

# Übung zu Betriebssystembau

## Interruptbehandlung

---

31. Oktober 2023

Peter Ulbrich & Alexander Lochmann  
(Mit Material vom Lehrstuhl 4 der FAU)

Arbeitsgruppe Systemsoftware  
Technische Universität Dortmund

# Interrupts und Traps

---

# Dann schlug ein Blitz ein ...



CPU

# Dann schlug ein Blitz ein ...



CPU

# Dann schlug ein Blitz ein ...



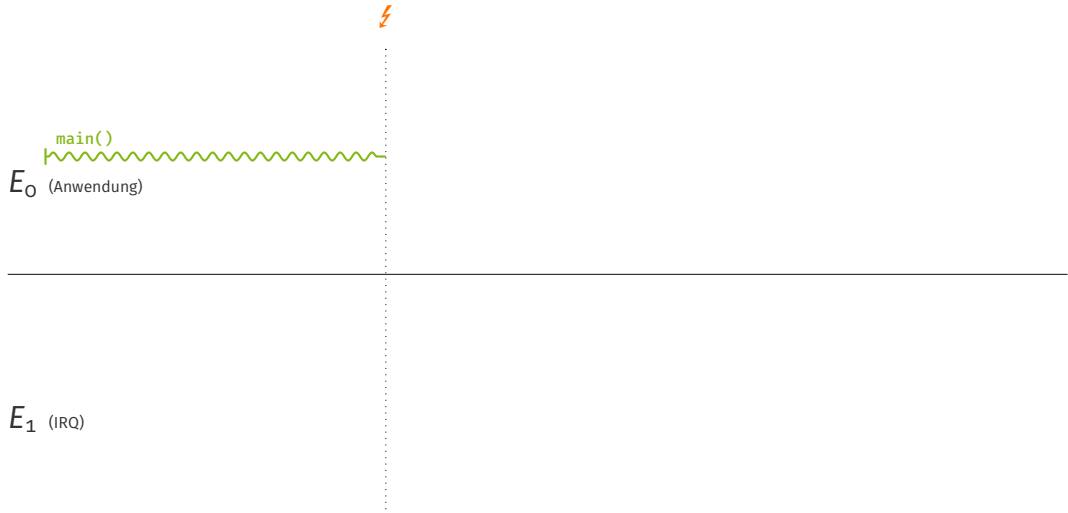
# Unterbrechung aus Anwendungssicht

`main()`  
  
 $E_0$  (Anwendung)

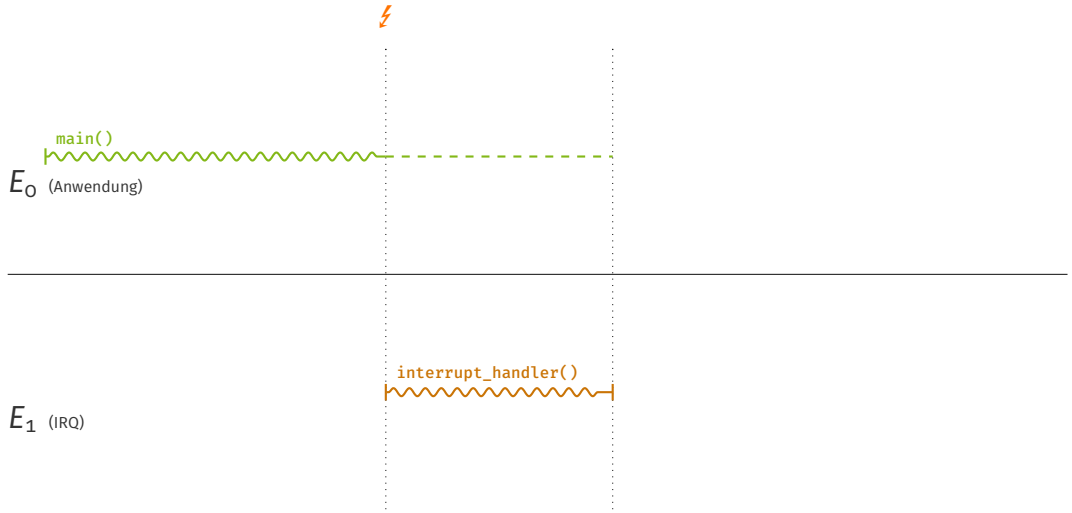
---

$E_1$  (IRQ)

# Unterbrechung aus Anwendungssicht

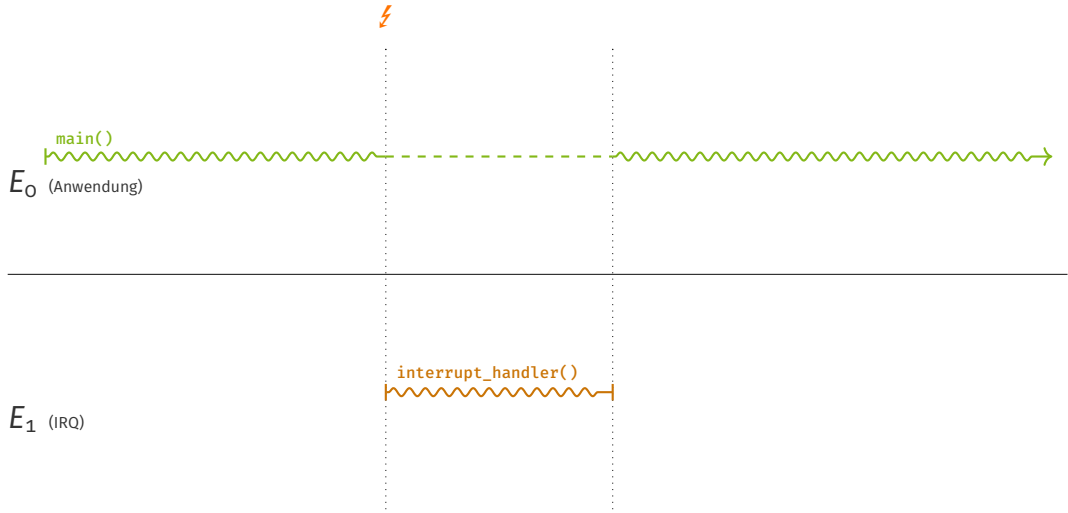


# Unterbrechung aus Anwendungssicht





# Unterbrechung aus Anwendungssicht



# Zustandsicherung (1)

Was muss **mindestens** gesichert werden?

# Zustandsicherung (1)

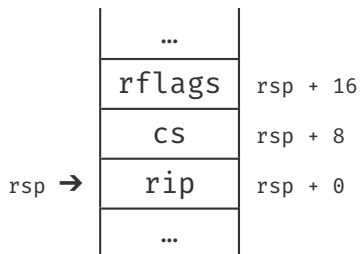
Was muss **mindestens** gesichert werden?

- CPU sichert automatisch

**rip** Instruktionszeiger/Rücksprungadresse

**rflags** Status/Condition Codes

**cs** Aktuelles Code Segment



# Zustandsicherung (1)

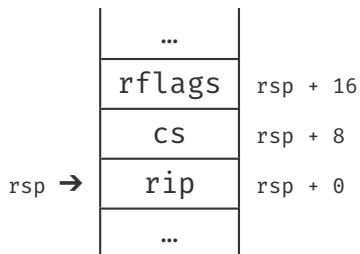
Was muss **mindestens** gesichert werden?

- CPU sichert automatisch

**rip** Instruktionszeiger/Rücksprungadresse

**rflags** Status/Condition Codes

**cs** Aktuelles Code Segment



- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iretq`

# Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04
05
06     iretq
```

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

```
01 // C++
02
03 void interrupt_handler()
04 {
05     // Magie.
06 }
```

# Unterbrechungsbehandlung

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

```
01 // C++
02
03 void interrupt_handler()
04 {
05     // Magie.
06 }
```

```
01 heinloth:~/oostubs$ make
02 LD          .build/system64
03 .build/interrupt/handler.asm.o: in function `interrupt_entry':
04 interrupt/handler.asm:(.text+0x16): undefined reference to `
    interrupt_handler'
```



# Unterbrechungsbehandlung

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

```
01 // C++
02
03 void interrupt_handler()
04 {
05     // Magie.
06 }
```

```
01 heinloth:~/oostubs$ objdump -d .build/interrupt/handler.o
02 Disassembly of section .text:
03
04 0000000000000000 <_Z17interrupt_handlerv>:
05     0:  c3                retq
```

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

```
01 // C++ (mit C Linkage)
02 extern "C"
03 void interrupt_handler()
04 {
05     // Magie.
06 }
```

# Unterbrechungsbehandlung

## unter Verwendung einer Hochsprache

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Behandle IRQ
04     ;; in Hochsprache
05     call interrupt_handler
06     iretq
```

```
01 // C++ (mit C Linkage)
02 extern "C"
03 void interrupt_handler()
04 {
05     // Magie.
06 }
```

```
01 heinloth:~/oostubs$ objdump -d .build/interrupt/handler.o
02 Disassembly of section .text:
03
04 0000000000000000 <interrupt_handler>:
05     0:  f3 c3                repz ret
```

## Zustandssicherung (2)

Was ist mit den restlichen Registern?

# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code

# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen

# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht

# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert



# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert
  3. Ein Teil wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert

# Zustandssicherung (2)

Was ist mit den restlichen Registern?

- Müssen durch den Interrupthandler selbst gesichert werden.
  - entweder im Assembler-Teil oder
  - der Compiler generiert bereits entsprechend Code
- Kontextsicherung beim Aufruf von Funktionen
  1. Aufrufende Funktion sichert alle Register, die sie braucht
  2. Aufgerufene Funktion sichert alle Register, die sie verändert
  3. Ein Teil wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert
- In der Praxis wird Variante 3 verwendet
  - Aufteilung ist grundsätzlich compilerspezifisch
  - Um Interoperabilität auf Binärcodeebene sicher zu stellen gibt es jedoch Konventionen (bei x64 zwei: *Microsoft* und *System V*)
  - Aufrufkonvention ist Teil der *Application Binary Interface* (ABI)

# Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x64 (nach System V ABI) sind **rax**, **rcx**, **rdx**, **rsi**, **rdi** und **r8–r11** als flüchtig definiert

# Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x64 (nach System V ABI) sind **rax**, **rcx**, **rdx**, **rsi**, **rdi** und **r8 – r11** als flüchtig definiert
- Nicht-flüchtige Register (*non-scratch registers*)
  - Compiler geht davon aus, dass der Inhalt durch Unterprogramm nicht verändert wird
  - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
  - Beim x64 sind alle sonstigen Register als nicht-flüchtig definiert: **rbx**, **rbp**, **rsp** und **r12 – r15**

# Aufteilung der Register in zwei Gruppen

- Flüchtige Register (*scratch registers*)
  - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
  - Aufrufer muss Inhalt gegebenenfalls sichern
  - Beim x64 (nach System V ABI) sind **rax**, **rcx**, **rdx**, **rsi**, **rdi** und **r8 – r11** als flüchtig definiert
- Nicht-flüchtige Register (*non-scratch registers*)
  - Compiler geht davon aus, dass der Inhalt durch Unterprogramm nicht verändert wird
  - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
  - Beim x64 sind alle sonstigen Register als nicht-flüchtig definiert: **rbx**, **rbp**, **rsp** und **r12 – r15**



**Unterbrechungsbehandlungen müssen flüchtige Register sichern!**

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03
04
05
06
07
08
09
10 call interrupt_handler
11
12
13
14
15
16 iretq
```

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08
09
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08
09
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02
03
04
05
06
07
08
09
10
11
12 extern "C"
13 void interrupt_handler()
14 {
15     // Magie.
16 }
```



# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08
09
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02 struct Context {
03
04
05
06
07
08
09
10 } __attribute__((packed));
11
12 extern "C"
13 void interrupt_handler(Context* c)
14 {
15     // Magie.
16 }
```

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08
09
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02 struct Context {
03     uint64_t r11;
04     // ...
05     uint64_t rcx;
06     uint64_t rax;
07
08
09
10 } __attribute__((packed));
11
12 extern "C"
13 void interrupt_handler(Context* c)
14 {
15     // Magie.
16 }
```

# Parameterübergabe

- auf Stack
  - bei x86 war dies gemäß der *C declaration* der Standard

# Parameterübergabe

- auf Stack
  - bei x86 war dies gemäß der *C declaration* der Standard
- in Register
  - Vorteil: schneller
  - Problem: Anzahl der Register begrenzt
- kombiniert
  - die ersten Parameter via Register, danach bei Bedarf Stack

# Parameterübergabe

- auf Stack
  - bei x86 war dies gemäß der *C declaration* der Standard
- in Register
  - Vorteil: schneller
  - Problem: Anzahl der Register begrenzt
- kombiniert
  - die ersten Parameter via Register, danach bei Bedarf Stack
  - auch nach x64 *System V ABI*:
    - Parameter zuerst in Register **rdi**, **rsi**, **rdx**, **rcx**, **r8** und **r9**
    - im Userspace danach auch in die Register **xmm0** – **xmm7**
    - Rest auf Stack (letzter Parameter wird als erstes gepushed)

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08
09
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02 struct Context {
03     uint64_t r11;
04     // ...
05     uint64_t rcx;
06     uint64_t rax;
07
08
09
10 } __attribute__((packed));
11
12 extern "C"
13 void interrupt_handler(Context* c)
14 {
15     // Magie.
16 }
```

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08     ;; Pointer auf Stack
09     mov rdi, rsp
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02 struct Context {
03     uint64_t r11;
04     // ...
05     uint64_t rcx;
06     uint64_t rax;
07
08
09
10 } __attribute__((packed));
11
12 extern "C"
13 void interrupt_handler(Context* c)
14 {
15     // Magie.
16 }
```

# Kontextsicherung in der Unterbrechungsbehandlung

```
01 ;; Assembler
02 interrupt_entry:
03     ;; Kontext sichern
04     push rax
05     push rcx
06     ;; ...
07     push r11
08     ;; Pointer auf Stack
09     mov rdi, rsp
10     call interrupt_handler
11     ;; wiederherstellen
12     pop r11
13     ;; ...
14     pop rcx
15     pop rax
16     iretq
```

```
01 // C++
02 struct Context {
03     uint64_t r11;
04     // ...
05     uint64_t rcx;
06     uint64_t rax;
07     uint64_t rip;
08     uint64_t cs;
09     uint64_t rflags;
10 } __attribute__((packed));
11
12 extern "C"
13 void interrupt_handler(Context* c)
14 {
15     // Magie.
16 }
```



# Dann schlug ein Blitz ein ...



# Dann schlug ein Blitz ein ...



# Interruptvektoren (x86/x64)

0

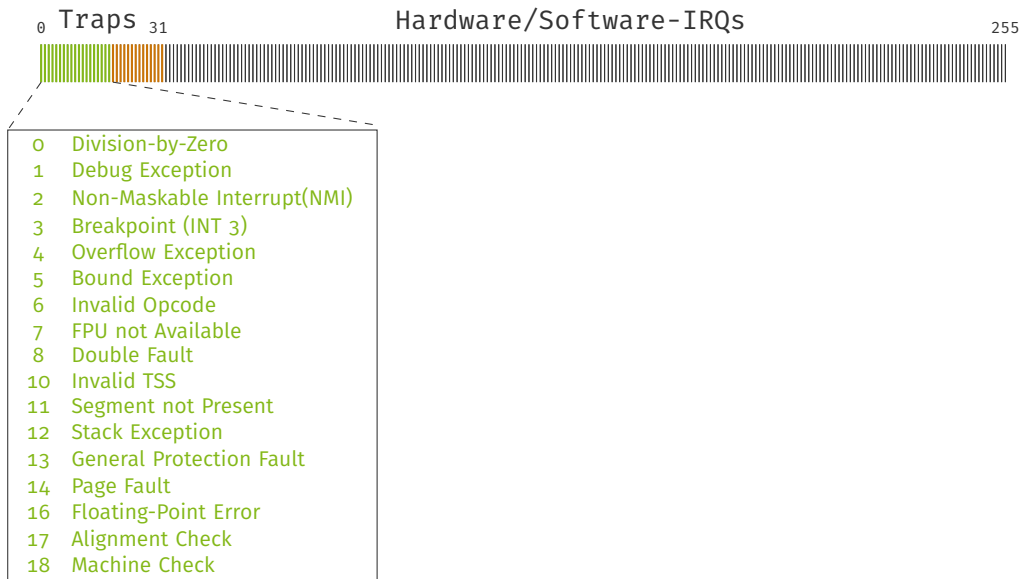
255



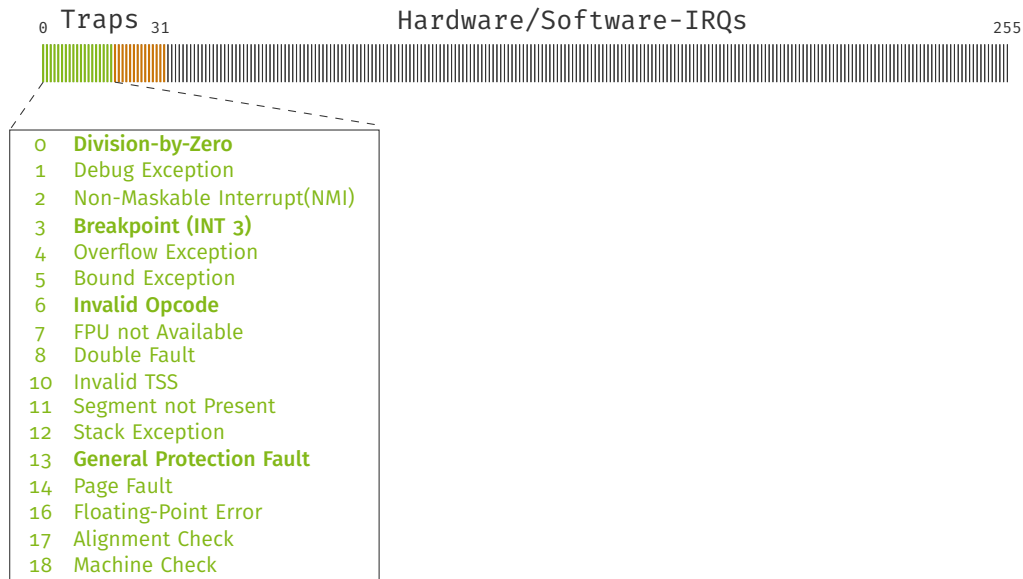
# Interruptvektoren (x86/x64)



