

# Übung zu Betriebssystemtechnik

## Aufgabe 2: Systemaufrufe

---

07. Mai 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

## 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

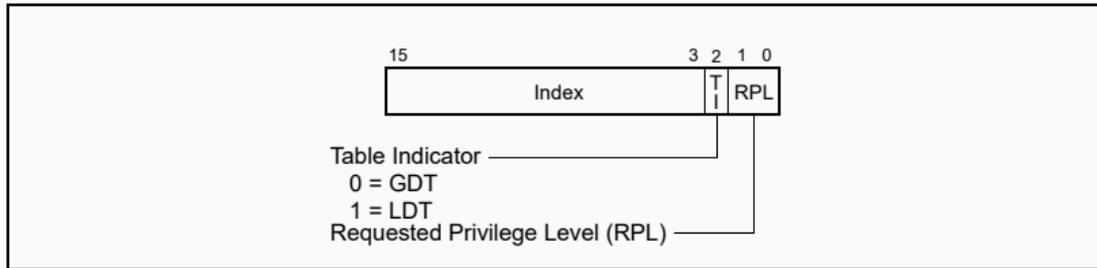


Figure 3-6. Segment Selector

## ***Wiederholung: Aufrufkonvention***

---

## Kontextsicherung gemäß Konvention

```
void baz(){
    ...

    // flüchtige Register
    // sichern

    func();

    // flüchtige Register
    // wiederherstellen

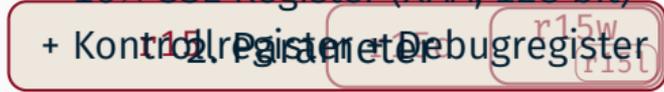
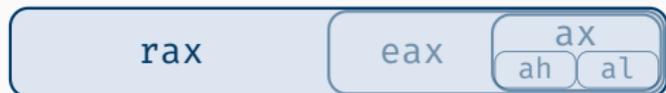
    ...
}
```

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

    ...

