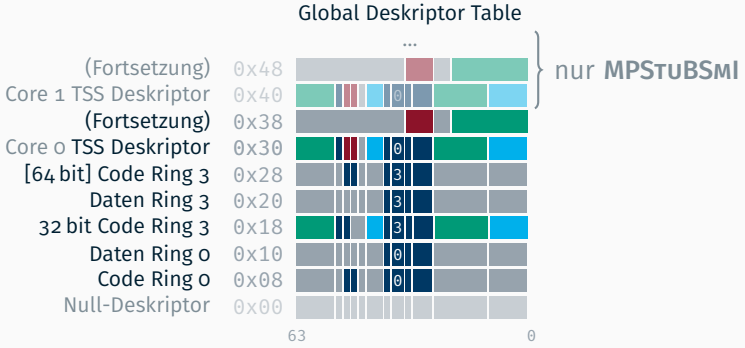
GDT Anpassung für `syscall/sysret`

- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen
- **`sysret`**: Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen



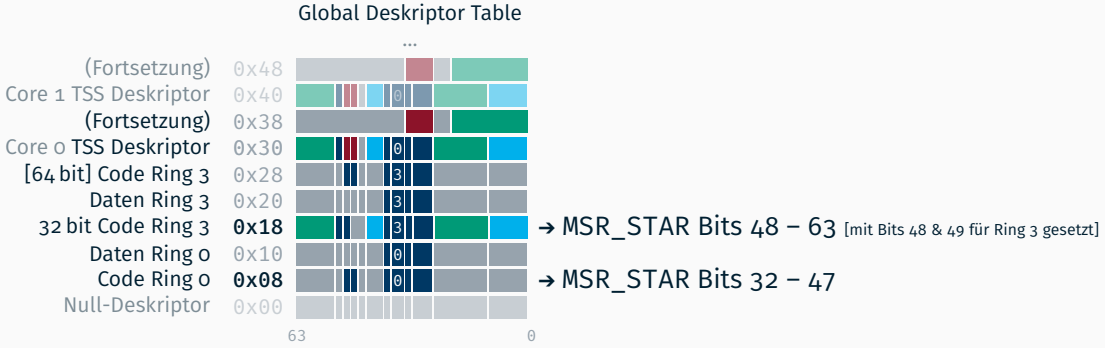
GDT Anpassung für `syscall/sysret`

- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen
- **`sysret`**: Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen



GDT Anpassung für `syscall/sysret`

- **`syscall`**: Ring 0 Datensegment muss direkt über dem Ring 0 Codesegment liegen
- **`sysret`**: Ring 3 Datensegment muss direkt unter dem 64 bit und direkt über dem 32 bit Ring 3 Codesegment liegen



Stapelzeiger

Woher bekommen wir bei `syscall` den korrekten Kernelstack?

Stapelzeiger

Woher bekommen wir bei `syscall` den korrekten Kernelstack?

OOStuBSmI

- Globale Variable, z.B. `syscall_kernelsp` (passend initialisiert und bei jedem Kontextwechsel aktualisiert)

→ `mov rsp, [syscall_kernelsp]` in Einsprungsfunktion

Stapelzeiger

Woher bekommen wir bei `syscall` den korrekten Kernelstack?

OOStuBSmI

- Globale Variable, z.B. `syscall_kernelsp` (passend initialisiert und bei jedem Kontextwechsel aktualisiert)

→ `mov rsp, [syscall_kernelsp]` in Einsprungsfunktion

MPStuBSmI

- *Core Local Storage* verwenden, in der Struktur einen Kernelstackpointer anlegen und aktuell halten

→ z.B. setzen mit `mov rsp, [gs:0x8]` in Einsprungsfunktion

Stapelzeiger

Woher bekommen wir bei `syscall` den korrekten Kernelstack?

OOStuBSmI

- Globale Variable, z.B. `syscall_kernelsp` (passend initialisiert und bei jedem Kontextwechsel aktualisiert)

→ `mov rsp, [syscall_kernelsp]` in Einsprungsfunktion

MPStuBSmI

- *Core Local Storage* verwenden, in der Struktur einen Kernelstackpointer anlegen und aktuell halten

→ z.B. setzen mit `mov rsp, [gs:0x8]` in Einsprungsfunktion

→ an `swapgs` denken!

Stapelzeiger

Woher bekommen wir bei `syscall` den korrekten Kernelstack?

OOStuBSmI

- Globale Variable, z.B. `syscall_kernelsp` (passend initialisiert und bei jedem Kontextwechsel aktualisiert)

→ `mov rsp, [syscall_kernelsp]` in Einsprungsfunktion

MPStuBSmI

- *Core Local Storage* verwenden, in der Struktur einen Kernelstackpointer anlegen und aktuell halten

→ z.B. setzen mit `mov rsp, [gs:0x8]` in Einsprungsfunktion

→ an `swapgs` denken!

Auch um den Userstackpointer muss sich gekümmert werden (bei Einsprung sichern und am Ende wiederherstellen)!

Testen der schnellen Systemaufrufe

Selben Systemaufrufe wie für die interruptbasierten Variante

- anderes Präfix (z.B. `sys_` und `fast_`)
- Codeduplikation vermeiden: gleichen `syscall_handler` verwenden, nur unterschiedlicher Einsprungspunkt (`fast_syscall_entry`)

Testen der schnellen Systemaufrufe

Selben Systemaufrufe wie für die interruptbasierten Variante

- anderes Präfix (z.B. `sys_` und `fast_`)
- Codeduplikation vermeiden: gleichen `syscall_handler` verwenden, nur unterschiedlicher Einsprungspunkt (`fast_syscall_entry`)

→ ausgiebig testen, idealerweise modularer Aufbau

- wird in den nachfolgenden Aufgaben verwendet & erweitert

Testen der schnellen Systemaufrufe

Selben Systemaufrufe wie für die interruptbasierten Variante

- anderes Präfix (z.B. `sys_` und `fast_`)
- Codeduplikation vermeiden: gleichen `syscall_handler` verwenden, nur unterschiedlicher Einsprungspunkt (`fast_syscall_entry`)

→ ausgiebig testen, idealerweise modularer Aufbau

- wird in den nachfolgenden Aufgaben verwendet & erweitert

Für 7.5 ECTS: Benchmark der beiden Varianten

- Einführung eines zusätzlichen nop-Systemaufrufs
- wieder TSC (gemäß White Paper) verwenden
- an Einschränkungen im Ring 3 (privilegierte Befehle) denken

Fragen?