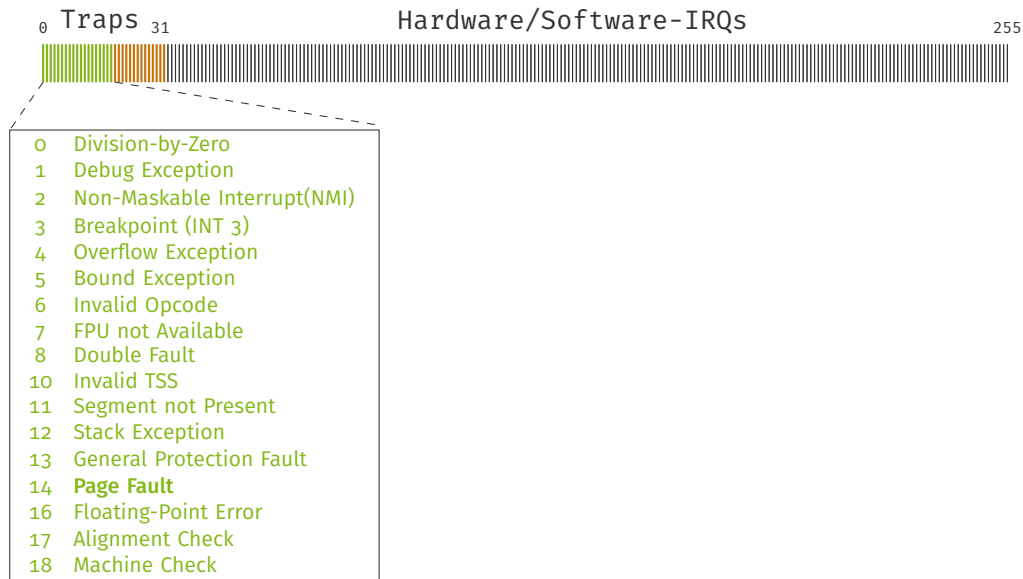
# Interruptvektoren (x86/x64)



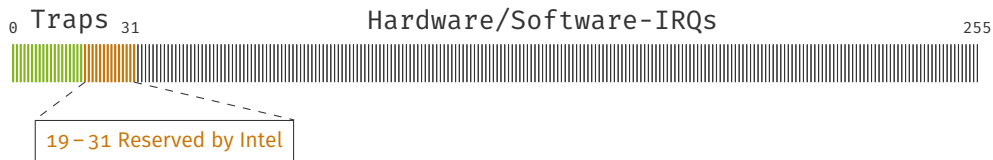
# Interruptvektoren (x86/x64)



# Interruptvektoren (x86/x64)



# Interruptvektoren (x86/x64)





# Interruptvektoren (x86/x64)



## 32 – 255 Einträge für IRQs

- Softwareauslösung mit `int <vec#>`
- Hardwareauslösung durch externe Geräte

# Interruptvektoren (x86/x64)



Kann durch Prozessorbefehle maskiert werden

**cli** (clear interrupt flag) Interruptleitung sperren

**sti** (set interrupt flag) Interruptleitung freigeben

# Unterbrechungsbehandlung

```
01
02 ;; Assembler
03 interrupt_entry:
04     ;; Kontext sichern
05     push rax
06     push rcx
07     ;; ...
08     push r11
09     ;; Pointer auf Stack
10     mov rdi, rsp
11
12
13     call interrupt_handler
14     ;; wiederherstellen
15     pop r11
16     ;; ...
17     pop rcx
18     pop rax
19     iretq
```

```
01
02 // C++
03 extern "C"
04 void interrupt_handler(
05     Context* c
06
07 )
08 {
09     // Magie:
10
11
12
13
14
15
16
17
18 }
```

# Unterbrechungsbehandlung

```
01
02 ;; Assembler
03 interrupt_entry:
04     ;; Kontext sichern
05     push rax
06     push rcx
07     ;; ...
08     push r11
09     ;; Pointer auf Stack
10     mov rdi, rsp
11
12
13     call interrupt_handler
14     ;; wiederherstellen
15     pop r11
16     ;; ...
17     pop rcx
18     pop rax
19     iretq
```

```
01
02 // C++
03 extern "C"
04 void interrupt_handler(
05     Context* c,
06     uint32_t vector
07 )
08 {
09     // Magie:
10     switch (vector){
11         case KBD:
12             kbd.magic();
13             break;
14         case TMR:
15             tmr.magic();
16             break;
17     }
18 }
```

# Unterbrechungsbehandlung für Vektor 6

```
01
02 ;; Assembler
03 interrupt_entry_6:
04     ;; Kontext sichern
05     push rax
06     push rcx
07     ;; ...
08     push r11
09     ;; Pointer auf Stack
10     mov rdi, rsp
11     ;; Vektornummer
12     mov rsi, 6
13     call interrupt_handler
14     ;; wiederherstellen
15     pop r11
16     ;; ...
17     pop rcx
18     pop rax
19     iretq
```

```
01
02 // C++
03 extern "C"
04 void interrupt_handler(
05     Context* c,
06     uint32_t vector
07 )
08 {
09     // Magie:
10     switch (vector){
11         case KBD:
12             kbd.magic();
13             break;
14         case TMR:
15             tmr.magic();
16             break;
17     }
18 }
```

# Unterbrechungsbehandlung beim Binden

```
01
02 ;; Assembler
03 interrupt_entry_6:
04 ;; Kontext sichern
05 push rax
06 push rcx
07 ;; ...
08 push r11
09 ;; Pointer auf Stack
10 mov rdi, rsp
11 ;; Vektornummer
12 mov rsi, 6
13 call 0x10070f0 <interrupt_handler>
14 ;; wiederherstellen
15 pop r11
16 ;; ...
17 pop rcx
18 pop rax
19 iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

**10070f0** <interrupt\_handler>

# Unterbrechungsbehandlung beim Binden

```
01
02 ;; Assembler
03 interrupt_entry_6:
04 ;; Kontext sichern
05 push rax
06 push rcx
07 ;; ...
08 push r11
09 ;; Pointer auf Stack
10 mov rdi, rsp
11 ;; Vektornummer
12 mov rsi, 6
13 call interrupt_handler
14 ;; wiederherstellen
15 pop r11
16 ;; ...
17 pop rcx
18 pop rax
19 iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

**100 70f0** <interrupt\_handler>

**100 0230** <interrupt\_entry\_6>

# Unterbrechungsbehandlung beim Binden

```
01
02 ;; Assembler
03 interrupt_entry_6:
04 ;; Kontext sichern
05 push rax
06 push rcx
07 ;; ...
08 push r11
09 ;; Pointer auf Stack
10 mov rdi, rsp
11 ;; Vektornummer
12 mov rsi, 6
13 call interrupt_handler
14 ;; wiederherstellen
15 pop r11
16 ;; ...
17 pop rcx
18 pop rax
19 iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <interrupt_handler>
...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```



# Generische Unterbrechungsbehandlungen

```
01 %macro IRQ 1
02 align 8
03 interrupt_entry_%1:
04     ;; Kontext sichern
05     push rax
06     push rcx
07     ;; ...
08     push r11
09     ;; Pointer auf Stack
10     mov rdi, rsp
11     ;; Vektornummer
12     mov rsi, %1
13     call interrupt_handler
14     ;; wiederherstellen
15     pop r11
16     ;; ...
17     pop rcx
18     pop rax
19     iretq
20 %endmacro
```

```
01 heinloth:~/oostubs$ make
02 ASM interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <interrupt_handler>
...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

