Übung zu Betriebssysteme

Interruptbehandlung

7. & 8. November 2018

Andreas Ziegler, Bernhard Heinloth, Christian Eichler

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

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Interrupts und Traps

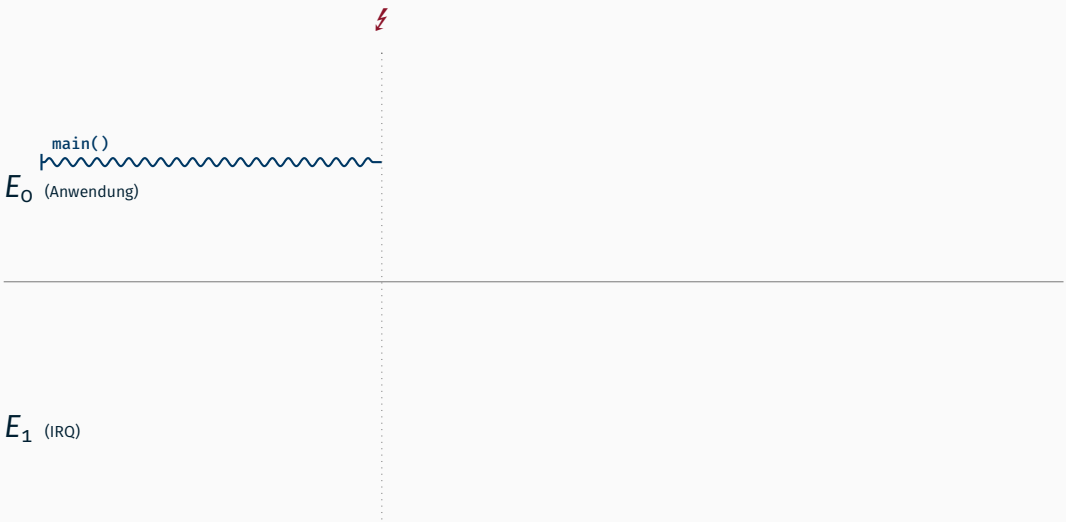

CPU


CPU

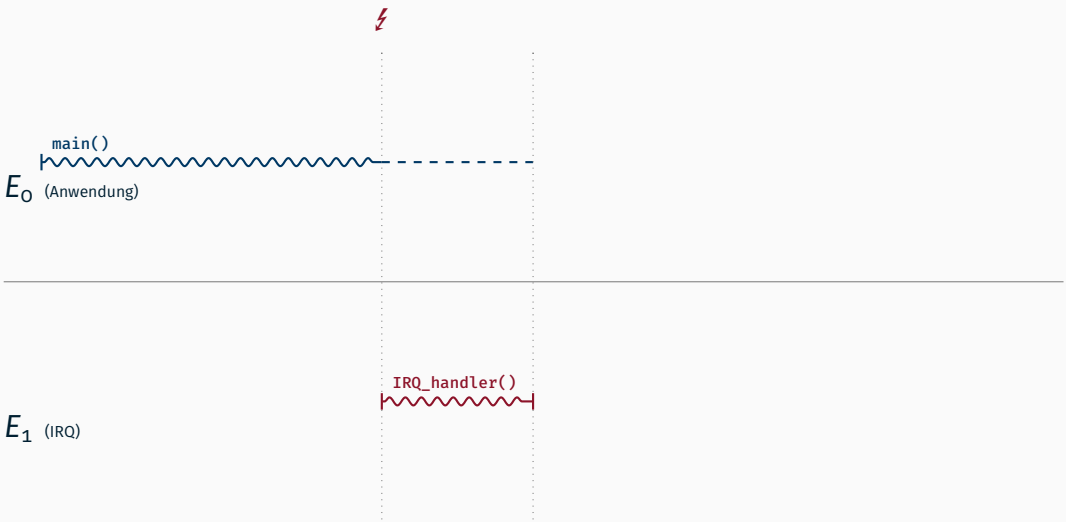
Unterbrechung (Anwendungssicht)

E_1 (IRQ)

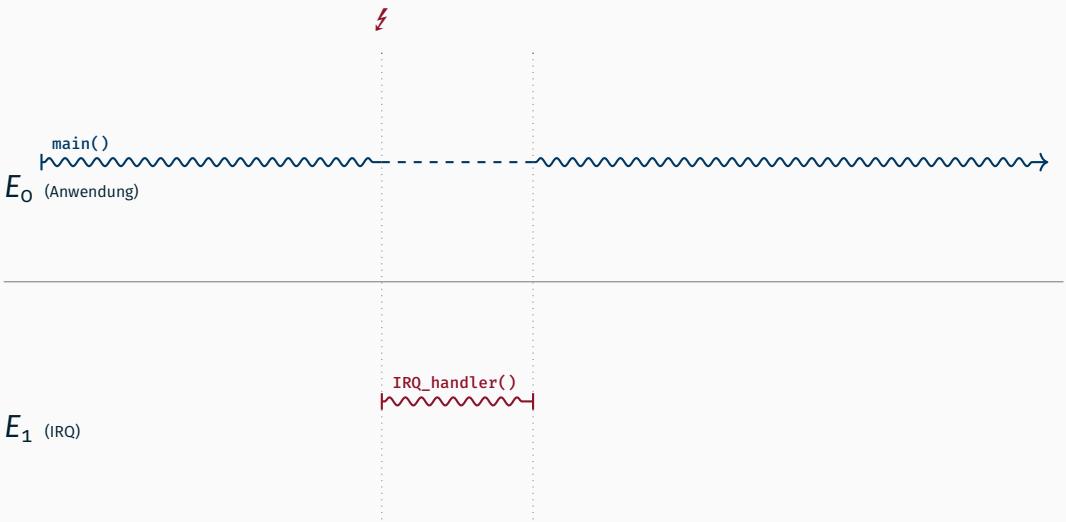
Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)

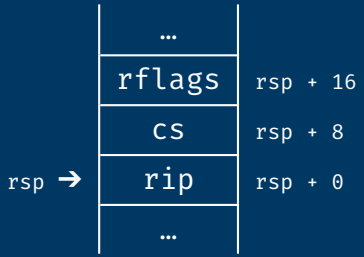


Minimaler zu sichernder Zustand?

Minimaler zu sichernder Zustand?

- CPU sichert automatisch

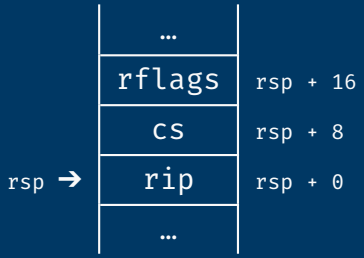
  rflags Condition Codes
  cs Aktuelles Code Segment
  rip Programmzeiger/Rücksprungadresse



Minimaler zu sichernder Zustand?

- CPU sichert automatisch

  rflags Condition Codes
  cs Aktuelles Code Segment
  rip Programmzeiger/Rücksprungadresse



- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iretq`

Unterbrechungsbehandlung

```
;; Assembler
irq_handler:
  ;; Behandle IRQ


  iretq
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

```
// C++

void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

```
// C++

void guardian()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ make
LD          .build/system64
.build/boot/interrupts.asm.o: in function `irq_handler':
boot/interrupts.asm:(.text+0x16): undefined reference to `guardian'
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

```
// C++

void guardian()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ objdump -d .build/guard/guardian.o
Disassembly of section .text:

0000000000000000 <_Z8guardianv>:
   0:   c3                      retq
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

```
// C++ (mit C Linkage)
extern "C"
void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
irq_handler:
  ;; Behandle IRQ
  ;; in Hochsprache
  call guardian
  iretq
```

```
// C++ (mit C Linkage)
extern "C"
void guardian()
{
    // Magie.
}
```

```
heinloth:~/oostubs$ objdump -d .build/guard/guardian.o
Disassembly of section .text:

0000000000000000 <guardian>:
   0:  f3 c3                    repz ret
```

Was ist mit den restlichen Registern?

Was ist mit den restlichen Registern?

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

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

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

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

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 der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert

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 der Register 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

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`

Interrupt-Handler müssen auch flüchtige Register sichern!