    // nicht-flüchtige
    // Register
    // wiederherstellen

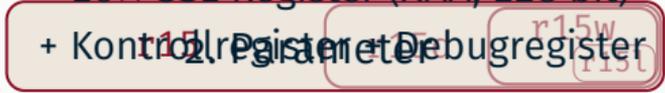
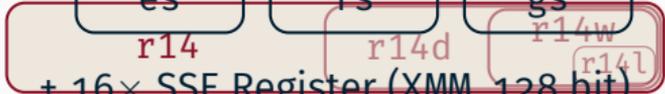
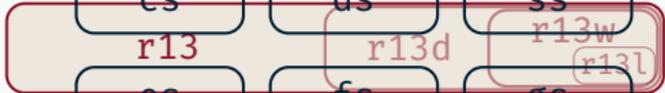
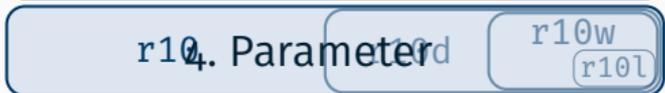
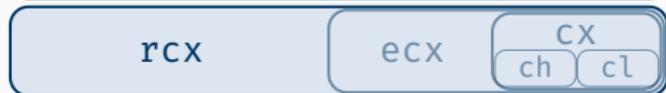
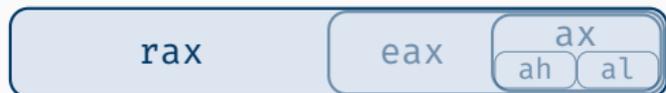
    return;
}
```



flüchtige (scratch / caller-save) Register

nicht-flüchtig (non-scratch / callee-save)

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



flüchtige (scratch / caller-save) Register

nicht-flüchtig (non-scratch / callee-save) 6. Parameter

## Aufgabe 2

---

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

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),
- synchrone Ausnahmen (durch Exceptions) sowie den
- Unterbrechungsbefehl (durch Softwareinstruktion `int`)

einschließt.

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

# 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

	<i>Kernelstack</i>
ss	} durch CPU bei einer Unterbrechung auf Stack gelegte Werte
rsp	
rflags	
cs	
rip	
Error Code?	} Scratch Register Sicherung durch <code>interrupt_entry</code> (in <code>interrupt/handler.asm</code> ) ggf. Sicherung durch den Compiler bei <code>interrupt_handler</code> (in <code>interrupt/handler.cc</code> )
rax	
...	
r11	
rbp?	
rbx?	}
...	

# Interrupt Deskriptor für 0x80 / Syscall

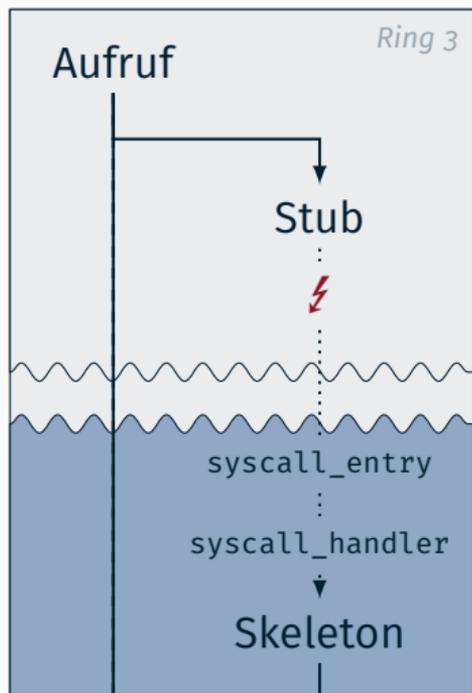
127	0	<b>Unused</b> – muss 0 sein
96		
95	0x100	<b>Offset (high):</b> oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. syscall_entry)
48		
47	1	<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)
46		
45	3	<b>Descriptor Privilege Level:</b> muss nun Ring 3 sein
44	0	<b>Storage Segment:</b> 0 für Interrupt und Traps
43	1	<b>Mode:</b> 16-bit (0) oder 32/64-bit (1)
42		
40	6	<b>Type:</b> Task (5), Interrupt (6) oder Trap (7)?
39		
35	0	<b>Unused</b> – muss 0 sein
34		
32	0	<b>Interrupt Stack Table</b>
31		
16	8	<b>Selector:</b> Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
15		
0	0x0430	<b>Offset (low):</b> 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
  - 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 Systemaufruf foo



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

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

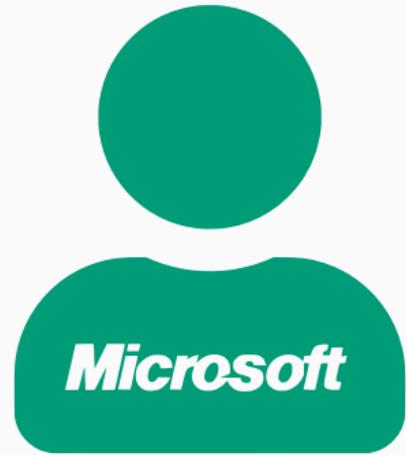
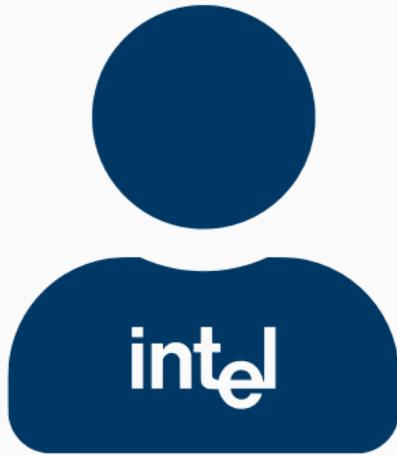
## Zu implementierende Systemaufrufe

