

Übung zu Betriebssysteme

Interruptbehandlung

15. November 2022

Phillip Raffeck, Dustin Nguyen & Bernhard Heinloth

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Interrupts und Traps



CPU



CPU



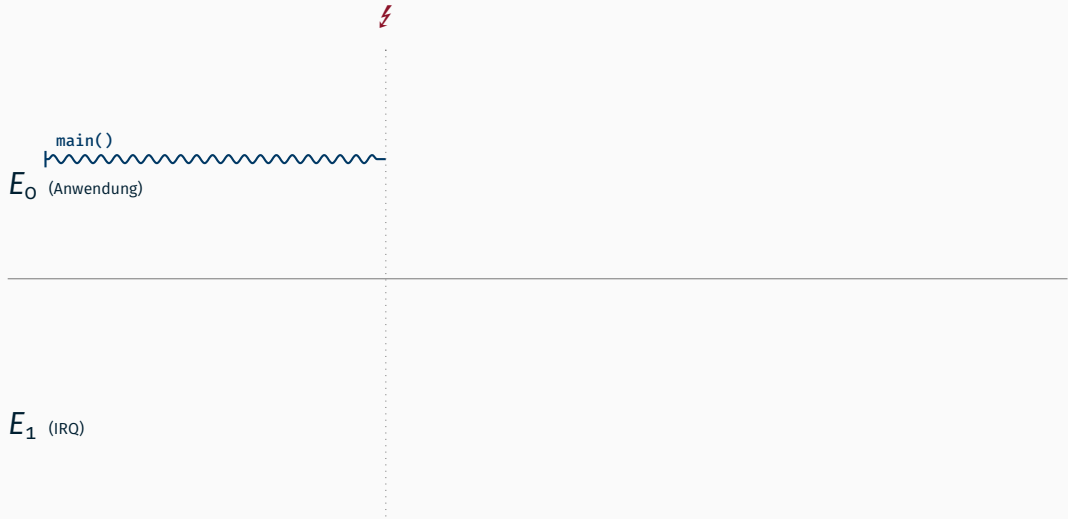
Unterbrechung (Anwendungssicht)

`main()`

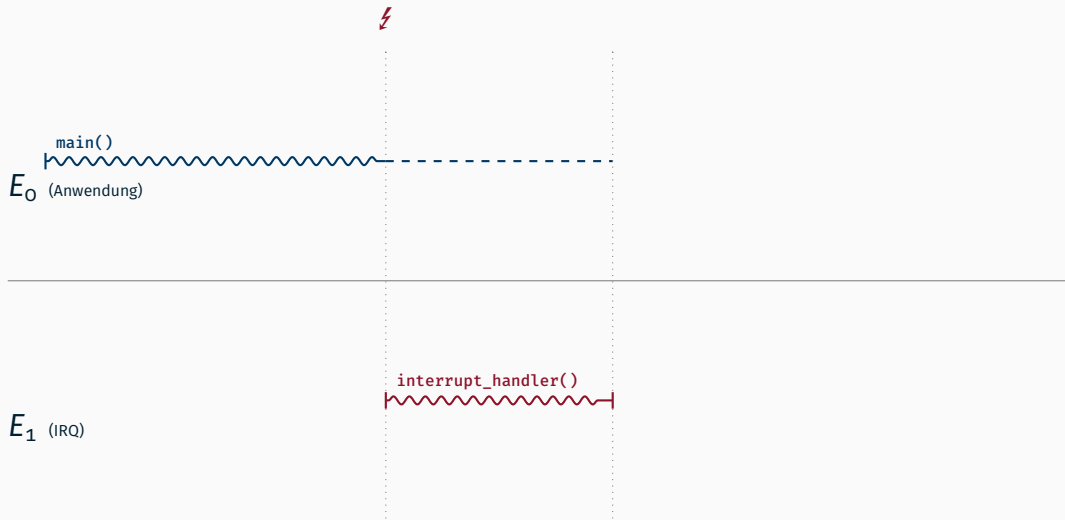
 E_0 (Anwendung)

E_1 (IRQ)

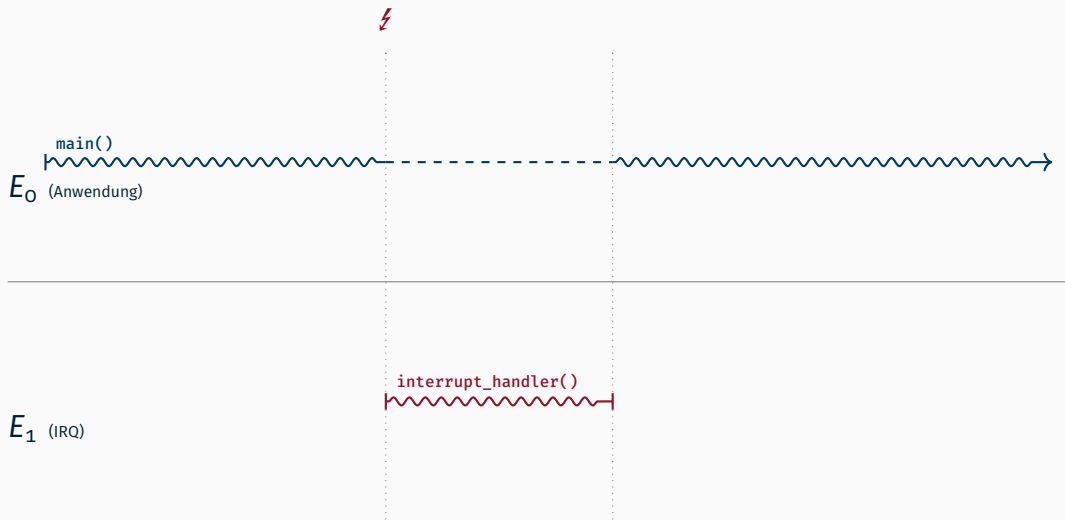
Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)



Unterbrechung (Anwendungssicht)



Minimaler zu sichernder Zustand?

Minimaler zu sichernder Zustand?

- CPU sichert automatisch

rip Instruktionszeiger/Rücksprungadresse

rflags Status/Condition Codes

cs Aktuelles Code Segment



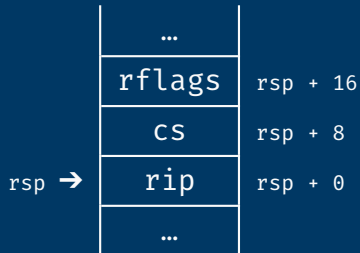
Minimaler zu sichernder Zustand?