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler
irq_handler:




  call guardian




  iretq
```

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++


extern "C"
void guardian()
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
struct irq_context {









} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
struct irq_context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;



} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Parameterübergabe

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)

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
struct irq_context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;



} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
struct irq_context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;



} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
struct irq_context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
} __attribute__((packed));

extern "C"
void guardian(irq_context* c)
{
    // Magie.
}
```


CPU

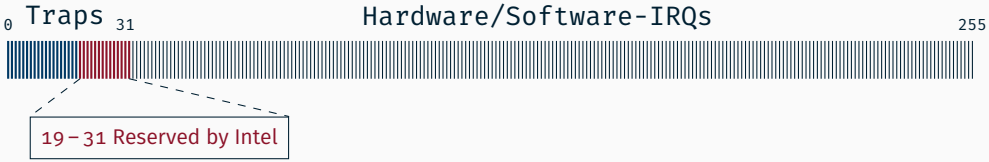

0-255
CPU

Interruptvektoren (x86/x64)

Interruptvektoren (x86/x64)

0	**Division-by-Zero**
1	Debug Exception
2	Non-Maskable Interrupt(NMI)
3	**Breakpoint (INT 3)**
4	Overflow Exception
5	Bound Exception
6	**Invalid Opcode**
7	FPU not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	**General Protection Fault**
14	**Page Fault**
15	*Reserved*
16	Floating-Point Error
17	Alignment Check
18	Machine Check

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` (**cl**ear **i**nterrupt flag) Interruptleitung sperren
- `sti` (**s**e**t** **i**nterrupt flag) Interruptleitung freigeben

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c

  )
{
    // Magie:









}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c

  )
{
    // Magie:
    switch (      ){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c

  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp


  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, 6
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, 6
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
// C++
extern "C"
void guardian(
    irq_context* c,
    uint32_t vector
  )
{
    // Magie:
    switch (vector){
        case KBD:
            kbd.magic();
            break;
        case TMR:
            tmr.magic();
            break;
    }
}
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, 6
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, 6
  call 0x10070f0 <guardian>
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Unterbrechungsbehandlung (Binden)

```
;; Assembler
irq_handler_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, 6
  call 0x10070f0 <guardian>
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>
```

Unterbrechungsbehandlung (Binden)

```
Maschinencode


50
51

41 53

48 89 e7

be 06 00 00 00
e8 a6 6e 00 00

41 5b

59
58
48 cf
```

```
01 heinloth:~/oostubs$ make
02 ASM   boot/interrupts.asm
03 CXX   guard/guardian.cc
04 LD    .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>
```

Unterbrechungsbehandlung (Binden)

```
Speicher     Maschinencode
addresse

1000230:     50
1000231:     51

100023b:     41 53

100023d:     48 89 e7

1000240:     be 06 00 00 00
1000245:     e8 a6 6e 00 00

100024a:     41 5b

1000255:     59
1000256:     58
1000257:     48 cf
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>
```

Unterbrechungsbehandlung (Binden)

```
Speicher     Maschinencode
addresse

1000230:     50
1000231:     51

100023b:     41 53

100023d:     48 89 e7

1000240:     be 06 00 00 00
1000245:     e8 a6 6e 00 00

100024a:     41 5b

1000255:     59
1000256:     58
1000257:     48 cf
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>

100 0230 <irq_handler_6>
```

Unterbrechungsbehandlungen

```
%macro IRQ 1
align 8
irq_handler_%1:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, %1
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
%endmacro
```

```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>

100 0230 <irq_handler_6>
```

Unterbrechungsbehandlung

```
%macro IRQ 1
align 8
irq_handler_%1:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Interruptnummer
  mov rsi, %1
  call guardian
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
%endmacro
```

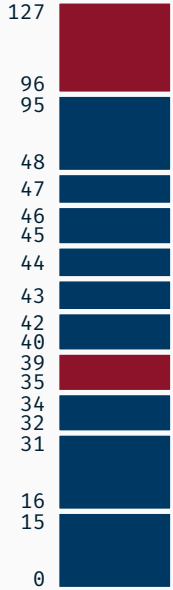
```
01 heinloth:~/oostubs$ make
02 ASM  boot/interrupts.asm
03 CXX  guard/guardian.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <guardian>
         ...
100 0230 <irq_handler_6>
100 0200 <irq_handler_5>
100 01d0 <irq_handler_4>
100 01a0 <irq_handler_3>
100 0170 <irq_handler_2>
100 0140 <irq_handler_1>
100 0110 <irq_handler_0>
```

Woher weiß die CPU wo die entsprechende Unterbrechungsbehandlung liegt?

Interrupt Deskriptor



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_handler_6`)

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`)?

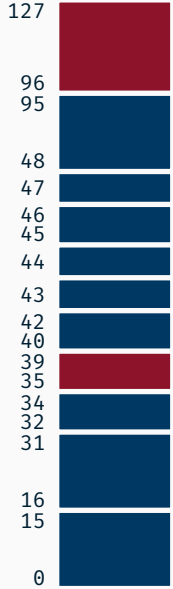
Unused – muss `0` sein

Interrupt Stack Table

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

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

Interrupt Deskriptor (Beispiel)



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_handler_6`)

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

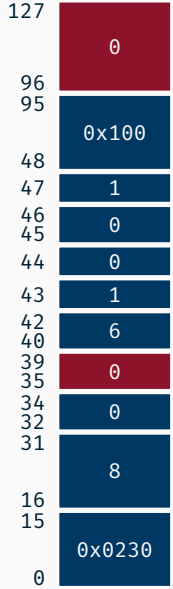
Interrupt Stack Table

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

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

für **`100 0230` <irq_handler_6>**

Interrupt Deskriptor (Beispiel)



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_handler_6`)

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

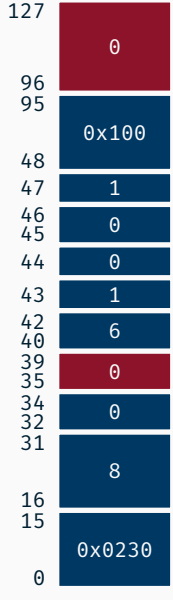
Interrupt Stack Table

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

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

für **`100 0230` <irq_handler_6>**

Interrupt Deskriptor (Beispiel)



Unused – muss `0` sein

Offset (high): *oberer* Teil der Einsprungsaddresse für die Interruptbehandlung (z.B. `irq_handler_6`)

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

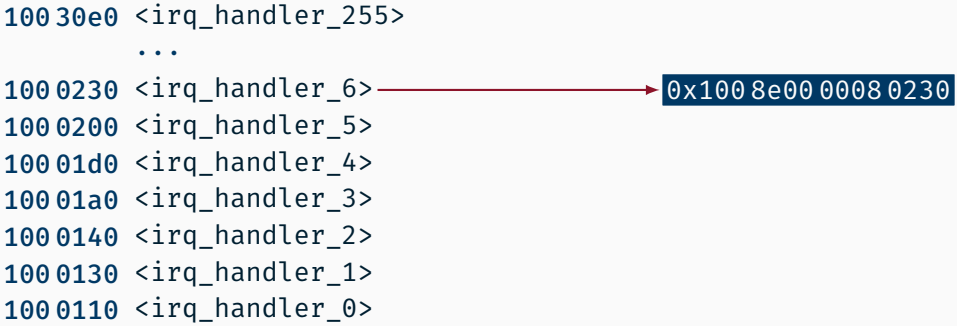
Interrupt Stack Table

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

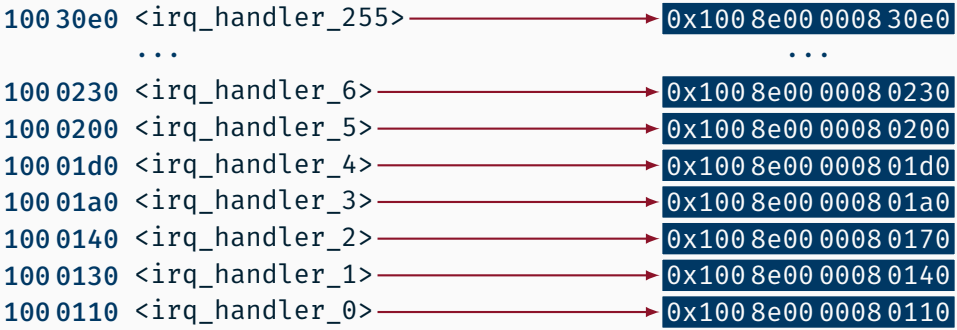
Offset (low): *unterer* Teil der Einsprungsaddresse für die Interruptbehandlung

für `100 0230 <irq_handler_6>` → `0x100 8e00 0008 0230`

Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

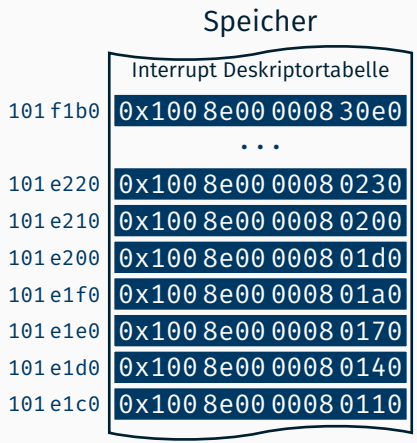


Interrupt Deskriptor Tabelle (IDT)