**Woher weiß die CPU wo die entsprechende  
Unterbrechungsbehandlung liegt?**

# Interrupt Deskriptor

127	Orange	<b>Unused</b> – muss 0 sein
96		
95	Green	<b>Offset (high):</b> oberer Teil der Einsprungsadresse
48		für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code> )
47	Green	<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)
46	Green	<b>Descriptor Privilege Level</b>
45		
44	Green	<b>Storage Segment:</b> 0 für Interrupt und Traps
43	Green	<b>Mode:</b> 16-bit (0) oder 32/64-bit (1)
42	Green	<b>Type:</b> Task (5), Interrupt (6) oder Trap (7)?
40		
39	Orange	<b>Unused</b> – muss 0 sein
35		
34	Green	<b>Interrupt Stack Table</b>
32		
31	Green	<b>Selector:</b> Codesegment, in das beim Interrupt
16		gewechselt wird (i.d.R. Kernel-Codesegment)
15	Green	<b>Offset (low):</b> unterer Teil der Einsprungsadresse
0		für die Interruptbehandlung

# Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6

127	Orange	<b>Unused</b> – muss 0 sein
96		
95	Green	<b>Offset (high):</b> oberer Teil der Einsprungsadresse
48		für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code> )
47	Green	<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)
46	Green	<b>Descriptor Privilege Level</b>
45		
44	Green	<b>Storage Segment:</b> 0 für Interrupt und Traps
43	Green	<b>Mode:</b> 16-bit (0) oder 32/64-bit (1)
42	Green	<b>Type:</b> Task (5), Interrupt (6) oder Trap (7)?
40		
39	Orange	<b>Unused</b> – muss 0 sein
35		
34	Green	<b>Interrupt Stack Table</b>
32		
31	Green	<b>Selector:</b> Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15	Green	<b>Offset (low):</b> unterer Teil der Einsprungsadresse
0		für die Interruptbehandlung

für **100 0230 <interrupt\_entry\_6>**

# Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6

127	0	<b>Unused</b> – muss 0 sein
96		
95	0x100	<b>Offset (high):</b> oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code> )
48		
47	1	<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)
46		
45	0	<b>Descriptor Privilege Level</b>
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	0	<b>Interrupt Stack Table</b>
32		
31		
	8	<b>Selector:</b> Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15		
	0x0230	<b>Offset (low):</b> unterer Teil der Einsprungsadresse für die Interruptbehandlung
0		

für **100 0230** `<interrupt_entry_6>`

# Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6

127	0	<b>Unused</b> – muss 0 sein
96		
95	0x100	<b>Offset (high):</b> oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code> )
48		
47	1	<b>Present:</b> Eintrag aktiv (1) oder inaktiv (0)
46		
45	0	<b>Descriptor Privilege Level</b>
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	0	<b>Interrupt Stack Table</b>
32		
31	8	<b>Selector:</b> Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15	0x0230	<b>Offset (low):</b> unterer Teil der Einsprungsadresse für die Interruptbehandlung
0		

für `100 0230 <interrupt_entry_6>` → `0x100 8e00 0008 0230`

# Interrupt Deskriptor Tabelle (IDT)

100 30e0 <interrupt\_entry\_255>

...

100 0230 <interrupt\_entry\_6>

0x100 8e00 0008 0230

100 0200 <interrupt\_entry\_5>

100 01d0 <interrupt\_entry\_4>

100 01a0 <interrupt\_entry\_3>

100 0170 <interrupt\_entry\_2>

100 0140 <interrupt\_entry\_1>

100 0110 <interrupt\_entry\_0>

# Interrupt Deskriptor Tabelle (IDT)

100 30e0	<interrupt_entry_255>	→	0x100 8e00 0008 30e0
...			...
100 0230	<interrupt_entry_6>	→	0x100 8e00 0008 0230
100 0200	<interrupt_entry_5>	→	0x100 8e00 0008 0200
100 01d0	<interrupt_entry_4>	→	0x100 8e00 0008 01d0
100 01a0	<interrupt_entry_3>	→	0x100 8e00 0008 01a0
100 0170	<interrupt_entry_2>	→	0x100 8e00 0008 0170
100 0140	<interrupt_entry_1>	→	0x100 8e00 0008 0140
100 0110	<interrupt_entry_0>	→	0x100 8e00 0008 0110