- CPU sichert automatisch

rip Instruktionszeiger/Rücksprungadresse

rflags Status/Condition Codes

cs Aktuelles Code Segment



- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iretq`

Unterbrechungsbehandlung

```
;; Assembler  
interrupt_entry:  
    ;; Behandle IRQ  
  
    iretq
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
```

```
interrupt_entry:
```

```
    ;; Behandle IRQ
```

```
    ;; in Hochsprache
```

```
    call interrupt_handler
```

```
    iretq
```

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
    ;; Behandle IRQ
    ;; in Hochsprache
    call interrupt_handler
    iretq
```

```
// C++

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
    ;; Behandle IRQ
    ;; in Hochsprache
    call interrupt_handler
    iretq
```

```
// C++

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

```
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

```
;; Assembler
interrupt_entry:
    ;; Behandle IRQ
    ;; in Hochsprache
    call interrupt_handler
    iretq
```

```
// C++

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

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
    ;; Behandle IRQ
    ;; in Hochsprache
    call interrupt_handler
    iretq
```

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

```
;; Assembler
interrupt_entry:
    ;; Behandle IRQ
    ;; in Hochsprache
    call interrupt_handler
    iretq
```

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

```
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
```

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**



Unterbrechungsbehandlungen müssen auch flüchtige Register sichern!

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler
```

```
interrupt_entry:
```

```
call interrupt_handler
```

```
iretq
```

Unterbrechungsbehandlung (Kontextsicherung)

```
;; Assembler
```

```
interrupt_entry:
```

```
;; Kontext sichern
```

```
push rax
```

```
push rcx
```

```
;; ...
```

```
push r11
```

```
call interrupt_handler
```

```
;; wiederherstellen
```

```
pop r11
```

```
;; ...
```

```
pop rcx
```

```
pop rax
```

```
iretq
```


Unterbrechungsbehandlung (Kontext)

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

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

```
// C++

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

Unterbrechungsbehandlung (Kontext)

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

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

```
// C++
struct Context {

} __attribute__((packed));

extern "C"
void interrupt_handler(Context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

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

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

```
// C++
struct Context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;
} __attribute__((packed));

extern "C"
void interrupt_handler(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
interrupt_entry:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11

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

```
// C++
struct Context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;
} __attribute__((packed));

extern "C"
void interrupt_handler(Context* c)
{
    // Magie.
}
```


Unterbrechungsbehandlung (Kontext)

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

```
// C++
struct Context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;
} __attribute__((packed));

extern "C"
void interrupt_handler(Context* c)
{
    // Magie.
}
```

Unterbrechungsbehandlung (Kontext)

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

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

extern "C"
void interrupt_handler(Context* c)
{
    // Magie.
}
```





Interruptvektoren (x86/x64)

0

255



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
- 10 Invalid TSS
- 11 Segment not Present
- 12 Stack Exception
- 13 General Protection Fault
- 14 Page Fault
- 16 Floating-Point Error
- 17 Alignment Check
- 18 Machine Check

Interruptvektoren (x86/x64)



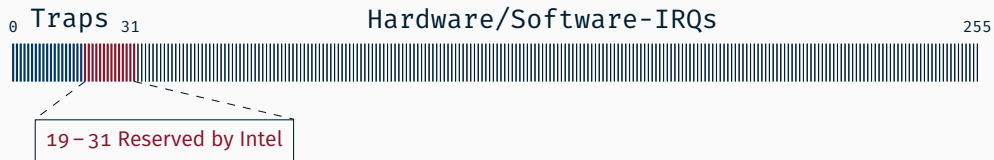
- 0 **Division-by-Zero**
- 1 Debug Exception
- 2 Non-Maskable Interrupt(NMI)
- 3 **Breakpoint (INT 3)**
- 4 Overflow Exception
- 5 Bound Exception
- 6 **Invalid Opcode**
- 7 FPU not Available
- 8 Double Fault
- 10 Invalid TSS
- 11 Segment not Present
- 12 Stack Exception
- 13 **General Protection Fault**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

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
- 10 Invalid TSS
- 11 Segment not Present
- 12 Stack Exception
- 13 General Protection Fault
- 14 **Page Fault**
- 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` (clear interrupt flag) Interruptleitung sperren

`sti` (set interrupt flag) Interruptleitung freigeben

Unterbrechungsbehandlung

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

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

```
// C++
extern "C"
void interrupt_handler(
    Context* c

)
{
    // Magie:

}
```

Unterbrechungsbehandlung

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

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

```
// C++
extern "C"
void interrupt_handler(
    Context* c

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

Unterbrechungsbehandlung

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

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

```
// C++
extern "C"
void interrupt_handler(
    Context* c

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

Unterbrechungsbehandlung

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

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

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


Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
interrupt_entry_6:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp

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

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

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
interrupt_entry_6:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, 6
    call interrupt_handler
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    iretq
```

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

Unterbrechungsbehandlung (für Vektor 6)

```
;; Assembler
interrupt_entry_6:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, 6
    call interrupt_handler
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    iretq
```

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

Unterbrechungsbehandlung (Binden)

```
;; Assembler
interrupt_entry_6:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, 6
    call interrupt_handler
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    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 (Binden)

```
;; Assembler
interrupt_entry_6:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, 6
    call 0x10070f0 <interrupt_handler>
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    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 (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 interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

10070f0 <interrupt_handler>

Unterbrechungsbehandlung (Binden)

Speicher adresse	Maschinencode
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 interrupt/handler.asm
03 CXX interrupt/handler.cc
04 LD .build/system64
```

Speicheradressen beim **Binden**:

10070f0 <interrupt_handler>

Unterbrechungsbehandlung (Binden)

Speicher adresse	Maschinencode
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 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 (Binden)

```
;; Assembler
interrupt_entry_6:
  ;; Kontext sichern
  push rax
  push rcx
  ;; ...
  push r11
  ;; Pointer auf Stack
  mov rdi, rsp
  ;; Vektornummer
  mov rsi, 6
  call interrupt_handler
  ;; wiederherstellen
  pop r11
  ;; ...
  pop rcx
  pop rax
  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>

Unterbrechungsbehandlungen

```
%macro IRQ 1
align 8
interrupt_entry_%1:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, %1
    call interrupt_handler
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    iretq
%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>