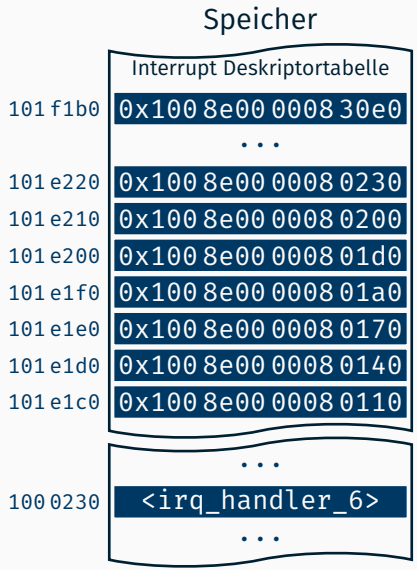
```
100 30e0 <irq_handler_255>
         ...
100 0230 <irq_handler_6>
100 0200 <irq_handler_5>
100 01d0 <irq_handler_4>
100 01a0 <irq_handler_3>
100 0140 <irq_handler_2>
100 0130 <irq_handler_1>
100 0110 <irq_handler_0>
```

Interrupt Deskriptor Tabelle (IDT)

```
100 30e0 <irq_handler_255>
         ...
100 0230 <irq_handler_6>
100 0200 <irq_handler_5>
100 01d0 <irq_handler_4>
100 01a0 <irq_handler_3>
100 0140 <irq_handler_2>
100 0130 <irq_handler_1>
100 0110 <irq_handler_0>
```

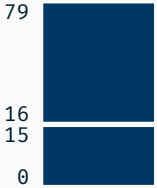


Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

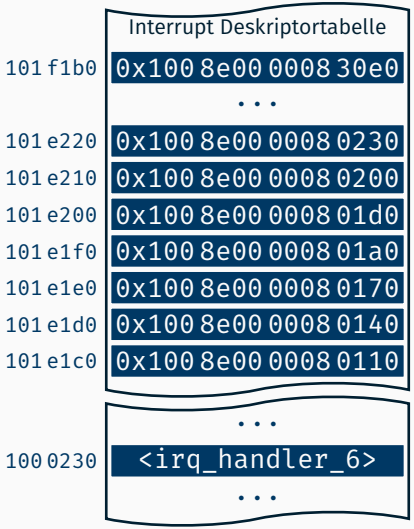
IDT-Register `idtr`



Basis: Startadresse der IDT

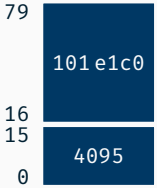
Limit: Bytes *Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

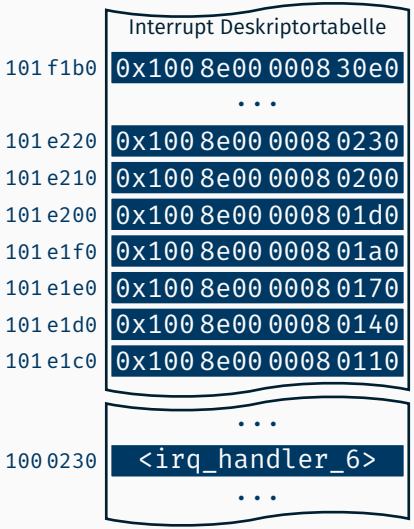
IDT-Register `idtr`



Basis: Startadresse der IDT

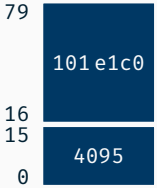
Limit: Bytes *Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



Basis: Startadresse der IDT

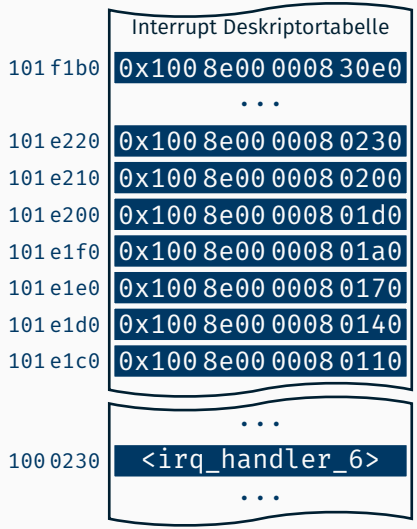
Limit: Bytes *Einträge * 16 - 1*

Instruktionen:

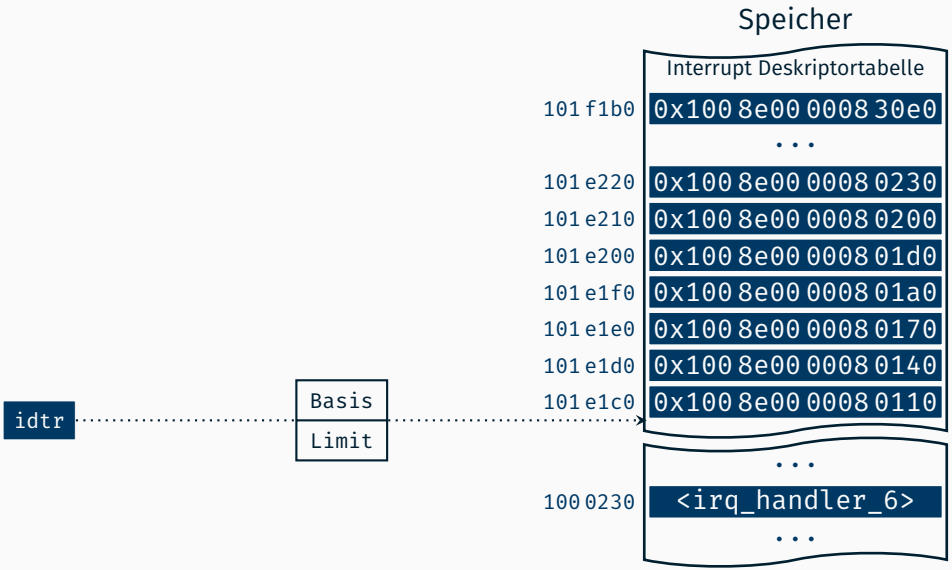
`lidt` in Register laden

`sidt` aus Register lesen

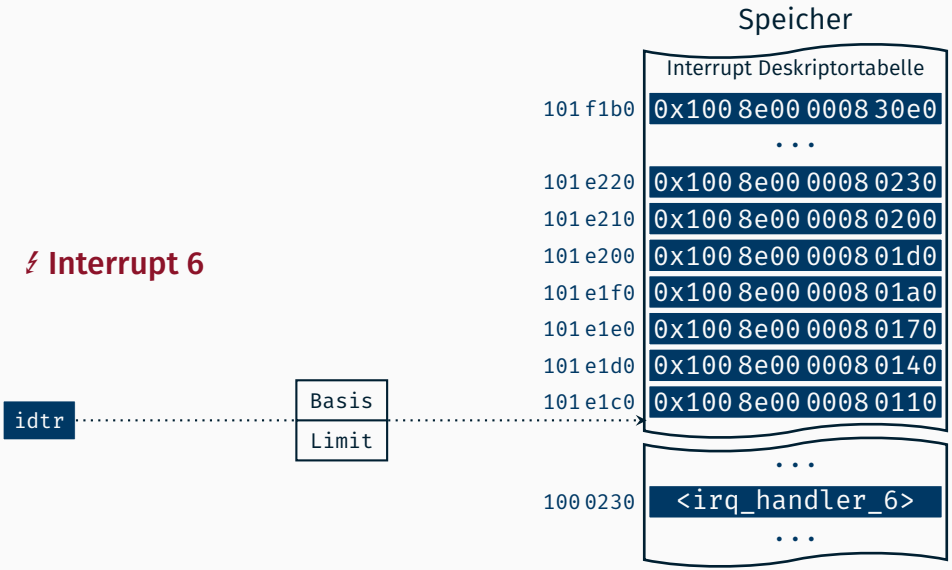
Speicher



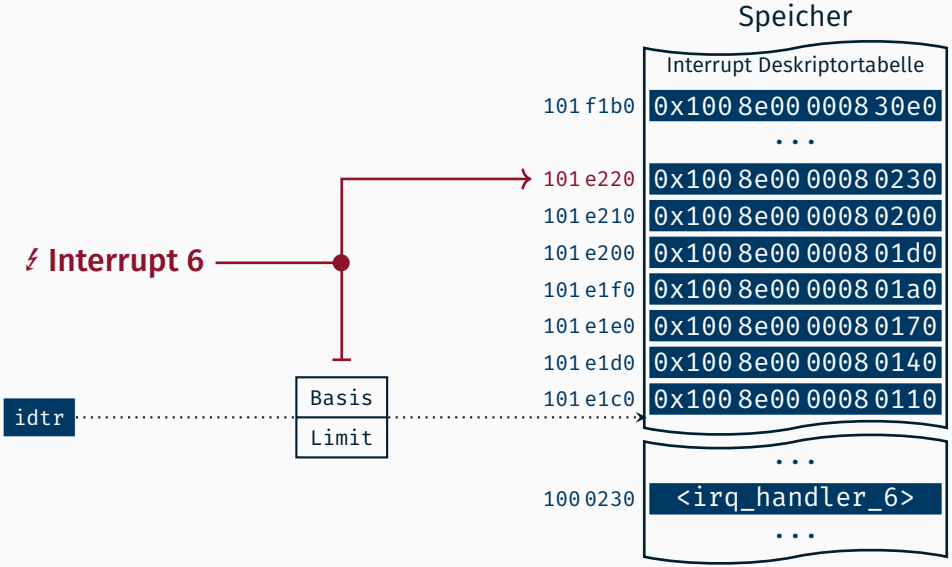
Interrupt Deskriptor Tabelle (IDT)