# Interrupt Deskriptor Tabelle (IDT)

```
100 30e0 <interrupt_entry_255>
    ...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

Speicher

Interrupt Deskriptortabelle
0x100 8e00 0008 30e0
...
0x100 8e00 0008 0230
0x100 8e00 0008 0200
0x100 8e00 0008 01d0
0x100 8e00 0008 01a0
0x100 8e00 0008 0170
0x100 8e00 0008 0140
0x100 8e00 0008 0110

# Interrupt Deskriptor Tabelle (IDT)

100 30e0 <interrupt\_entry\_255>

...

100 0230 <interrupt\_entry\_6>

100 0200 <interrupt\_entry\_5>

100 01d0 <interrupt\_entry\_4>

100 01a0 <interrupt\_entry\_3>

100 0170 <interrupt\_entry\_2>

100 0140 <interrupt\_entry\_1>

100 0110 <interrupt\_entry\_0>

101 f1b0

101 e220

101 e210

101 e200

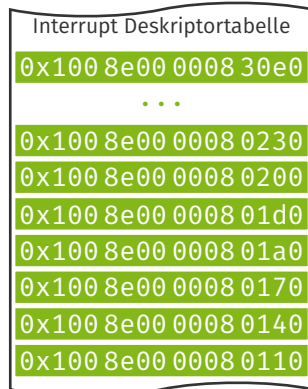
101 e1f0

101 e1e0

101 e1d0

101 e1c0

Speicher



# Interrupt Deskriptor Tabelle (IDT)



# Interrupt Deskriptor Tabelle (IDT)

## IDT-Register `idtr`

79



**Basis:** Startadresse der IDT

16

15

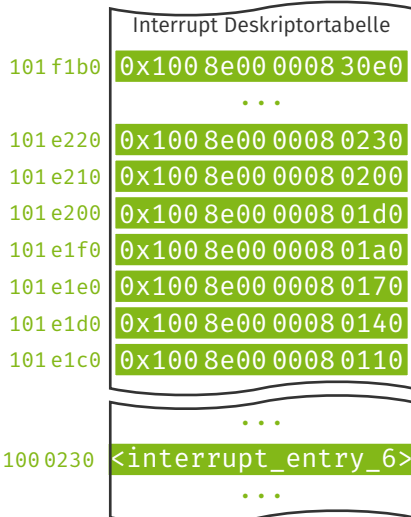


**Limit:** Bytes

*Einträge \* 16 - 1*

0

## Speicher



# Interrupt Deskriptor Tabelle (IDT)

## IDT-Register `idttr`

79

101e1c0

**Basis:** Startadresse der IDT

16

15

4095

**Limit:** Bytes

0

*Einträge \* 16 - 1*

## Speicher



# Interrupt Deskriptor Tabelle (IDT)

## IDT-Register `idtr`

79	101e1c0	<b>Basis:</b> Startadresse der IDT
16		
15	4095	<b>Limit:</b> Bytes
0		<i>Einträge * 16 - 1</i>

### Instruktionen:

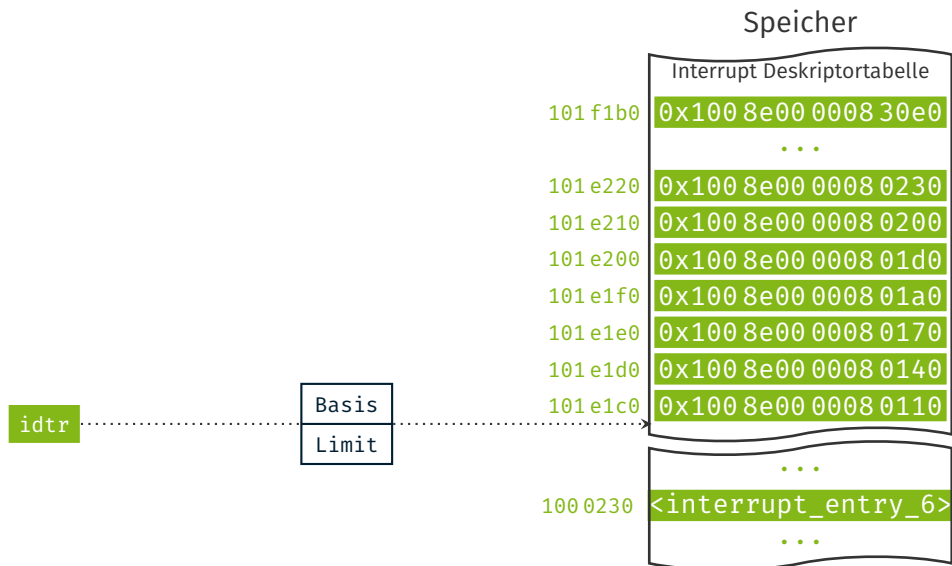
`lidt` in Register laden

`sidt` aus Register lesen

## Speicher

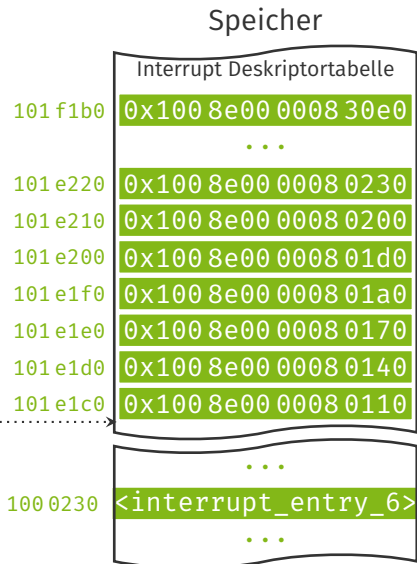
Interrupt Deskriptortabelle	
101f1b0	0x1008e00000830e0
	...
101e220	0x1008e0000080230
101e210	0x1008e0000080200
101e200	0x1008e00000801d0
101e1f0	0x1008e00000801a0
101e1e0	0x1008e0000080170
101e1d0	0x1008e0000080140
101e1c0	0x1008e0000080110
	...
1000230	<interrupt_entry_6>
	...

# Interrupt Deskriptor Tabelle (IDT)



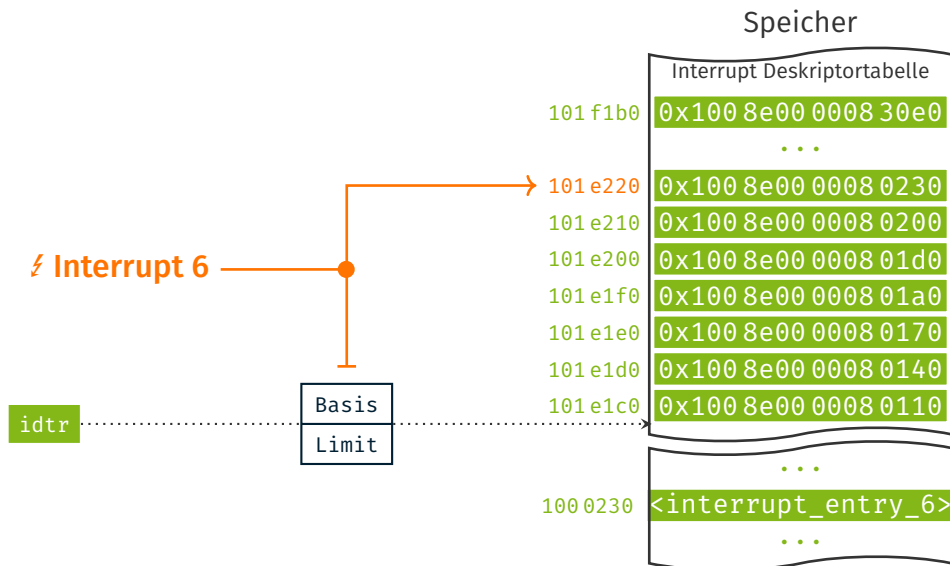
# Interrupt Deskriptor Tabelle (IDT)

⚡ Interrupt 6

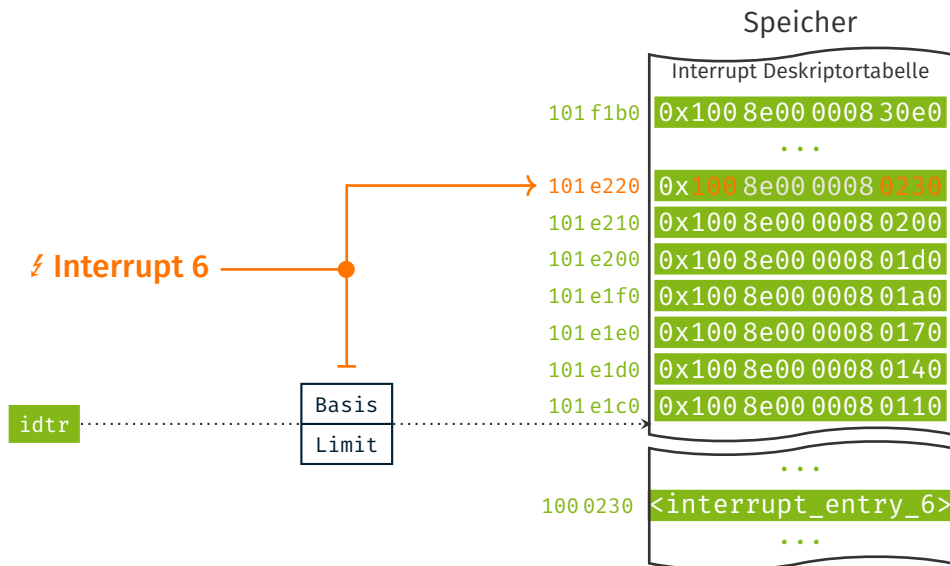




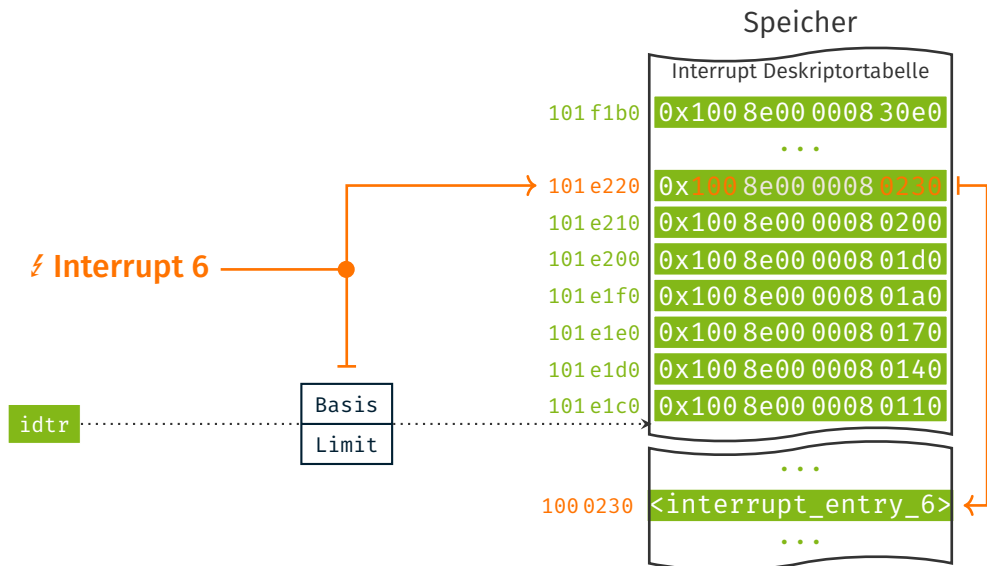
# Interrupt Deskriptor Tabelle (IDT)



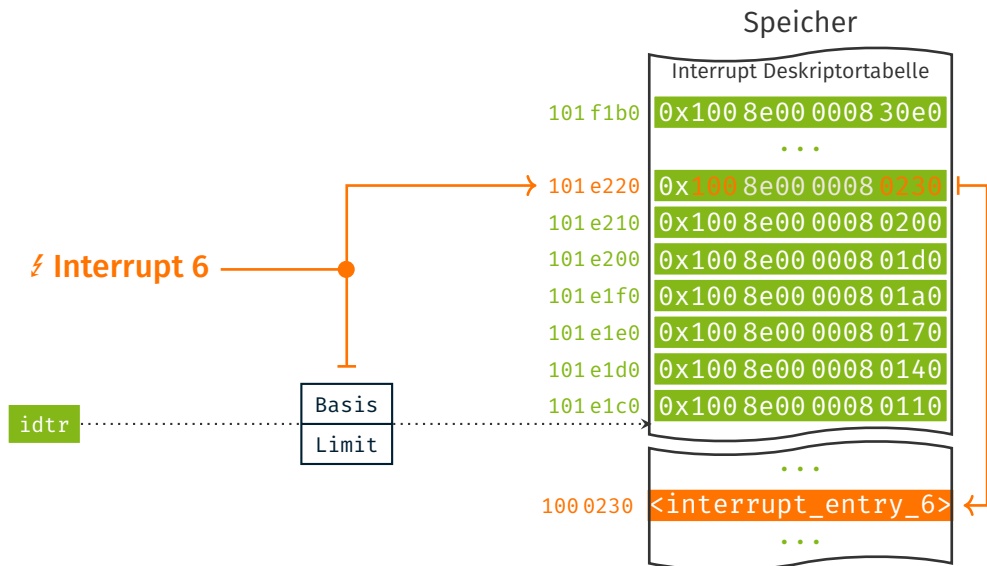
# Interrupt Deskriptor Tabelle (IDT)



# Interrupt Deskriptor Tabelle (IDT)



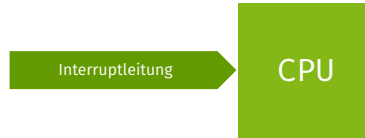
# Interrupt Deskriptor Tabelle (IDT)



# Externe Interrupts

---

# Unterbrechungen durch externe Geräte bei x86-CPUs



# Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren Geräten durch Programmable Interrupt Controller

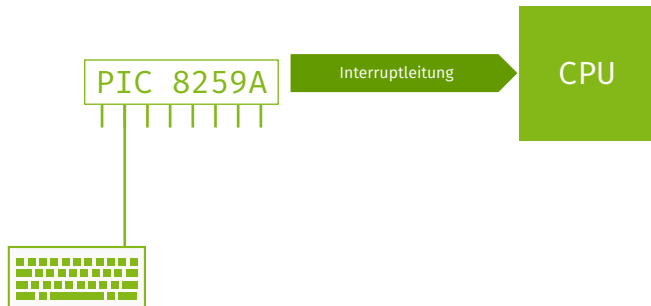
# Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

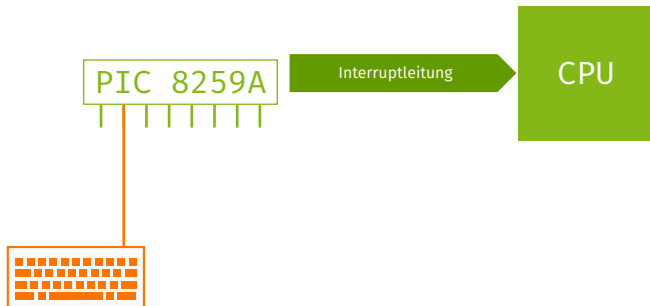


# Unterbrechungen durch externe Geräte bei x86-CPUs



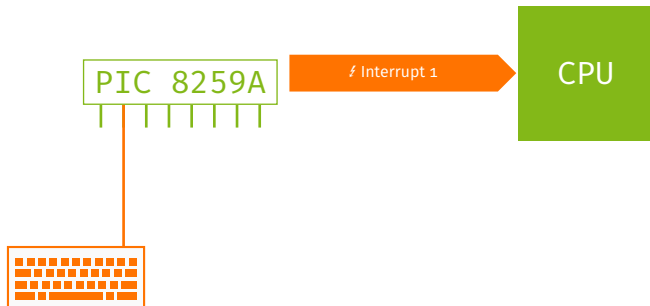
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

# Unterbrechungen durch externe Geräte bei x86-CPUs



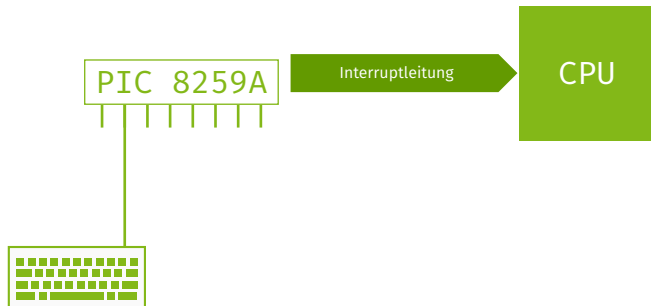
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

# Unterbrechungen durch externe Geräte bei x86-CPUs



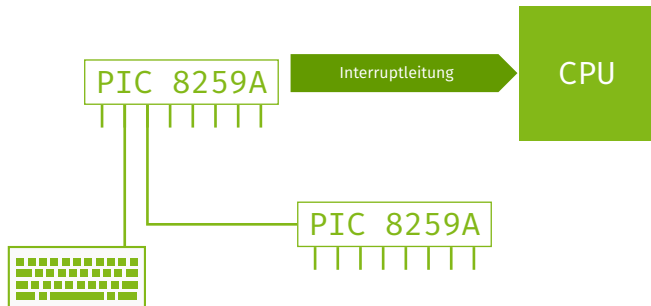
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

# Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

# Unterbrechungen durch externe Geräte bei x86-CPUs

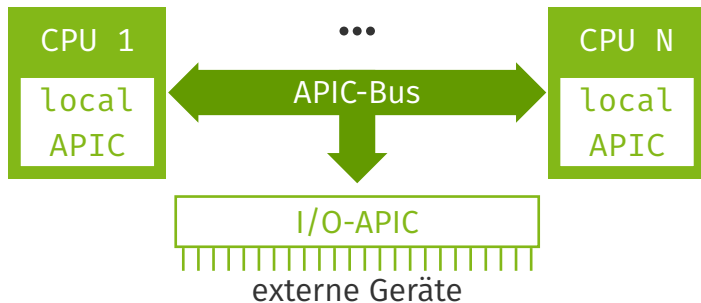


- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung

# Externe Interrupts mit dem APIC

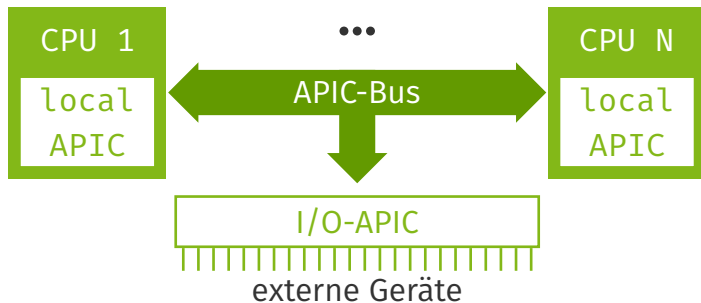
---