Unterbrechungsbehandlung

```
%macro IRQ 1
align 8
interrupt_entry_%1:
    ;; Kontext sichern
    push rax
    push rcx
    ;; ...
    push r11
    ;; Pointer auf Stack
    mov rdi, rsp
    ;; Vektornummer
    mov rsi, %1
    call interrupt_handler
    ;; wiederherstellen
    pop r11
    ;; ...
    pop rcx
    pop rax
    iretq
%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		
96		
95		Unused – muss 0 sein
48		Offset (high): oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code>)
47		Present: Eintrag aktiv (1) oder inaktiv (0)
46		Descriptor Privilege Level
45		Storage Segment: 0 für Interrupt und Traps
44		Mode: 16-bit (0) oder 32/64-bit (1)
43		Type: Task (5), Interrupt (6) oder Trap (7)?
42		
40		
39		
35		Unused – muss 0 sein
34		Interrupt Stack Table
32		
31		Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15		Offset (low): unterer Teil der Einsprungsadresse für die Interruptbehandlung
0		

Interrupt Deskriptor (Beispiel)

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

für `100 0230 <interrupt_entry_6>`

Interrupt Deskriptor (Beispiel)

127	0	Unused – muss 0 sein
96		
95	0x100	Offset (high): oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code>)
48		
47	1	Present: Eintrag aktiv (1) oder inaktiv (0)
46		
45	0	Descriptor Privilege Level
44		
44	0	Storage Segment: 0 für Interrupt und Traps
43		
43	1	Mode: 16-bit (0) oder 32/64-bit (1)
42		
40	6	Type: Task (5), Interrupt (6) oder Trap (7)?
39		
35	0	Unused – muss 0 sein
34		
32	0	Interrupt Stack Table
31		
31	8	Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15	0x0230	Offset (low): unterer Teil der Einsprungsadresse für die Interruptbehandlung
0		

für `100 0230 <interrupt_entry_6>`

Interrupt Deskriptor (Beispiel)

127	0	Unused – muss 0 sein
96		
95	0x100	Offset (high): oberer Teil der Einsprungsadresse für die Interruptbehandlung (z.B. <code>interrupt_entry_6</code>)
48		
47	1	Present: Eintrag aktiv (1) oder inaktiv (0)
46		
45	0	Descriptor Privilege Level
44		
44	0	Storage Segment: 0 für Interrupt und Traps
43		
43	1	Mode: 16-bit (0) oder 32/64-bit (1)