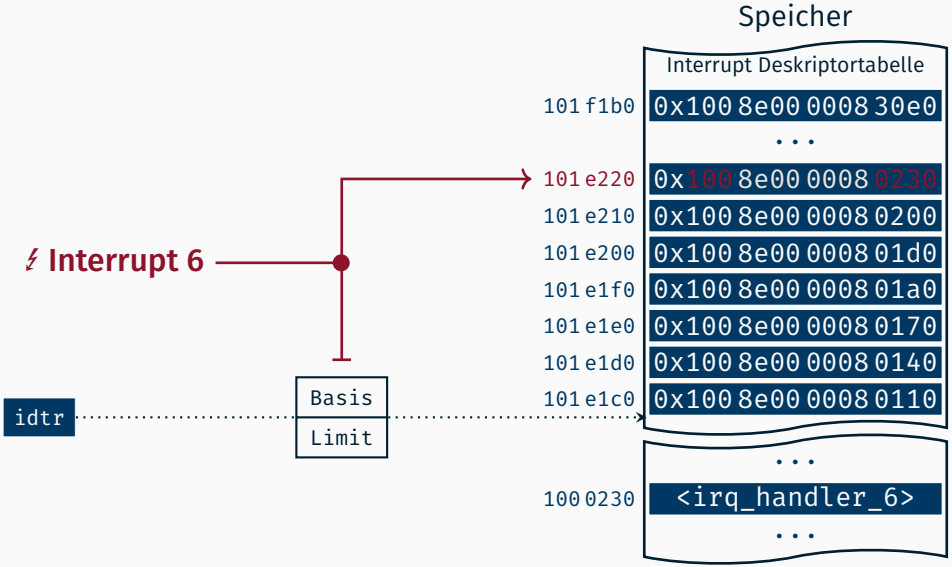
Interrupt Deskriptor Tabelle (IDT)



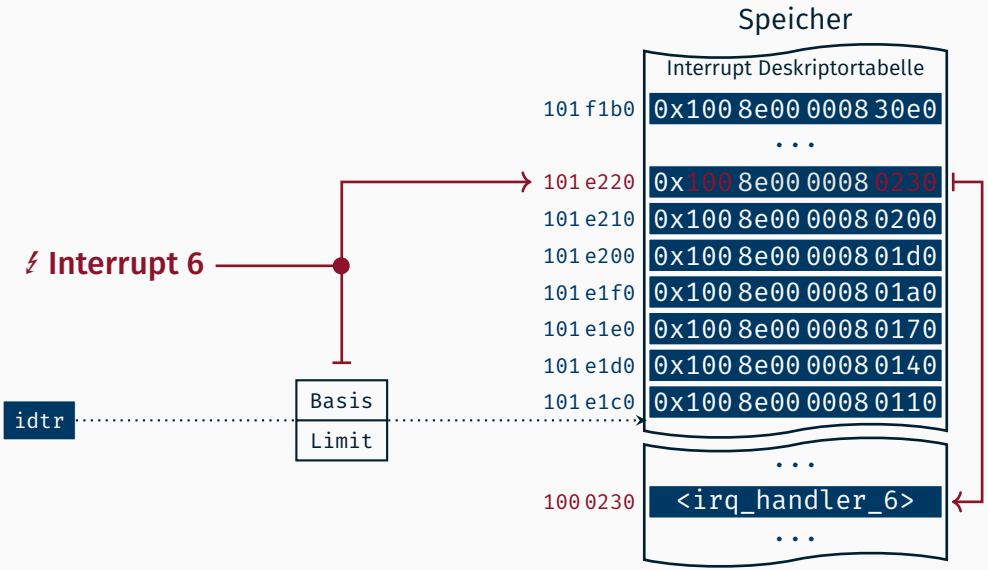
Interrupt Deskriptor Tabelle (IDT)



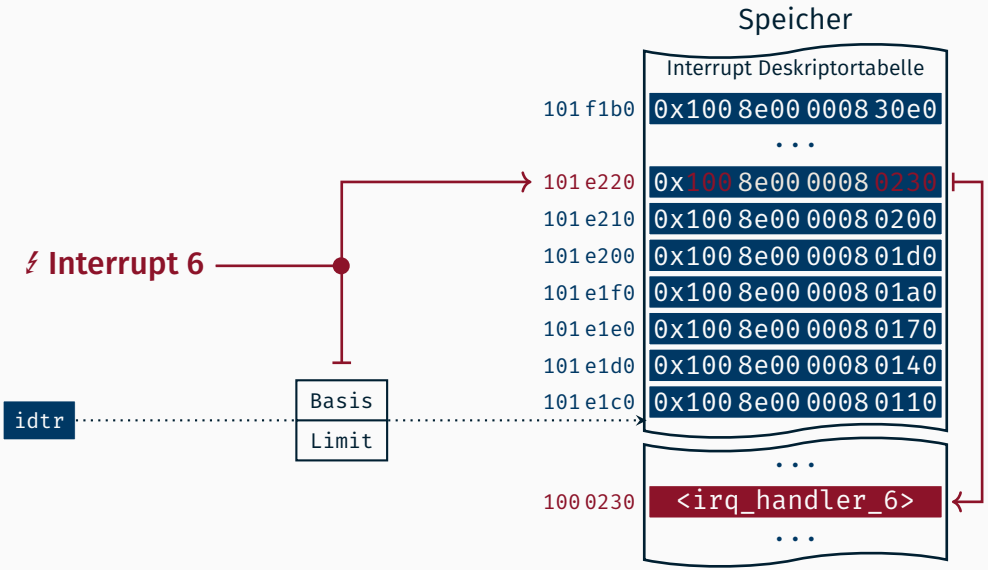
Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)



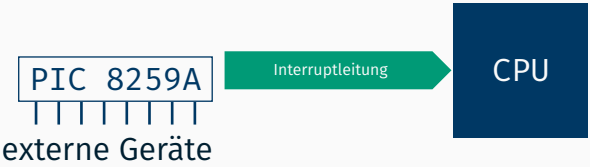
Interrupt Deskriptor Tabelle (IDT)



Externe Interrupts

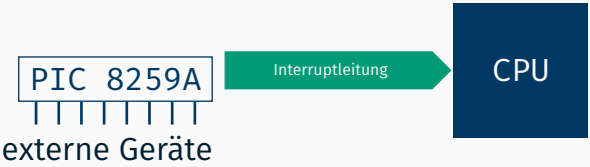
Unterbrechungen durch externe Geräte bei x86-CPUs

Unterbrechungen durch externe Geräte bei x86-CPUs



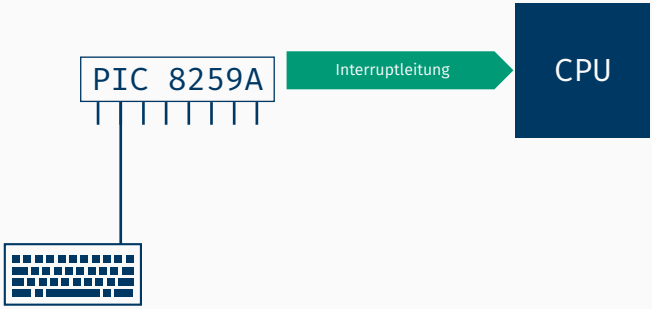
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller

Unterbrechungen durch externe Geräte bei x86-CPUs



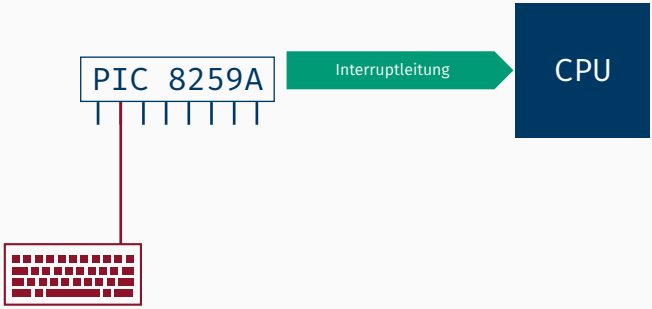
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



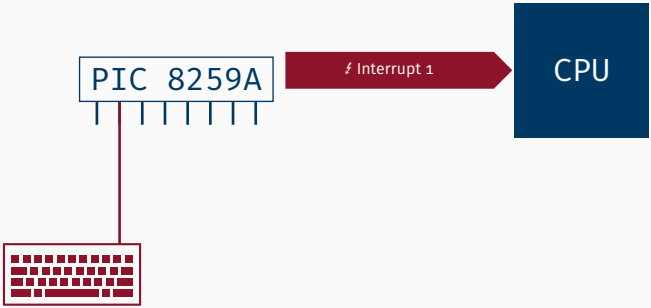
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