# Aufbau der APIC-Architektur



## Aufteilung in lokalen APIC und I/O APIC

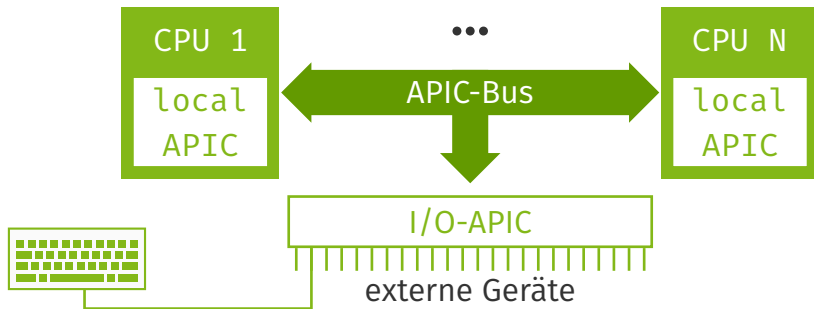
# Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

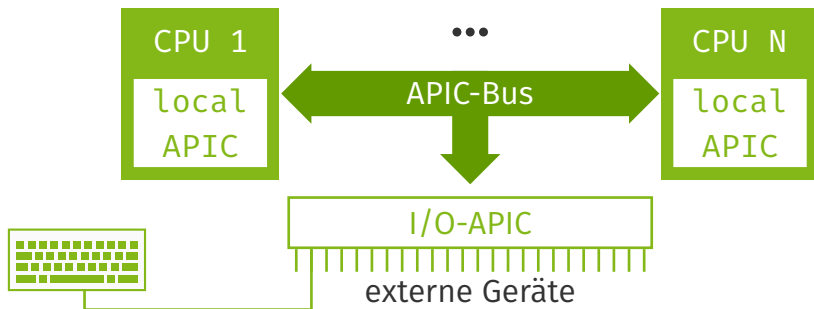


# Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

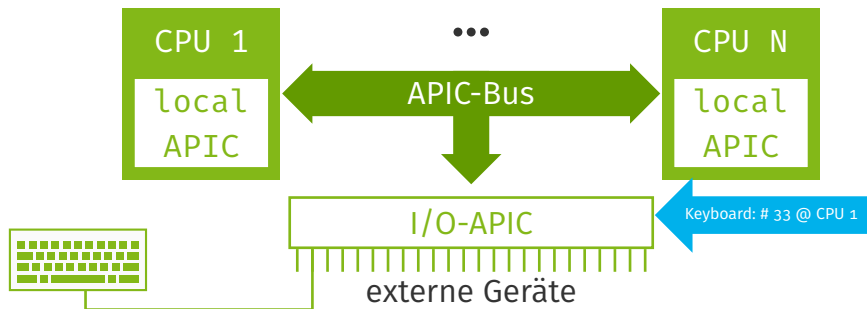
# Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

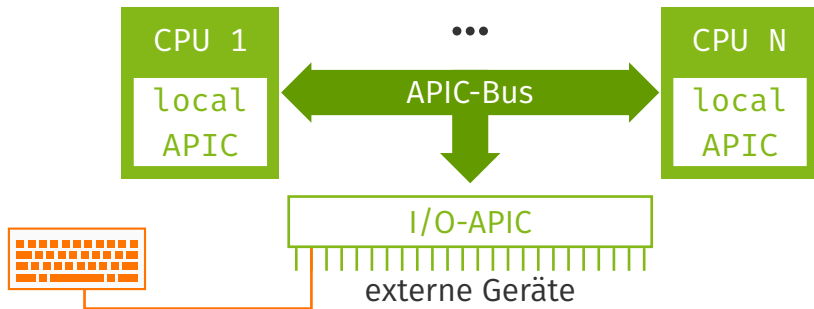
# Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

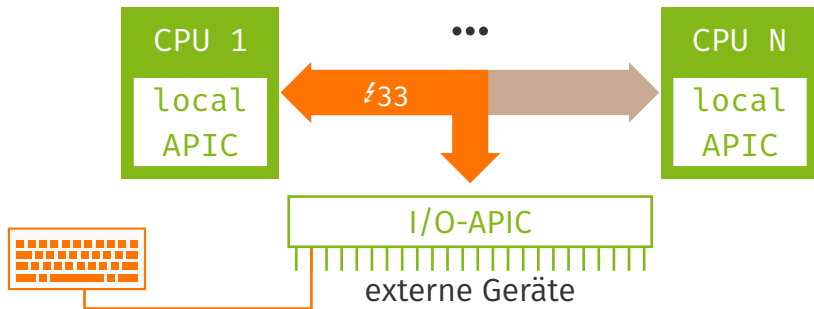
# Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

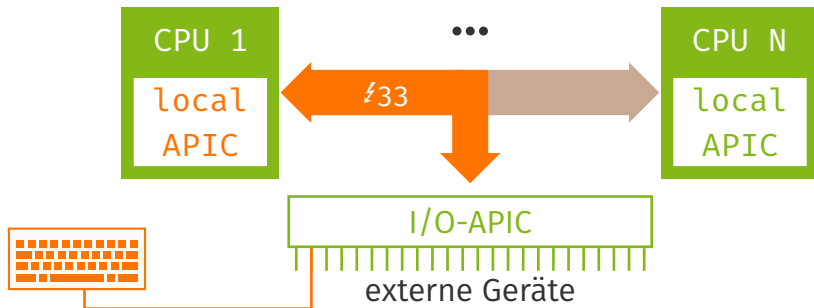
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

# Aufbau der APIC-Architektur



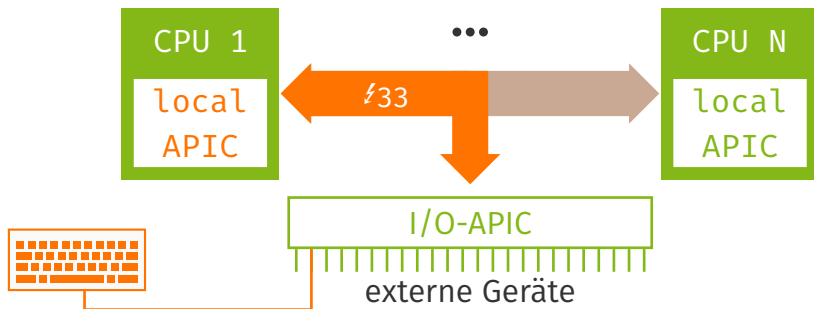
Interrupts werden zu Nachrichten auf dem APIC-Bus

# Aufbau der APIC-Architektur



Empfang durch Local APIC

# Aufbau der APIC-Architektur



## Empfang durch Local APIC

- Verbindet eine CPU mit dem APIC-Bus
- Liest Nachrichten vom APIC-Bus und unterbricht die CPU
- Muss Interrupts explizit quittieren (ACK)

# Programmierung des Intel I/O-APIC

Zugriff auf die internen Register über memory-mapped I/O

- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister

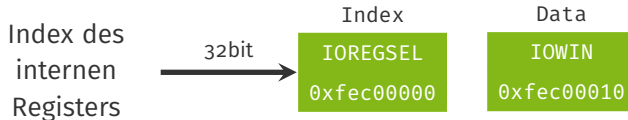
Index	Data
IOREGSEL	IOWIN
0xfec00000	0xfec00010



# Programmierung des Intel I/O-APIC

Zugriff auf die internen Register über memory-mapped I/O

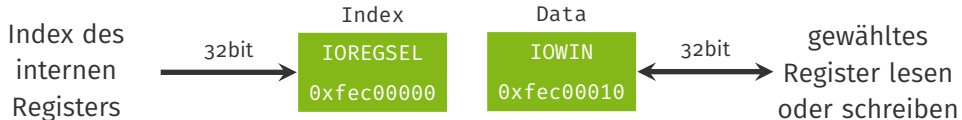
- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister



# Programmierung des Intel I/O-APIC

Zugriff auf die internen Register über memory-mapped I/O

- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister



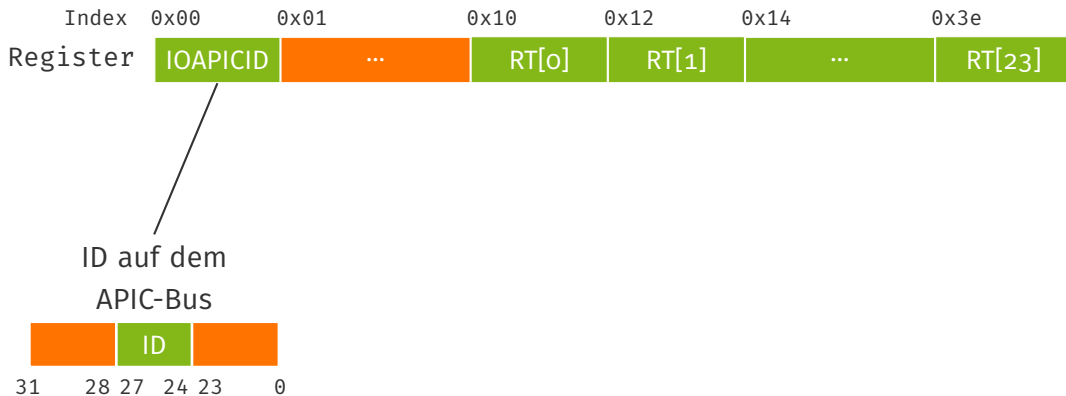
# Programmierung des Intel I/O-APIC (2)

## Interne Register des I/O-APICs

Index	0x00	0x01	0x10	0x12	0x14	0x3e
Register	IOAPICID	...	RT[0]	RT[1]	...	RT[23]

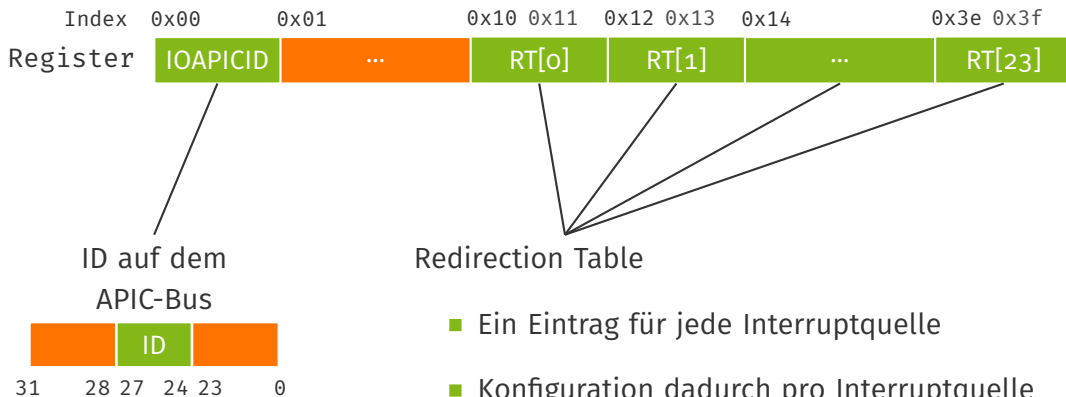
# Programmierung des Intel I/O-APIC (2)

## Interne Register des I/O-APICs

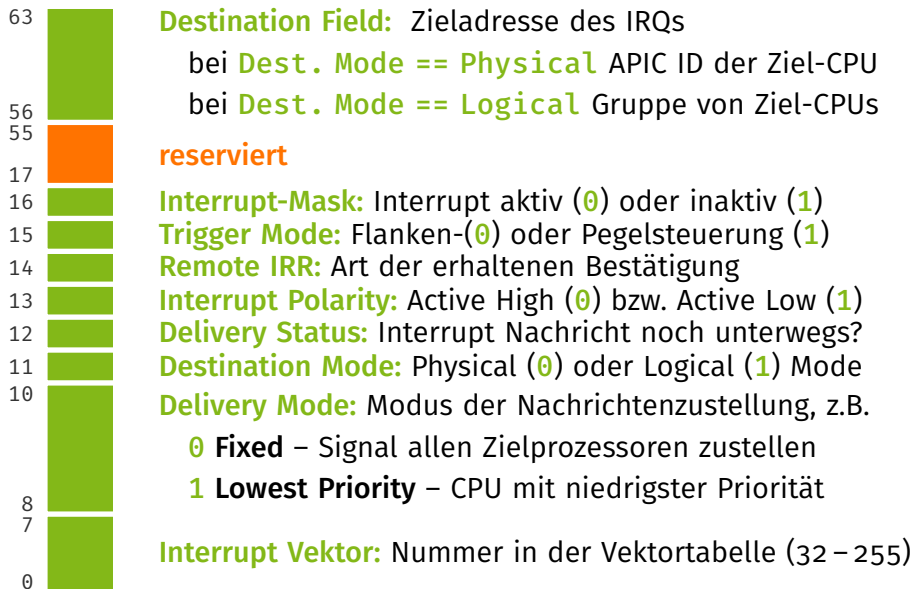


# Programmierung des Intel I/O-APIC (2)

## Interne Register des I/O-APICs



# Aufbau eines Redirection Table Eintrags



# Umsetzung in StuBS

---

# Programmierung des APIC Systems



**Wo ist welches Gerät angeschlossen?**

## Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!

## Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
- Steht in der Systemkonfiguration, heutzutage i.d.R. **ACPI** (Advanced Configuration and Power Interface)

## Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
- Steht in der Systemkonfiguration, heutzutage i.d.R. **ACPI** (Advanced Configuration and Power Interface)
- Bei uns stellt **APIC** die relevanten Teile dieser Informationen bereit

## Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
- Steht in der Systemkonfiguration, heutzutage i.d.R. **ACPI** (Advanced Configuration and Power Interface)
- Bei uns stellt **APIC** die relevanten Teile dieser Informationen bereit  
**APIC::getIOAPICSlot** liefert für jedes Gerät den Index in die Redirection Table (siehe enum Device in machine/apic.h)

## Wo ist welches Gerät angeschlossen?

- Das kann evtl. unterschiedlich sein von Rechner zu Rechner!
- Steht in der Systemkonfiguration, heutzutage i.d.R. **ACPI** (Advanced Configuration and Power Interface)
- Bei uns stellt **APIC** die relevanten Teile dieser Informationen bereit  