42		
40	6	Type: Task (5), Interrupt (6) oder Trap (7)?
39		
35	0	Unused – muss 0 sein
34		
32	0	Interrupt Stack Table
31		
31	8	Selector: Codesegment, in das beim Interrupt gewechselt wird (i.d.R. Kernel-Codesegment)
16		
15	0x0230	Offset (low): 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>

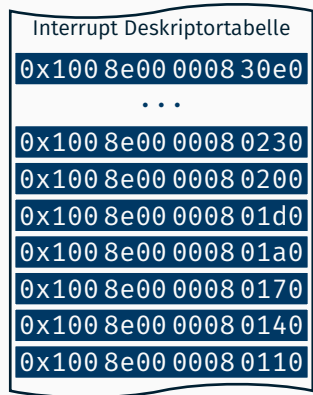
100 01a0 <interrupt_entry_3>

100 0170 <interrupt_entry_2>

100 0140 <interrupt_entry_1>

100 0110 <interrupt_entry_0>

Speicher



Interrupt 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 Deskriptortabelle

0x100 8e00 0008 30e0

...

0x100 8e00 0008 0230

0x100 8e00 0008 0200

0x100 8e00 0008 01d0

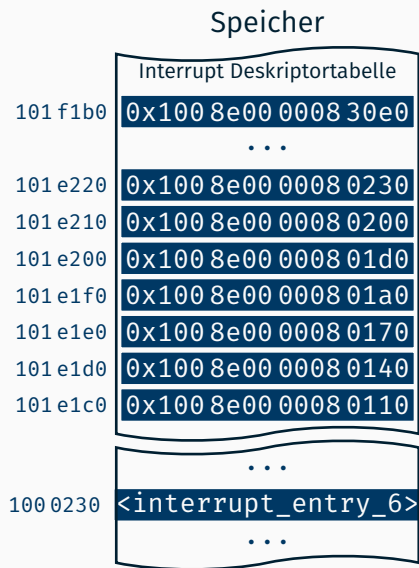
0x100 8e00 0008 01a0

0x100 8e00 0008 0170

0x100 8e00 0008 0140

0x100 8e00 0008 0110

Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`

79



Basis: Startadresse der IDT

16

15

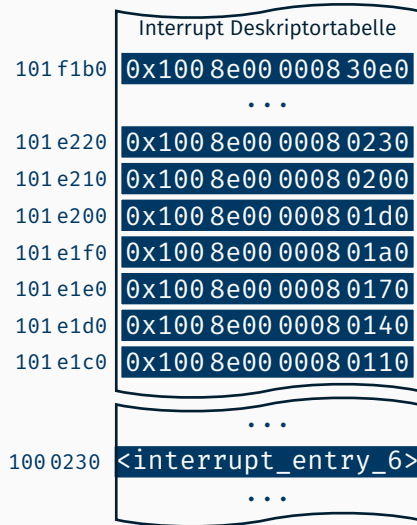


Limit: Bytes

0

*Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`

79

101 e1c0

Basis: Startadresse der IDT

16

15

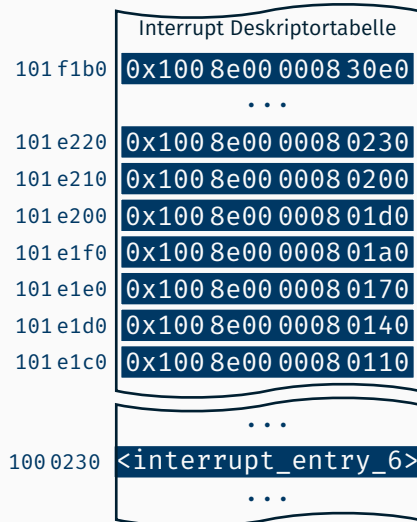
4095

Limit: Bytes

0

*Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`

79

101 e1c0

Basis: Startadresse der IDT

16

15

4095

Limit: Bytes

0

*Einträge * 16 - 1*

Instruktionen:

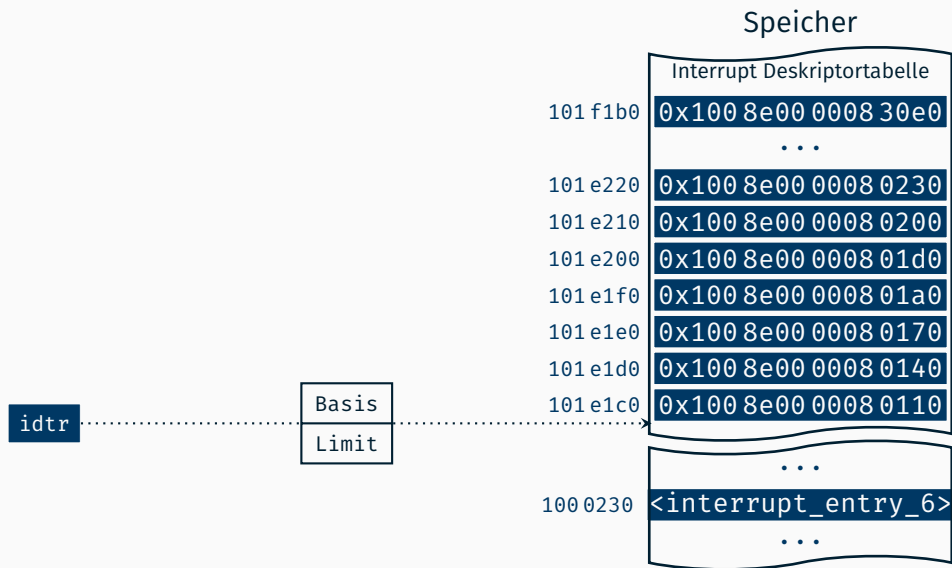
`lidt` in Register laden

`sidt` aus Register lesen

Speicher



Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

⚡ Interrupt 6

idtr

Basis

Limit

Speicher

Interrupt Deskriptortabelle

101 f1b0 0x100 8e00 0008 30e0

...

101 e220 0x100 8e00 0008 0230

101 e210 0x100 8e00 0008 0200

101 e200 0x100 8e00 0008 01d0

101 e1f0 0x100 8e00 0008 01a0

101 e1e0 0x100 8e00 0008 0170

101 e1d0 0x100 8e00 0008 0140

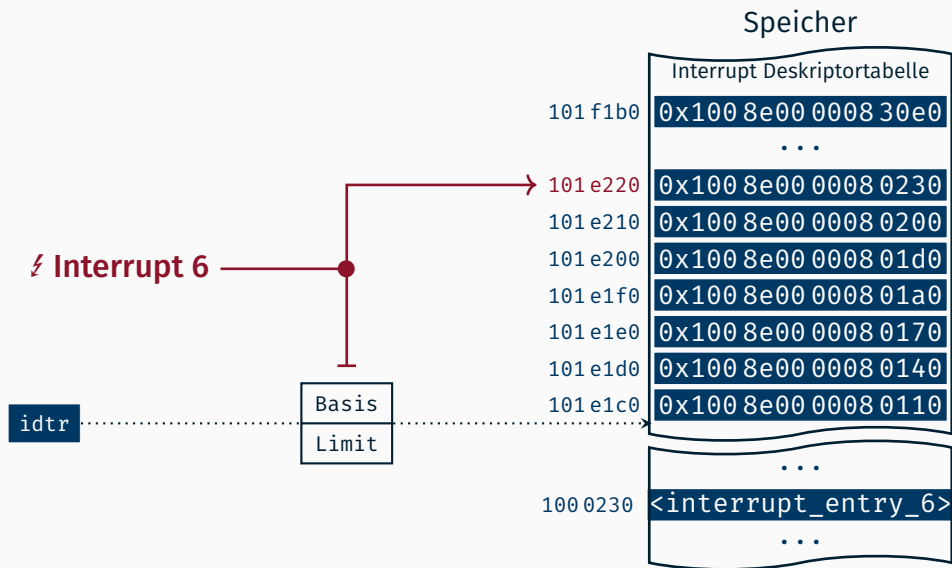
101 e1c0 0x100 8e00 0008 0110

...

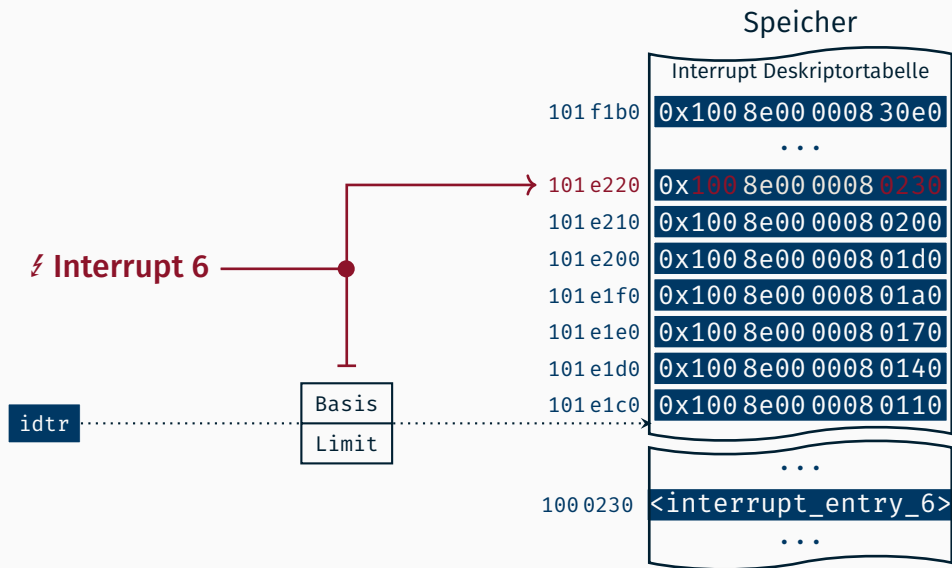
100 0230 <interrupt_entry_6>

...

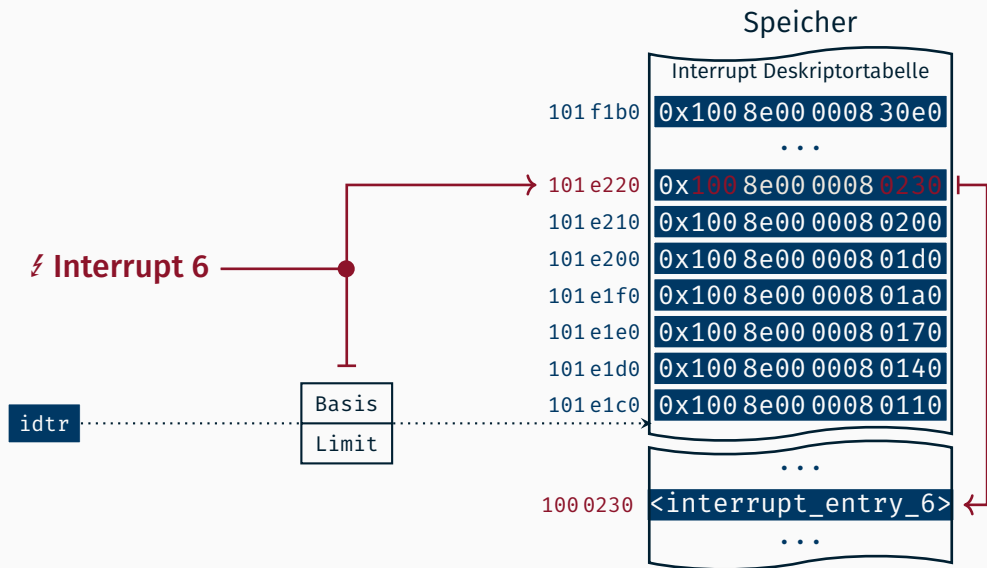
Interrupt Deskriptor Tabelle (IDT)



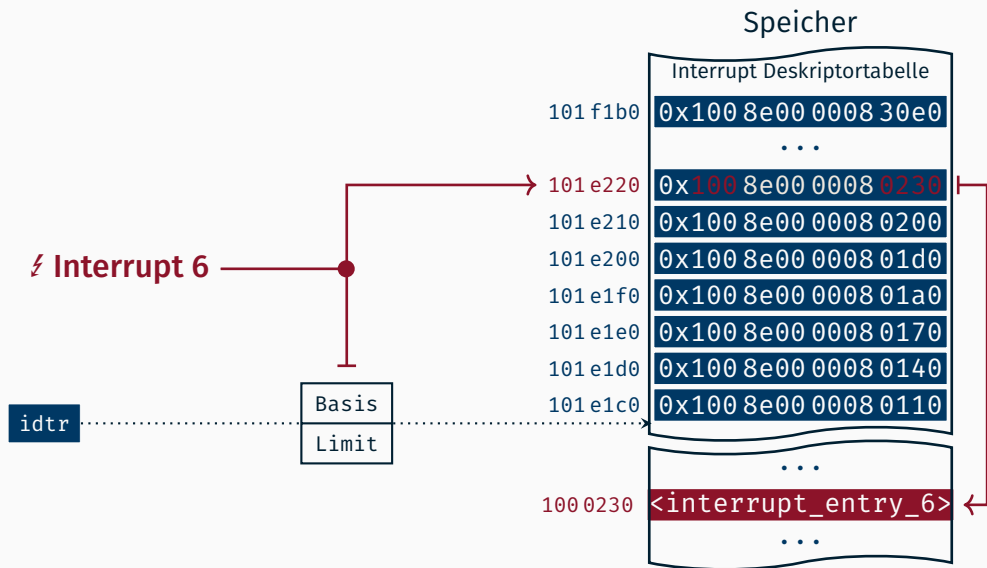
Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)