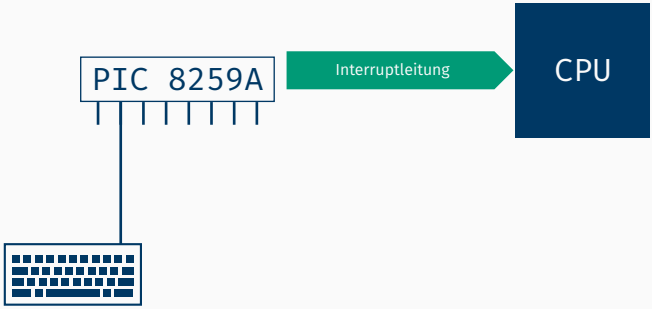
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



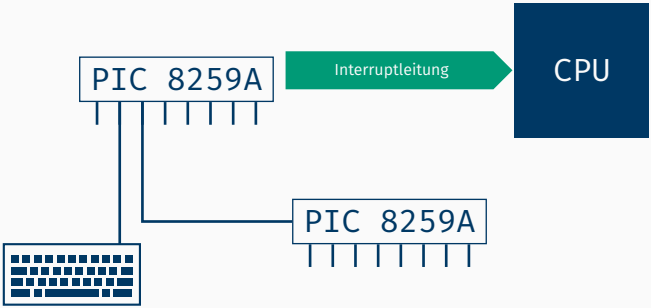
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



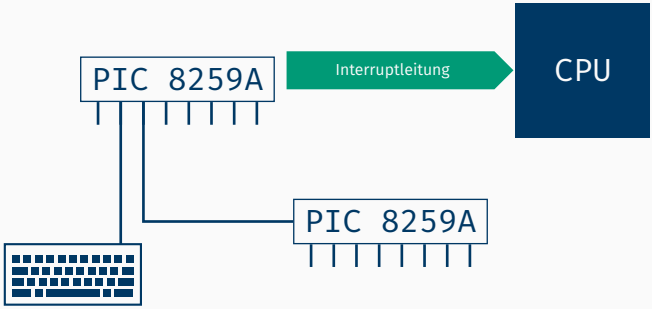
- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung

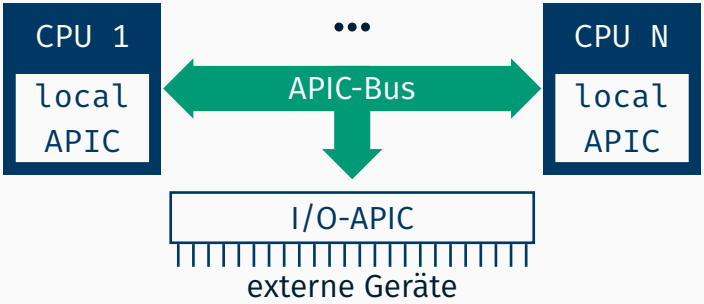
Unterbrechungen durch externe Geräte bei x86-CPUs



- Anschluss von mehreren externen Geräten durch [Programmable Interrupt] Controller
  - feste Prioritätenreihenfolge
  - Interruptnummern bedingt änderbar (Vielfaches von 8)
  - Konfiguration über IO-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung
- Nicht für Mehrprozessorsysteme geeignet

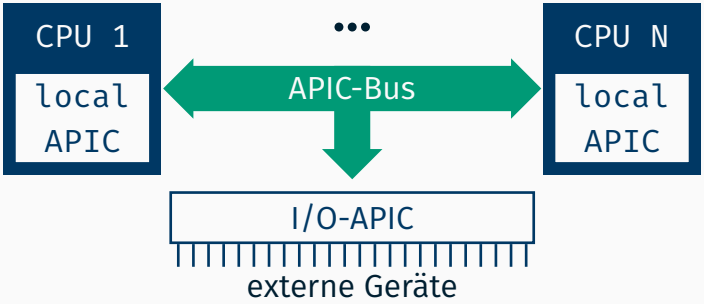
Externe Interrupts mit dem APIC

Aufbau der APIC-Architektur



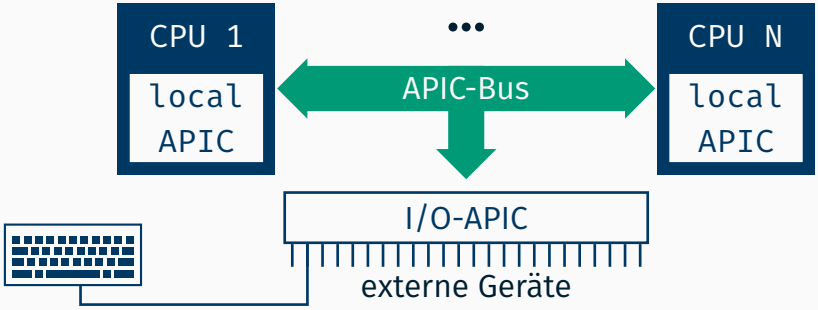
Aufteilung in lokalen APIC und I/O APIC

Aufbau der APIC-Architektur



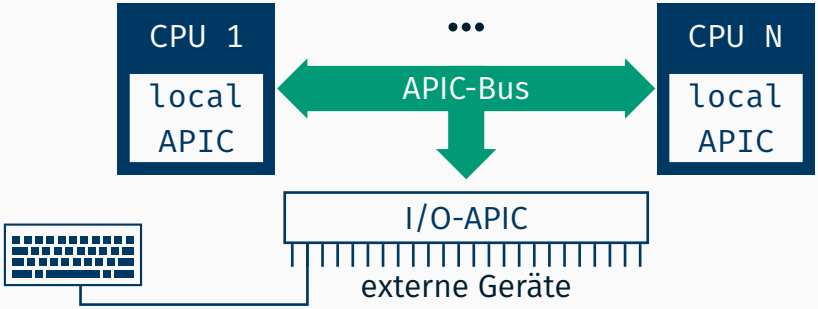
I/O-APIC dient zum Anschluss der Geräte

Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

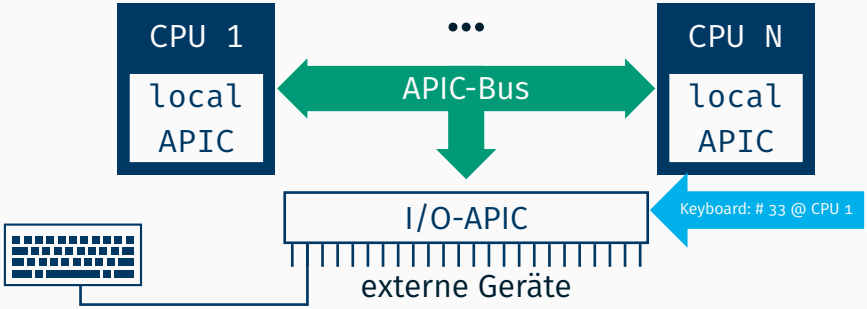
Aufbau der APIC-Architektur



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

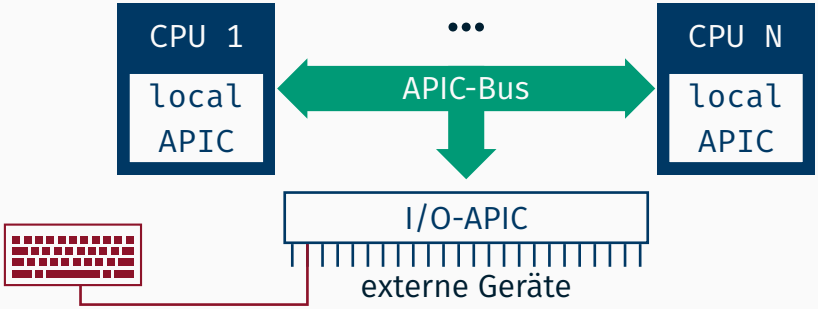
Aufbau der APIC-Architektur



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

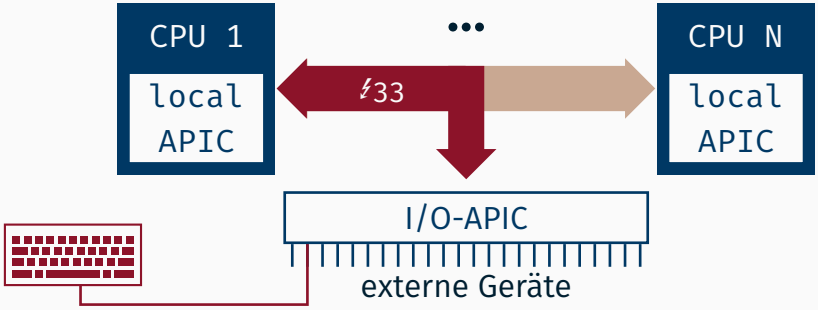
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