**APIC::getIOAPICSlot** liefert für jedes Gerät den Index in die Redirection Table (siehe enum Device in machine/apic.h)  
**APIC::getIOAPICID** liefert die ID des I/O-APICs

## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs

## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- **Ziel:** Gleichverteilung der Interrupts auf alle CPUs



## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- **Ziel:** Gleichverteilung der Interrupts auf alle CPUs
  - Priorität der Prozessoren im Local APIC fest auf 0 einstellen

## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- **Ziel:** Gleichverteilung der Interrupts auf alle CPUs
  - Priorität der Prozessoren im Local APIC fest auf 0 einstellen
  - Im I/O-APIC **Lowest Priority** als Delivery Mode verwenden

## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- **Ziel:** Gleichverteilung der Interrupts auf alle CPUs
  - Priorität der Prozessoren im Local APIC fest auf 0 einstellen
  - Im I/O-APIC **Lowest Priority** als Delivery Mode verwenden
  - Verwendung des **Logical Destination Mode**; bis zu 8 CPUs adressierbar

## Adressierung der APIC Nachrichten in StuBS

- Zusammenspiel mehrerer Faktoren
  - **Destination Mode**, **Destination Field** und **Delivery Mode** im I/O-APIC
  - Prozessor Priorität in den Local APICs der einzelnen CPUs
- **Ziel:** Gleichverteilung der Interrupts auf alle CPUs
  - Priorität der Prozessoren im Local APIC fest auf 0 einstellen
  - Im I/O-APIC **Lowest Priority** als Delivery Mode verwenden
  - Verwendung des **Logical Destination Mode**; bis zu 8 CPUs adressierbar
  - **Destination Field:** Bitmaske mit gesetztem Bit pro aktivierter CPU

# Redirection Table Einträge in StuBS

63	0x01 bzw. 0x0f	<b>Destination Field:</b> Zieladresse des IRQs bei <b>Dest. Mode == Physical</b> APIC ID der Ziel-CPU bei <b>Dest. Mode == Logical</b> Gruppe von Ziel-CPU
56 55	0	<b>reserviert</b>
17		
16	0/1	<b>Interrupt-Mask:</b> Interrupt aktiv (0) oder inaktiv (1)
15	0/1	<b>Trigger Mode:</b> Flanken-(0) oder Pegelsteuerung (1)
14	RO	<b>Remote IRR:</b> Art der erhaltenen Bestätigung
13	0	<b>Interrupt Polarity:</b> Active High (0) bzw. Active Low (1)
12	RO	<b>Delivery Status:</b> Interrupt Nachricht noch unterwegs?
11	1	<b>Destination Mode:</b> Physical (0) oder Logical (1) Mode
10		<b>Delivery Mode:</b> Modus der Nachrichtenzustellung, z.B. 0 <b>Fixed</b> – Signal allen Zielprozessoren zustellen 1 <b>Lowest Priority</b> – CPU mit niedrigster Priorität
8 7		
0		<b>Interrupt Vektor:</b> Nummer in der Vektortabelle (32 – 255)

# Zusammenfassendes Beispiel: Keyboard Interrupt in StuBS

---



- I/O APIC initialisieren



# Vorbereitung

- I/O APIC initialisieren
  - I/O APIC ID setzen

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)

# Vorbereitung

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren

# Vorbereitung

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen **interrupt\_entry** mit Aufruf zu **interrupt\_handler** schreiben [wird in der Vorgabe bereits erledigt]

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen **interrupt\_entry** mit Aufruf zu **interrupt\_handler** schreiben [wird in der Vorgabe bereits erledigt]
  - Eintragen in die **Interrupt Deskriptor Tabelle (IDT)** und diese in das Register **idtr** laden [ebenfalls erledigt]

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen **interrupt\_entry** mit Aufruf zu **interrupt\_handler** schreiben [wird in der Vorgabe bereits erledigt]