Externe Interrupts

Unterbrechungen durch externe Geräte bei x86-CPUs



Unterbrechungen durch externe Geräte bei x86-CPUs



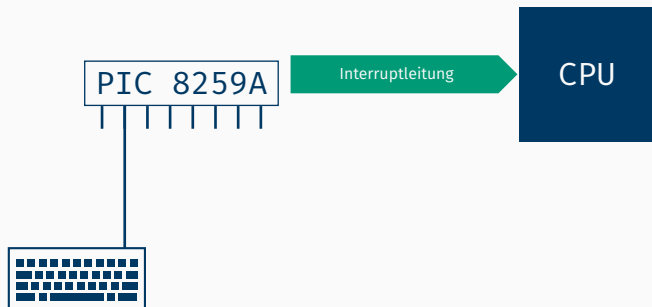
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller

Unterbrechungen durch externe Geräte bei x86-CPUs



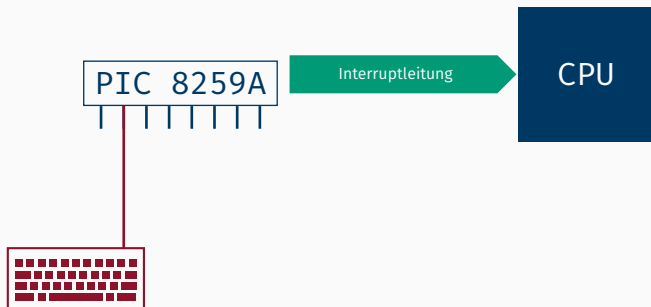
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
 - feste Prioritätenreihenfolge
 - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
 - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



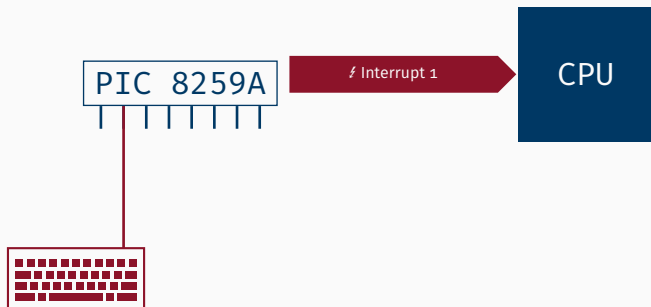
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
 - feste Prioritätenreihenfolge
 - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
 - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



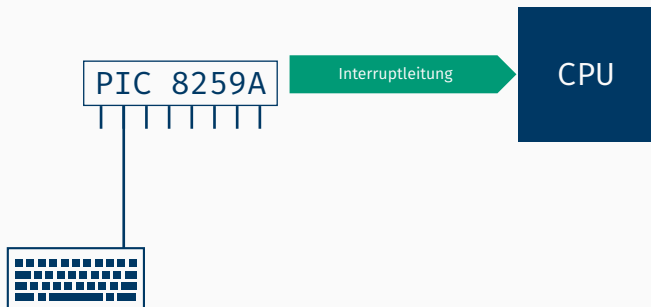
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
 - feste Prioritätenreihenfolge
 - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
 - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



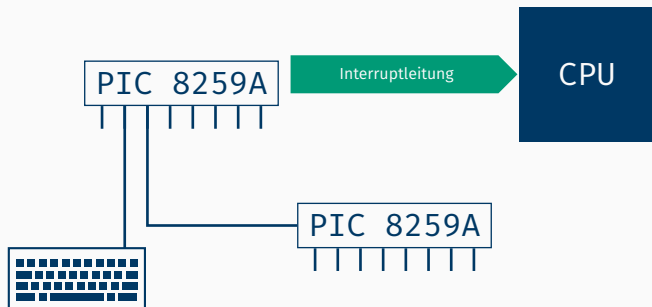
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
 - feste Prioritätenreihenfolge
 - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
 - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



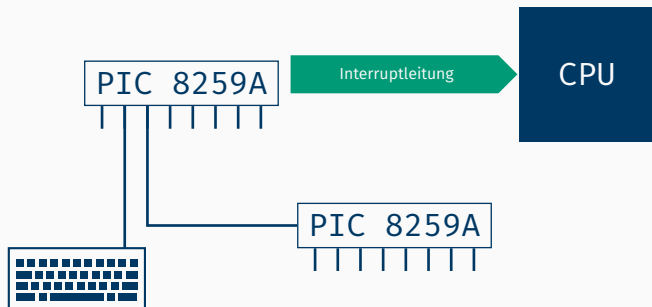
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
 - feste Prioritätenreihenfolge
 - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
 - Konfiguration über I/O-Ports

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

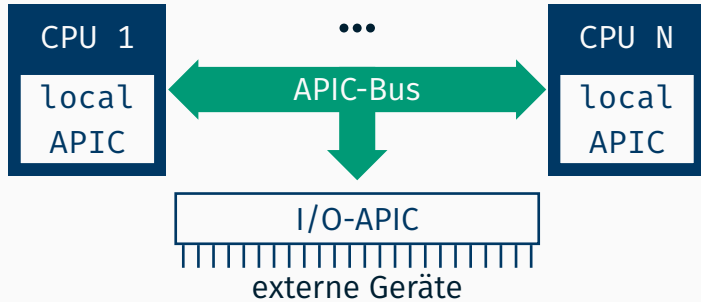
Unterbrechungen durch externe Geräte bei x86-CPUs



- 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
- Nicht für Mehrprozessorsysteme geeignet

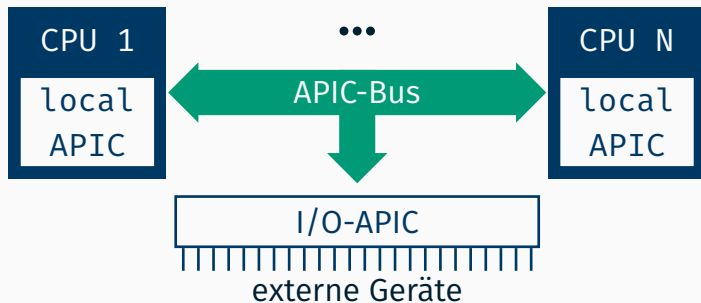
Externe Interrupts mit dem APIC

Aufbau der APIC-Architektur



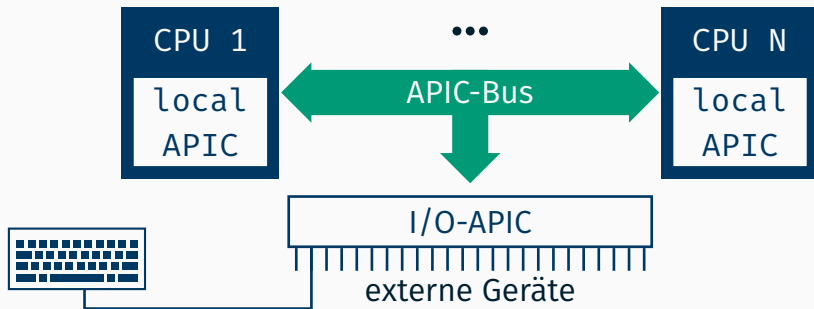
Aufteilung in lokalen APIC und I/O APIC

Aufbau der APIC-Architektur



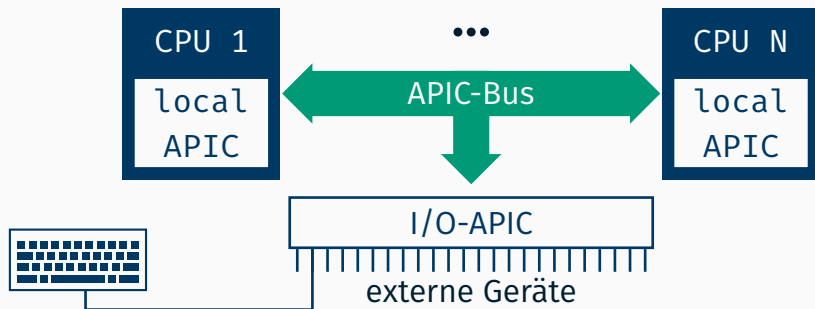
I/O-APIC dient zum Anschluss der Geräte

Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