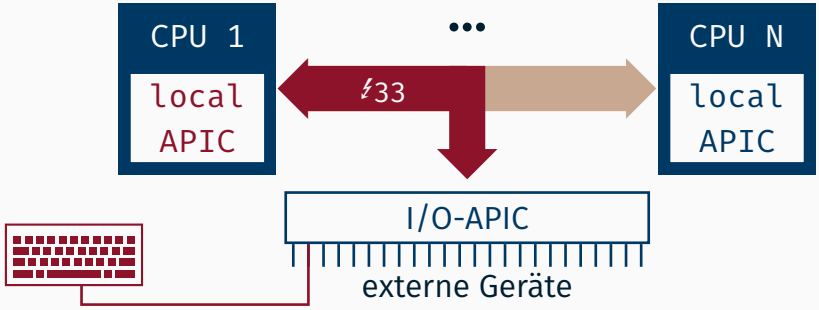
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

Aufbau der APIC-Architektur



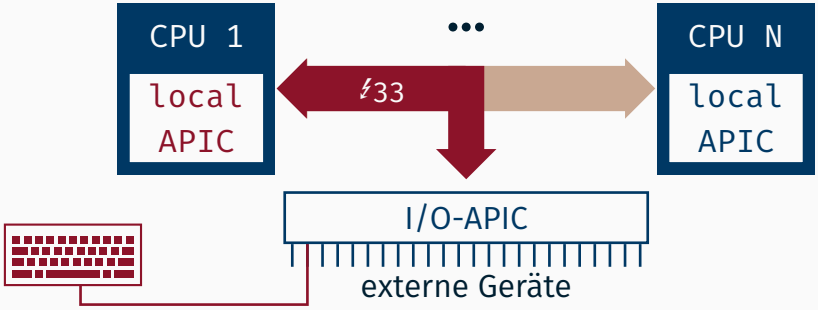
Interrupts werden zu Nachrichten auf dem APIC-Bus

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

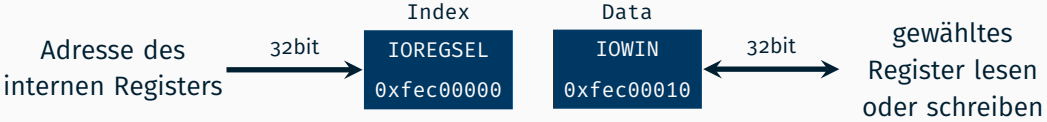
Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe

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

Programmierung des Intel I/O-APIC

Zugriff auf die internen Register über memory-mapped Ein-/Ausgabe

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



Programmierung des Intel I/O-APIC (2)

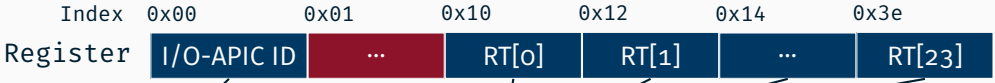
Interne Register des I/O-APICs

Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

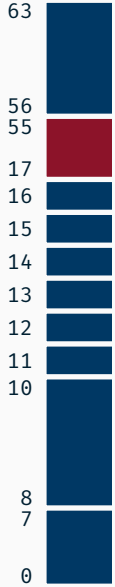


ID auf dem APIC-Bus

Redirection Table

- Ein Eintrag für jede Interruptquelle
- Konfiguration dadurch pro Interruptquelle
- Zwei interne Register pro Eintrag (64bit)

Aufbau eines Redirection Table Eintrags



Destination Field: Zieladresse des IRQs

bei `Dest. Mode == Physical` APIC ID der Ziel-CPU

bei `Dest. Mode == Logical` Gruppe von Ziel-CPUs

reserviert

Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)
Trigger Mode: Flanken-(`0`) oder Pegelsteuerung (`1`)
Remote IRR: Art der erhaltenen Bestätigung
Interrupt Polarity: Active High (`0`) bzw. Active Low (`1`)
Delivery Status: Interrupt Nachricht noch unterwegs?
Destination Mode: Physical (`0`) oder Logical (`1`) Mode
Delivery Mode: Modus der Nachrichtenzustellung, z.B.

- `0` **Fixed** – Signal allen Zielprozessoren zustellen
- `1` **Lowest Priority** – CPU mit niedrigster Priorität

Interrupt Vektor: Nummer in der Vektortabelle (32 – 255)

Programmierung des APIC Systems

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)
- `System` stellt die relevanten Teile diese Informationen bereit

  `System::getIOAPICID` liefert die ID des I/O-APICs

  `System::getIOAPICSlot` liefert für jedes Gerät den Index in die Redirection Table (siehe `enum System::Device` in `machine/apicsystem.h`)

Programmierung des APIC Systems

Adressierung der APIC Nachrichten in OOStuBS/MPStuBS

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

Programmierung des APIC Systems

Adressierung der APIC Nachrichten in OOStuBS/MPStuBS

- 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 OOStuBS/MPStuBS

Destination Field: Zieladresse des IRQs

bei `Dest. Mode == Physical` APIC ID der Ziel-CPU

bei `Dest. Mode == Logical` Gruppe von Ziel-CPUs

reserviert

Interrupt-Mask: Interrupt aktiv (`0`) oder inaktiv (`1`)

Trigger Mode: Flanken-(`0`) oder Pegelsteuerung (`1`)

Remote IRR: Art der erhaltenen Bestätigung

Interrupt Polarity: Active High (`0`) bzw. Active Low (`1`)

Delivery Status: Interrupt Nachricht noch unterwegs?

Destination Mode: Physical (`0`) oder Logical (`1`) Mode

Delivery Mode: Modus der Nachrichtenzustellung, z.B.

- `0` **Fixed** – Signal allen Zielprozessoren zustellen
- `1` **Lowest Priority** – CPU mit niedrigster Priorität

Interrupt Vektor: Nummer in der Vektortabelle (32 – 255)

(RO: Read Only)

Zusammenfassendes Beispiel: Keyboard Interrupt in StuBS

Vorbereitung

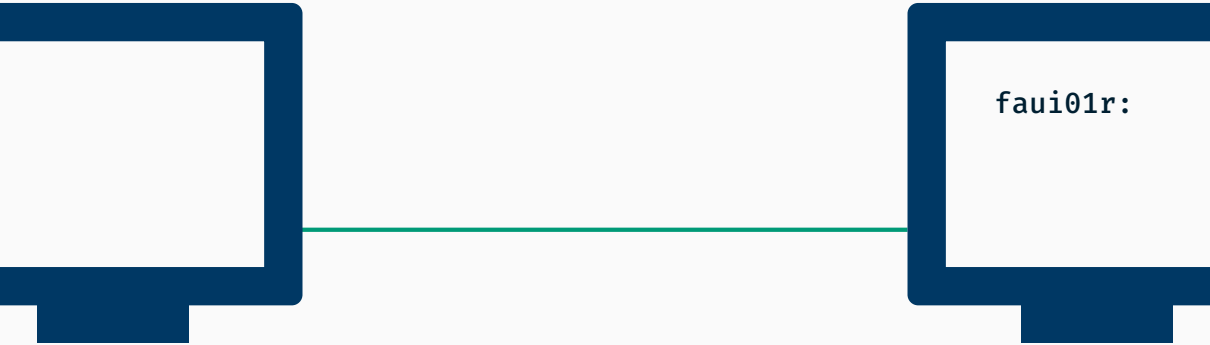
- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden und den entsprechenden Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutine `irq_handler` mit Aufruf zu `guardian` schreiben [wird in der Vorgabe bereits erledigt]
  - Eintragen in die **Interrupt Description Table** (`idt`) und diese in das Register `idtr` laden [ebenfalls erledigt]
  - Behandlung in `guardian` mittels **Plugbox**
- Interrupts mit `CPU::Interrupt::enable()` aktivieren