  - Eintragen in die **Interrupt Deskriptor Tabelle (IDT)** und diese in das Register **idtr** laden [ebenfalls erledigt]
  - Ereignisbehandlung in **interrupt\_handler** mittels **Plugbox**



- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen **interrupt\_entry** mit Aufruf zu **interrupt\_handler** schreiben [wird in der Vorgabe bereits erledigt]
  - Eintragen in die **Interrupt Deskriptor Tabelle (IDT)** und diese in das Register **idtr** laden [ebenfalls erledigt]
  - Ereignisbehandlung in **interrupt\_handler** mittels **Plugbox**
- Interrupts mit **Core::Interrupt::enable()** aktivieren



1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**

# Ablauf

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**

# Ablauf

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**
4. **CPU** führt Unterbrechungsbehandlung aus



1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**
4. **CPU** führt Unterbrechungsbehandlung aus
  - 4.1 Mittels Register **idtr** wird der entsprechende Eintrag in der **Interrupt Deskriptor Tabelle** ausgewählt und in die Einsprungsroutine gesprungen

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**
4. **CPU** führt Unterbrechungsbehandlung aus
  - 4.1 Mittels Register **idtr** wird der entsprechende Eintrag in der **Interrupt Deskriptor Tabelle** ausgewählt und in die Einsprungsroutine gesprungen
  - 4.2 Einsprungsroutine **interrupt\_entry\_33** sichert Register und ruft **interrupt\_handler** mit Parameter **vector = 33** auf

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**
4. **CPU** führt Unterbrechungsbehandlung aus
  - 4.1 Mittels Register **idtr** wird der entsprechende Eintrag in der **Interrupt Deskriptor Tabelle** ausgewählt und in die Einsprungsroutine gesprungen
  - 4.2 Einsprungsroutine **interrupt\_entry\_33** sichert Register und ruft **interrupt\_handler** mit Parameter **vector = 33** auf
  - 4.3 **interrupt\_handler** behandelt mittels **Plugbox** den Interrupt

1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
  - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
  - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**
4. **CPU** führt Unterbrechungsbehandlung aus
  - 4.1 Mittels Register **idtr** wird der entsprechende Eintrag in der **Interrupt Deskriptor Tabelle** ausgewählt und in die Einsprungsroutine gesprungen
  - 4.2 Einsprungsroutine **interrupt\_entry\_33** sichert Register und ruft **interrupt\_handler** mit Parameter **vector = 33** auf
  - 4.3 **interrupt\_handler** behandelt mittels **Plugbox** den Interrupt
5. **LAPIC** quittiert die Behandlung

## Aufgabe 2

---

# Aufgabe 2

## Lernziele

- Behandlung asynchroner Ereignisse
- Problematik und Schutz kritischer Abschnitte

## Aufgaben

- Konfiguration externer Geräte über I/O APIC
- Treiber für **Tastatur** und (*optional*) **Maus**
- **MPSTuBS**: Wechselseitiger Ausschluss (**Spinlock**)
- *Optional*: **GDB Stub**



- **Problem:** Tastatur-Interrupts in KVM nur auf Core 1





## KVM nutzt ggf. Vector Hashing

- **Problem:** Tastatur-Interrupts in KVM nur auf Core 1
- **Lösung:** Datei `/etc/modprobe.d/kvm_options.conf` editieren, Eintrag `options kvm vector_hashing=N` hinzufügen und System neu starten.
- Betrifft aber **nur** eure eigene Entwicklungsumgebung!

(weitere Details siehe FAQ auf der Webseite)

**Gibt es noch Fragen?**

---

**Abgabe der 2. Aufgabe bis Mittwoch, 15. November**