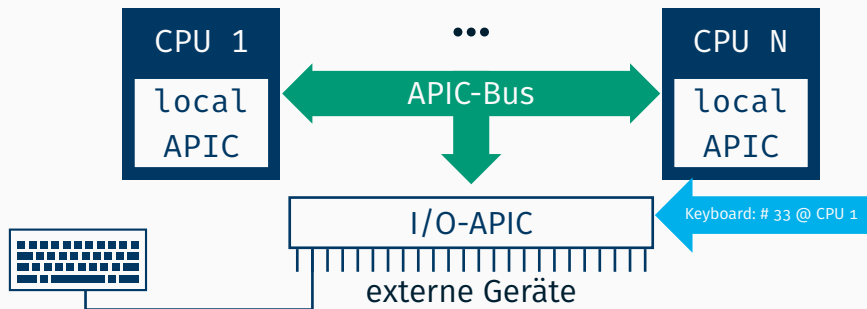
Aufbau der APIC-Architektur



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

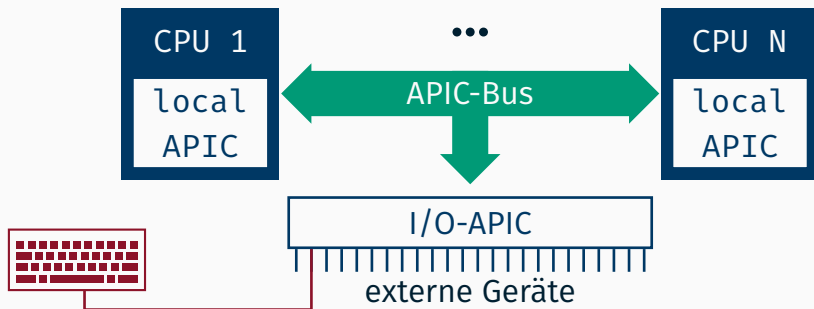
Aufbau der APIC-Architektur



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

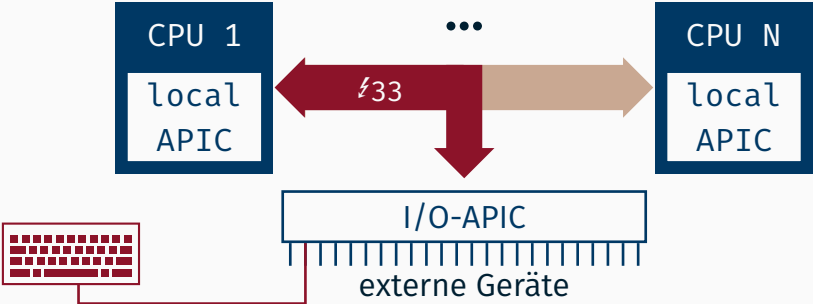
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

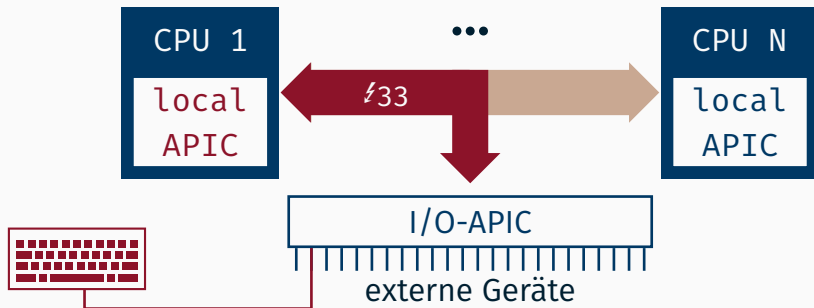
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

Aufbau der APIC-Architektur



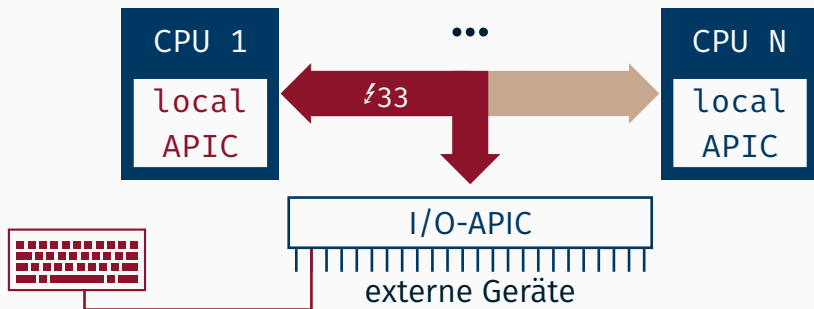
Interrupts werden zu Nachrichten auf dem APIC-Bus

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 I/O

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

Programmierung des Intel I/O-APIC

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

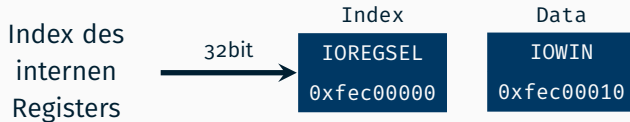
- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister

Index	Data
IOREGSEL	IOWIN
0xfec00000	0xfec00010

Programmierung des Intel I/O-APIC

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

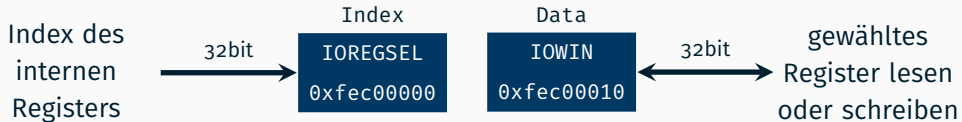
- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister



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



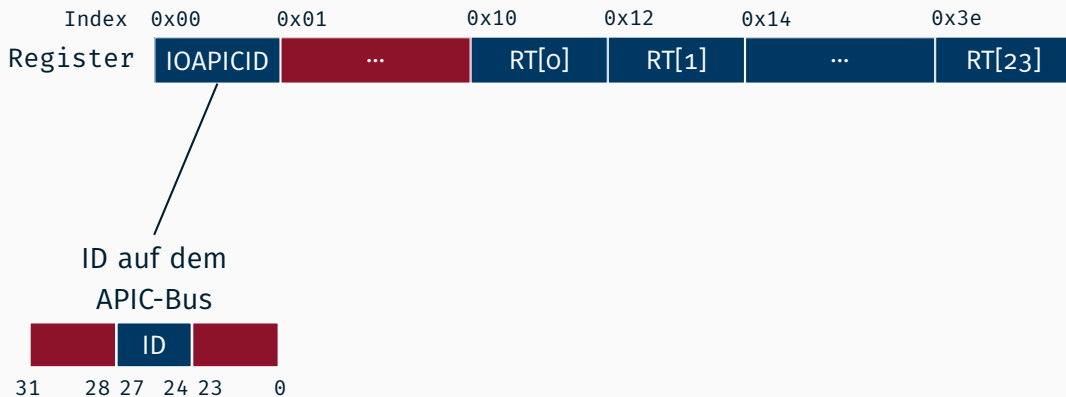
Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

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

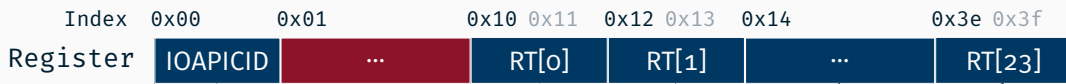
Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



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

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

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

(RO: Read Only)

Zusammenfassendes Beispiel:
Keyboard Interrupt in StuBS

Vorbereitung

- I/O APIC initialisieren

Vorbereitung

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

Vorbereitung

- 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**

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

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
 - Einsprungsroutinen `interrupt_entry` mit Aufruf zu `interrupt_handler` schreiben [wird in der Vorgabe bereits erledigt]