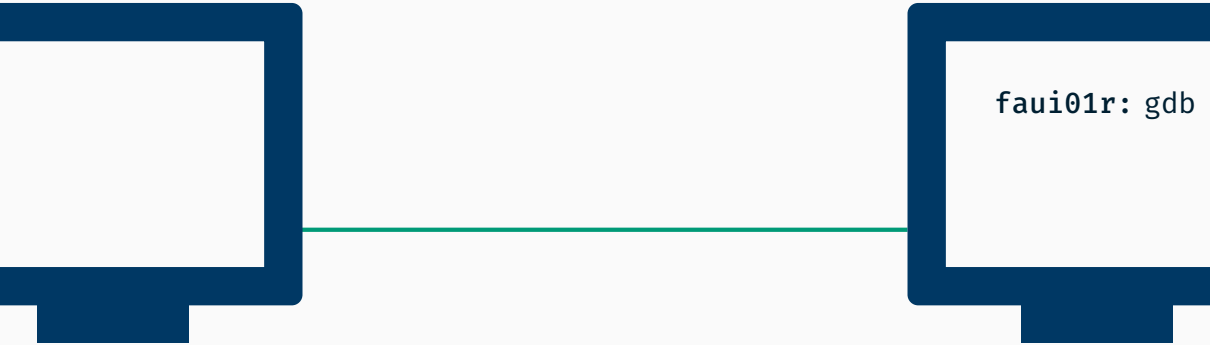
Ablauf

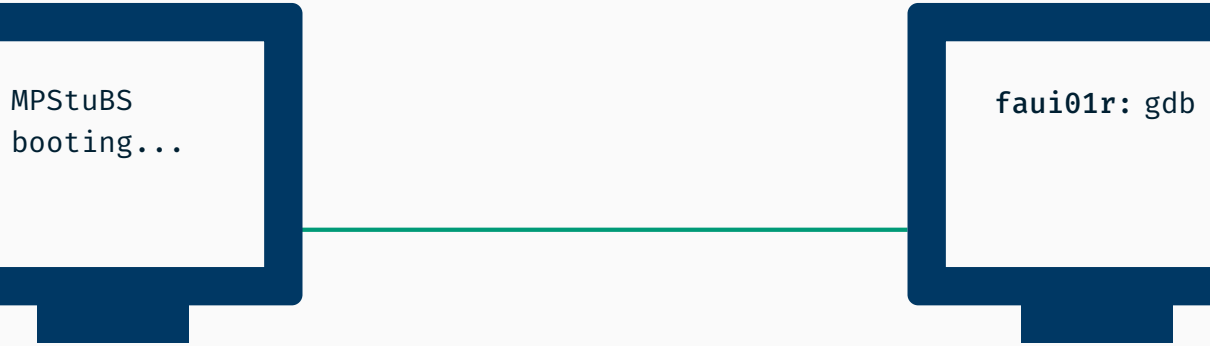
1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet seriell (via **PS/2**) an Tastaturcontroller
2. **Tastaturcontroller** aktiviert Interruptleitung zu **I/O APIC**
   2.1 Anhand der **Redirection Table** wird Aktion gewählt
   2.2 Nachricht auf **APICBUS** mit Interruptnr. `33` und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APICBUS** und unterbricht CPU
4. **CPU** führt Unterbrechungsbehandlung aus
   4.1 Mittels Register `idtr` wird der entsprechende Eintrag in der **Interrupt Description Table** ausgewählt und in die Einsprungsroutine gesprungen
   4.2 Einsprungsroutine `irq_handler_33` sichert Register und ruft `guardian` auf
   4.3 `guardian` behandelt mittels **Plugbox** den Interrupt
5. **LAPIC** quittiert die Behandlung

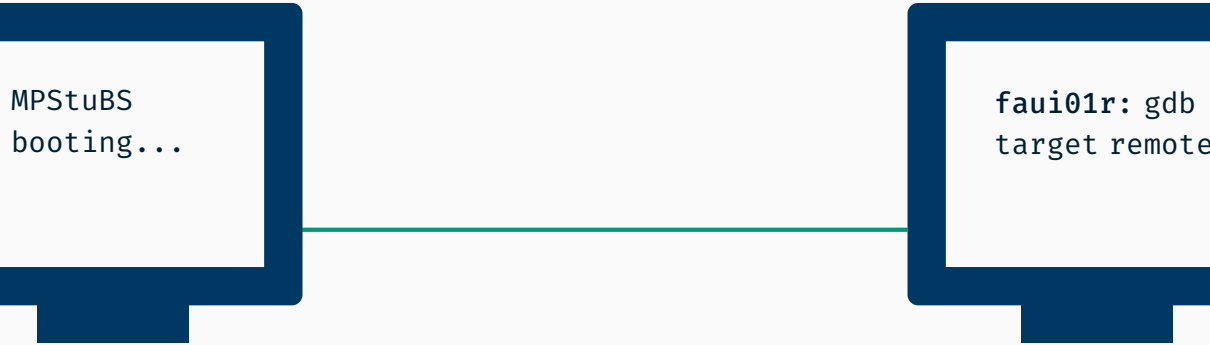
Remotedebugging mit GDB


faui01r:


faui01r:

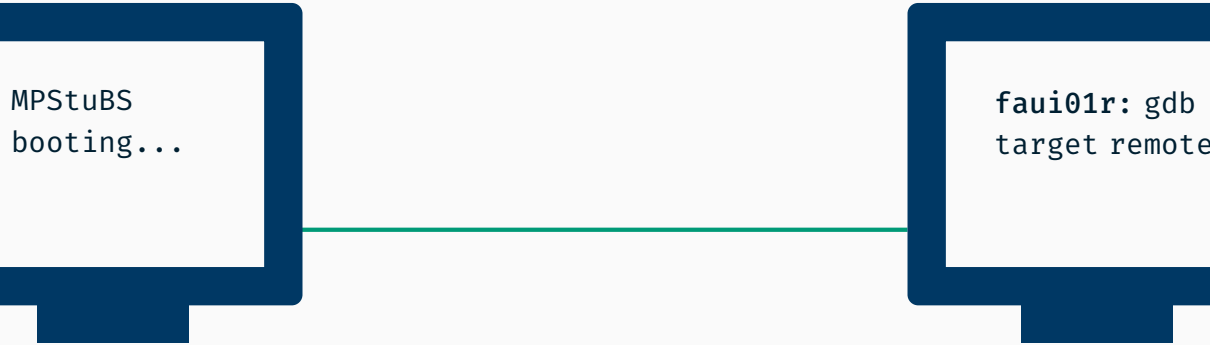

faui01r: gdb

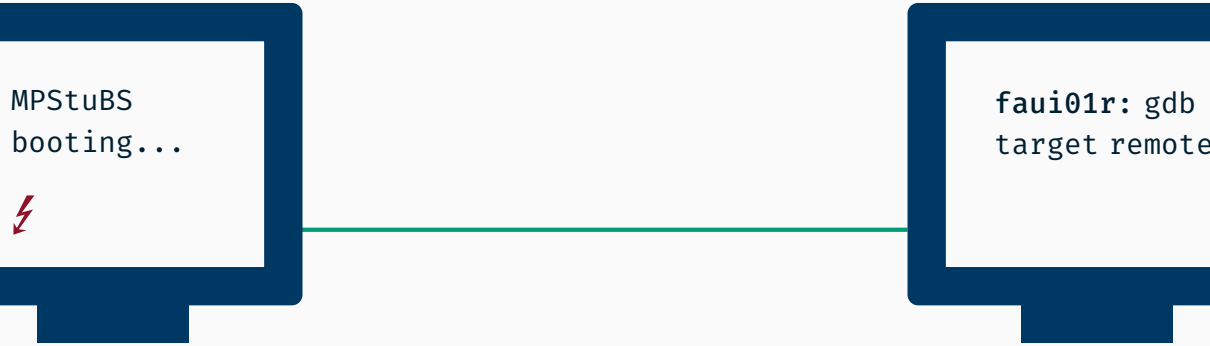

MPStuBS
booting...
faui01r: gdb

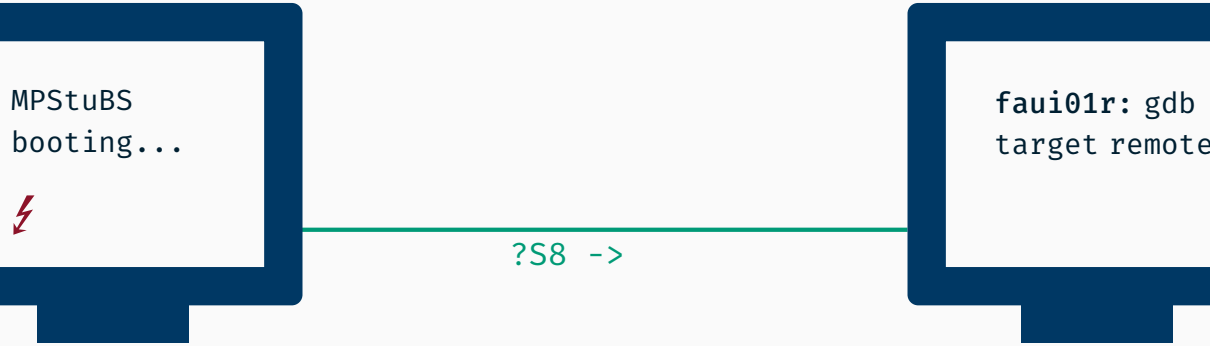

MPStuBS
booting...
faui01r: gdb
target remote

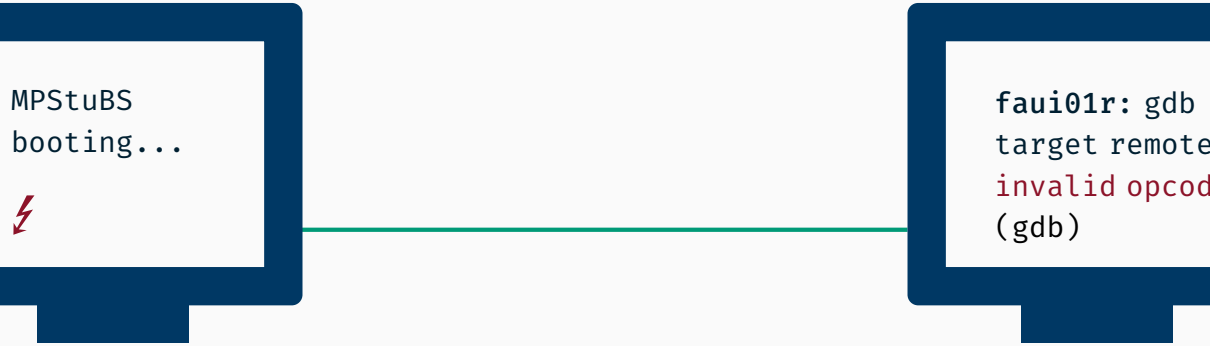

MPStuBS
booting...
<- $R00#23
faui01r: gdb
target remote

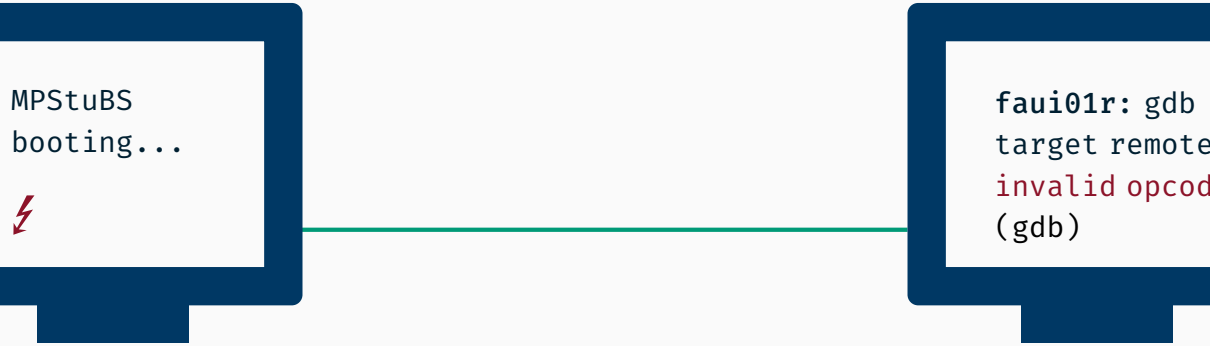

MPStuBS
booting...
+ ->
faui01r: gdb
target remote


MPStuBS
booting...
faui01r: gdb
target remote

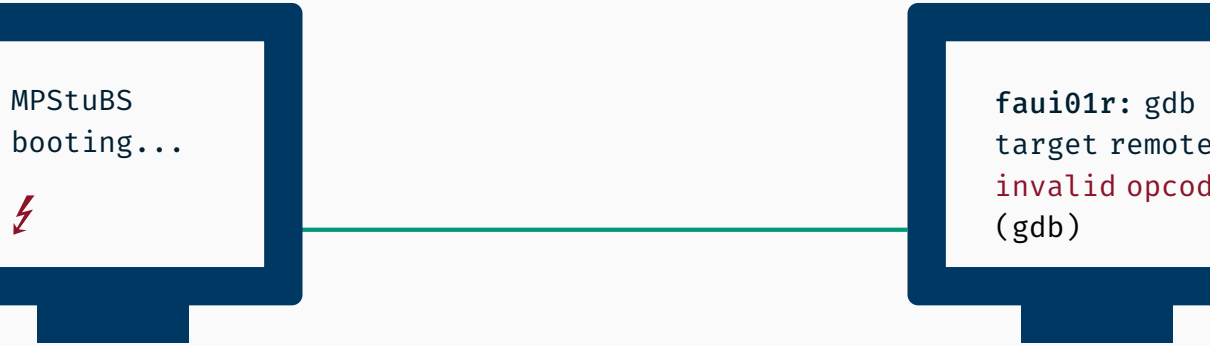

MPStuBS
booting...
faui01r: gdb
target remote


MPStuBS
booting...
?S8 ->
faui01r: gdb
target remote

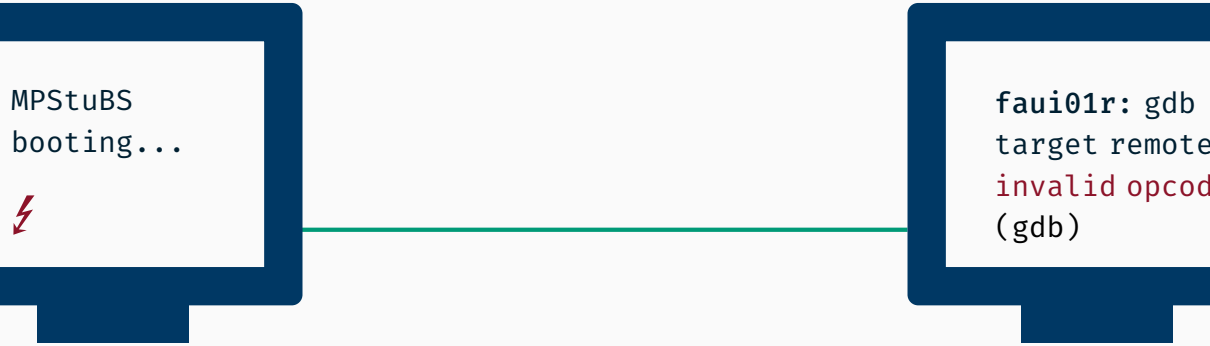

MPStuBS
booting...
faui01r: gdb
target remote
invalid opcod
(gdb)



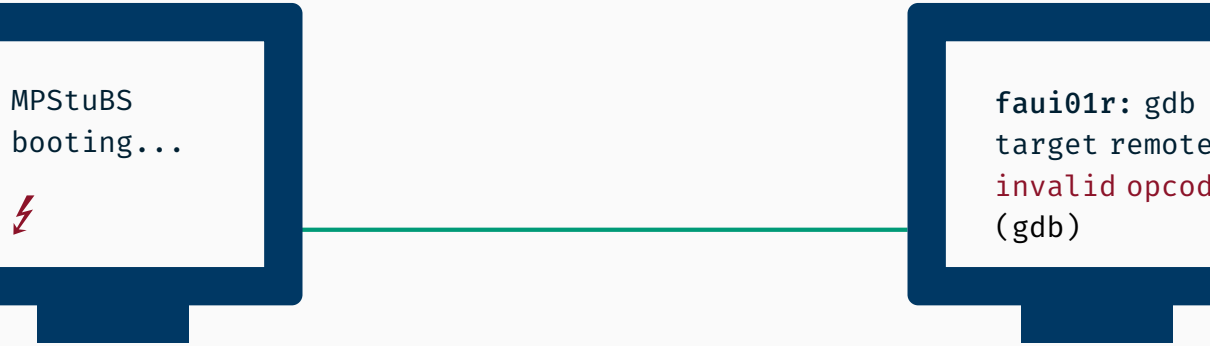
- benötigt serielle Schnittstelle (von Aufgabe 1)



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert

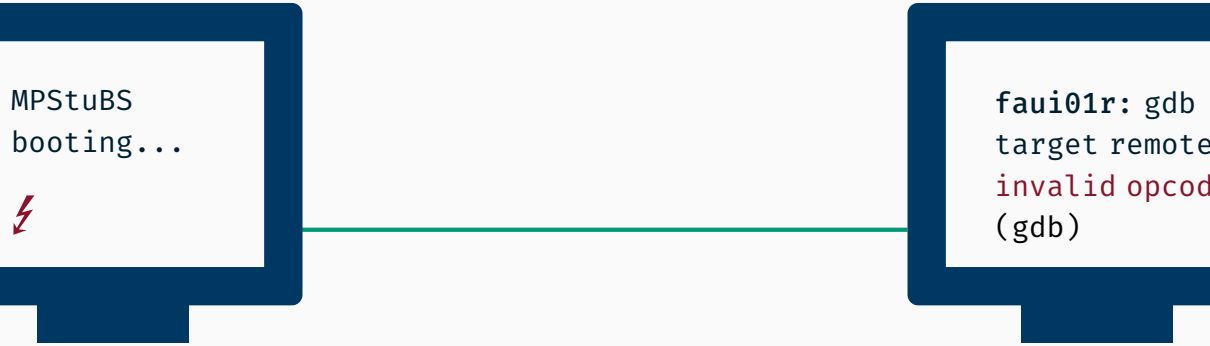


- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps

→ „nur“ die IDT muss bearbeitet werden



- benötigt serielle Schnittstelle (von Aufgabe 1)
- Protokoll ist bereits implementiert
- verwendet eigene IRQ-Handler für Traps

→ „nur“ die IDT muss bearbeitet werden

- aber ist freiwillig

Fragen?

Zur Erinnerung:
Nächste Woche ist wieder ein Seminar (14. November)
und Abgabe von Aufgabe 1 bis 15. November im WinCIP
(keine Tafelübung!)