```
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

### 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)



→ 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`

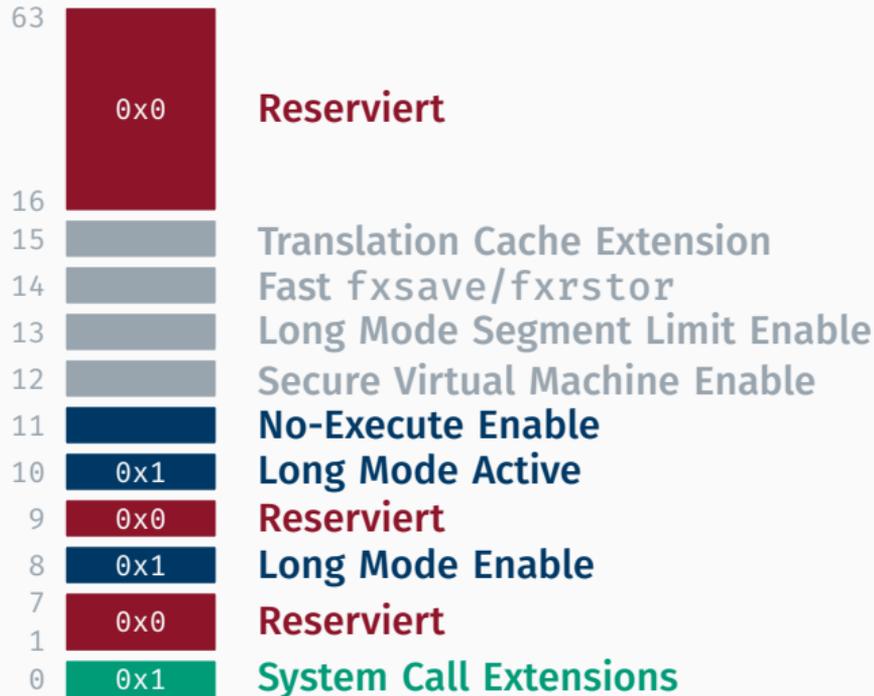
**Gemeinsamkeiten:** Konfiguration über *Model-specific Register (MSR)*, spezielles Layout der *Global Descriptor Table (GDT)* notwendig.

**Unterschiede bei der Kompatibilität:**

	<b>Intel-CPU</b> s	<b>AMD-CPU</b> s
<b>32 bit</b>	<code>sysenter</code>	<code>sysenter &amp; syscall</code>
<b>64 bit</b>	<code>sysenter &amp; syscall</code>	<code>syscall</code>

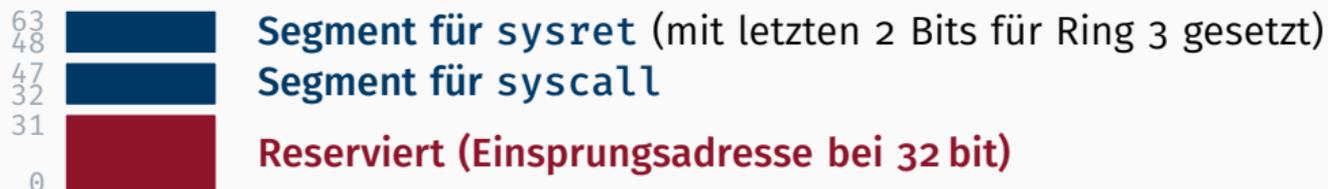
→ Wir wollen `syscall/sysret` für unser 64 bit **STUBSMI** verwenden

## MSR\_EFER (0xc000 0080)

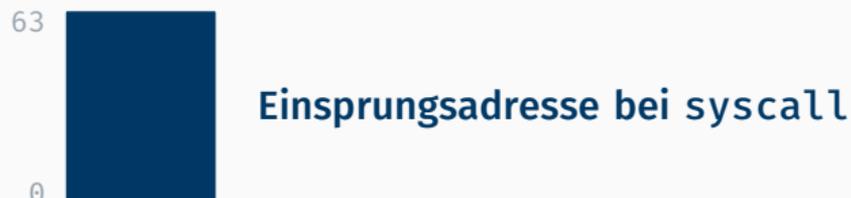


Einfacher Zugriff → `machine/core_msr.h`

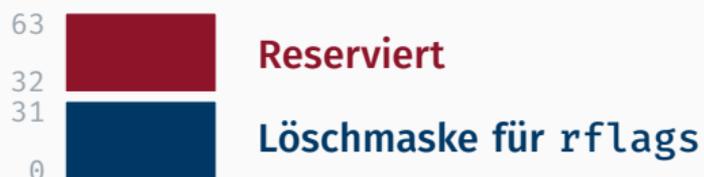
## MSR\_STAR (0xc000 0081)



## MSR\_LSTAR (0xc000 0082)



## MSR\_SFMASK (0xc000 0084)



# 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
  - 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.



Woher bekommen wir bei `syscall` den korrekten Kernelstack?

## OOSTUBSMI

- Globale Variable, z.B. `syscall_kernel_sp` (passend initialisiert und bei jedem Kontextwechsel aktualisiert)
- `mov rsp, [syscall_kernel_sp]` 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`)

→ 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?**