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
 - 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]

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
 - 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**

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
 - 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

Ablauf

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

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus**

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus** und unterbricht CPU

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus** und unterbricht CPU
4. **CPU** führt Unterbrechungsbehandlung aus

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus** und unterbricht CPU
4. **CPU** führt Unterbrechungsbehandlung aus
 - 4.1 Mittels Register **idt r** wird der entsprechende Eintrag in der **Interrupt Deskriptor Tabelle** ausgewählt und in die Einsprungsroutine gesprungen

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus** und unterbricht CPU
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

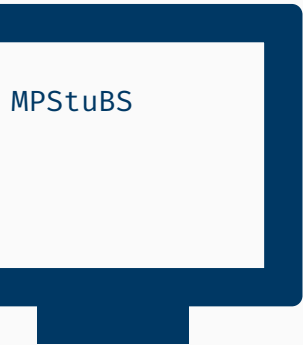
Ablauf

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
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU
3. entsprechender **LAPIC** empfängt Nachricht vom **APIC-Bus** und unterbricht CPU
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

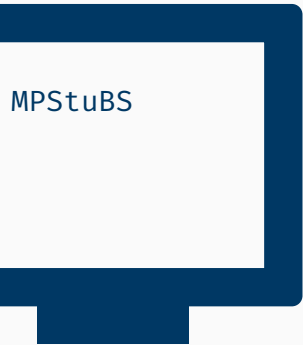
Ablauf

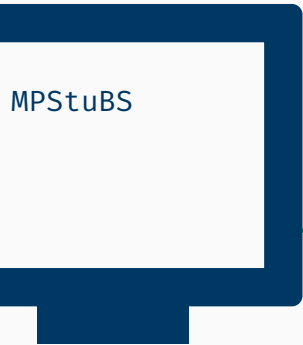
1. **Tastendruck** – Tastaturprozessor (in der Tastatur) meldet dies seriell an den **PS/2-Controller**
2. **PS/2-Controller** aktiviert Interruptleitung zu **I/O APIC**
 - 2.1 Anhand der **Redirection Table** wird Aktion gewählt
 - 2.2 Nachricht auf **APIC-Bus** mit Interruptnr. **33** und Ziel-CPU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

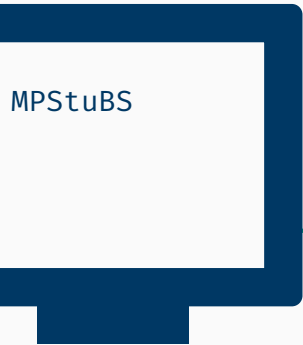
Remotedebugging mit GDB

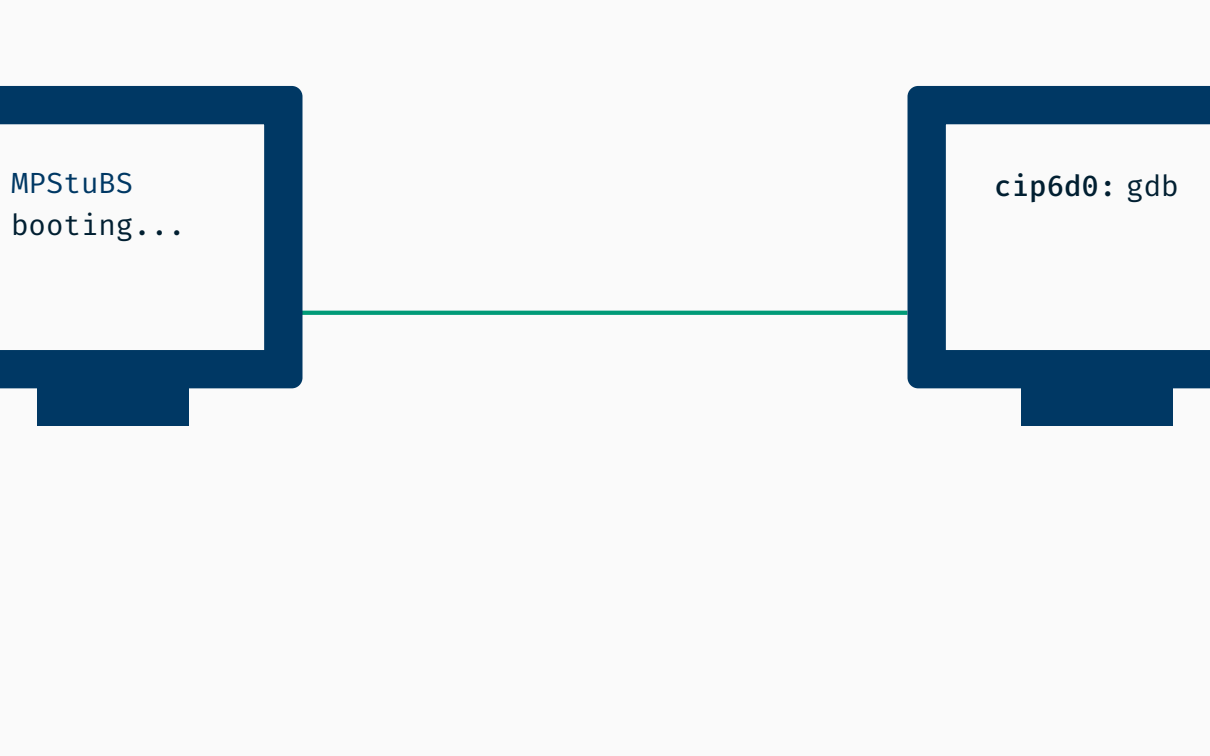


MPStuBS



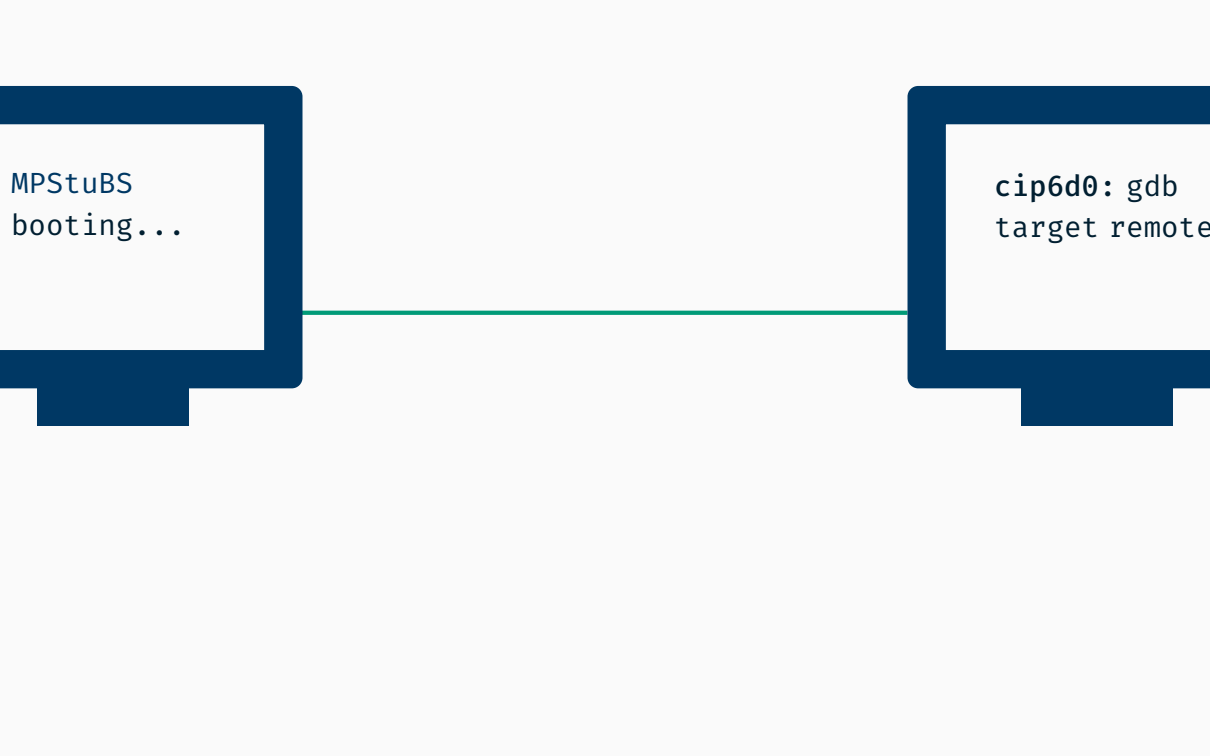






MPStuBS
booting...

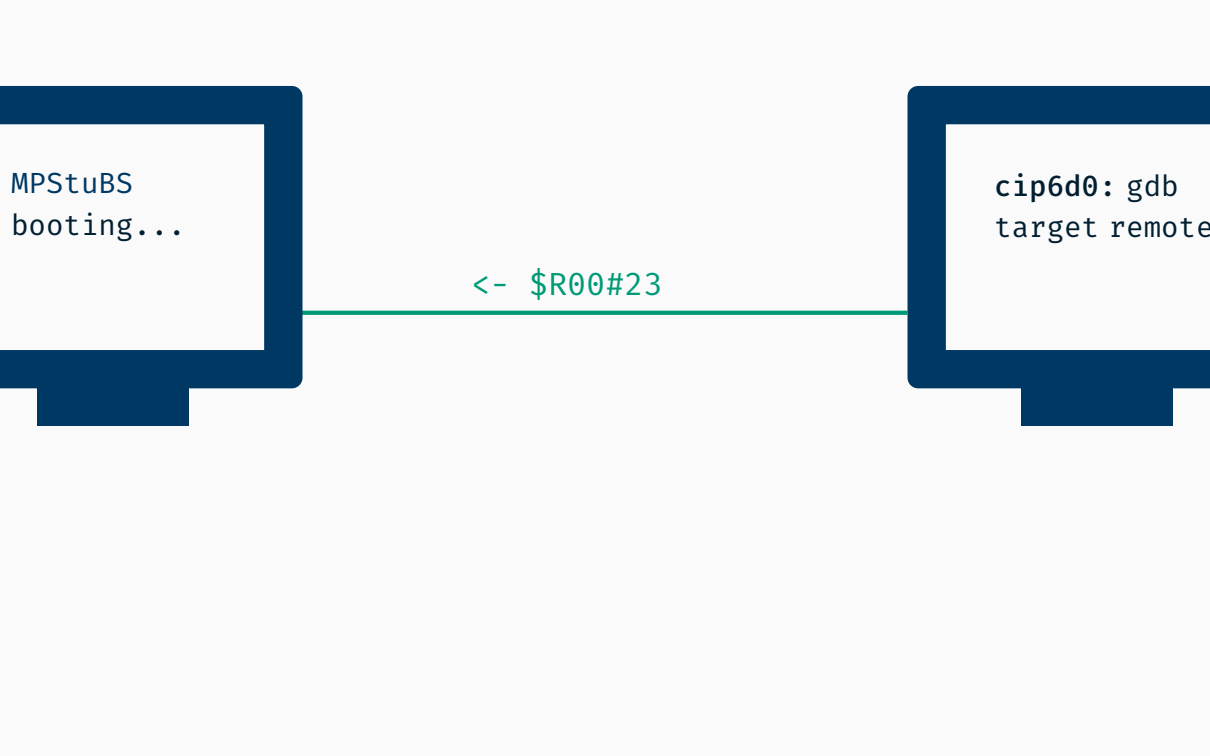
cip6d0: gdb



```
graph LR; A[MPStuBS booting...] --- B[cip6d0: gdb target remote];
```

MPStuBS
booting...

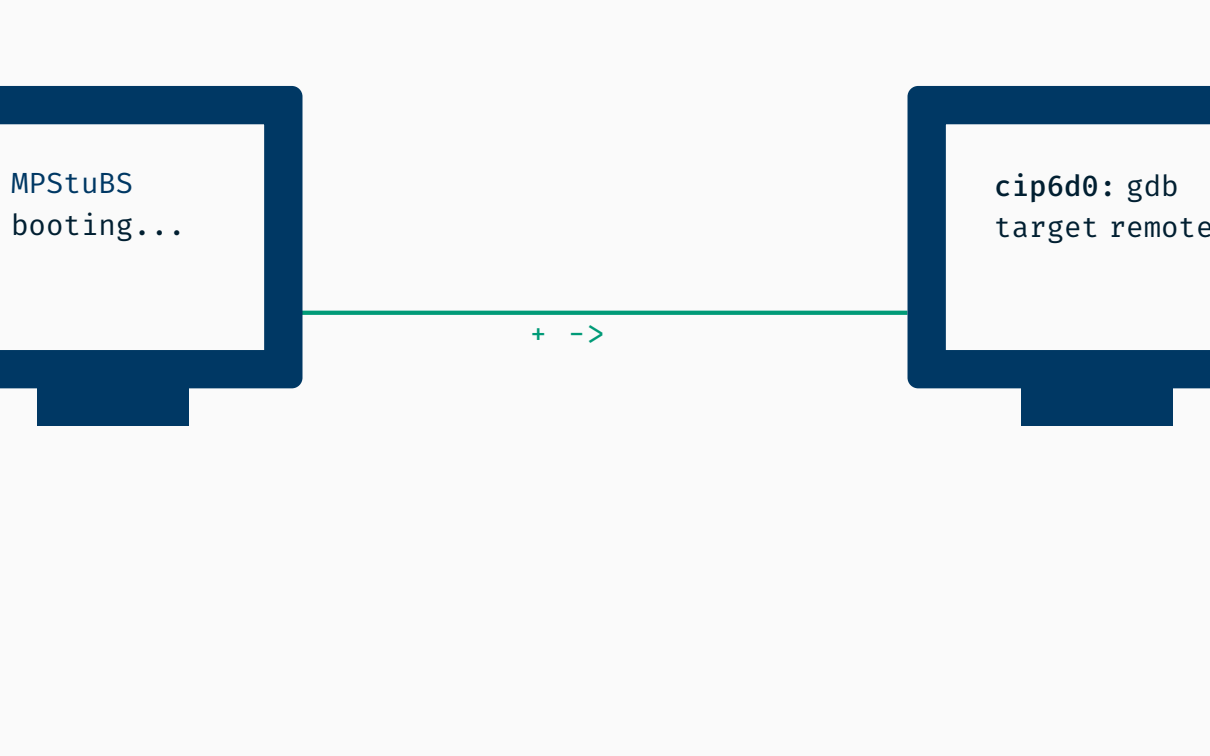
cip6d0: gdb
target remote



```
MPStuBS
booting...
```

<- \$R00#23

```
cip6d0: gdb
target remote
```

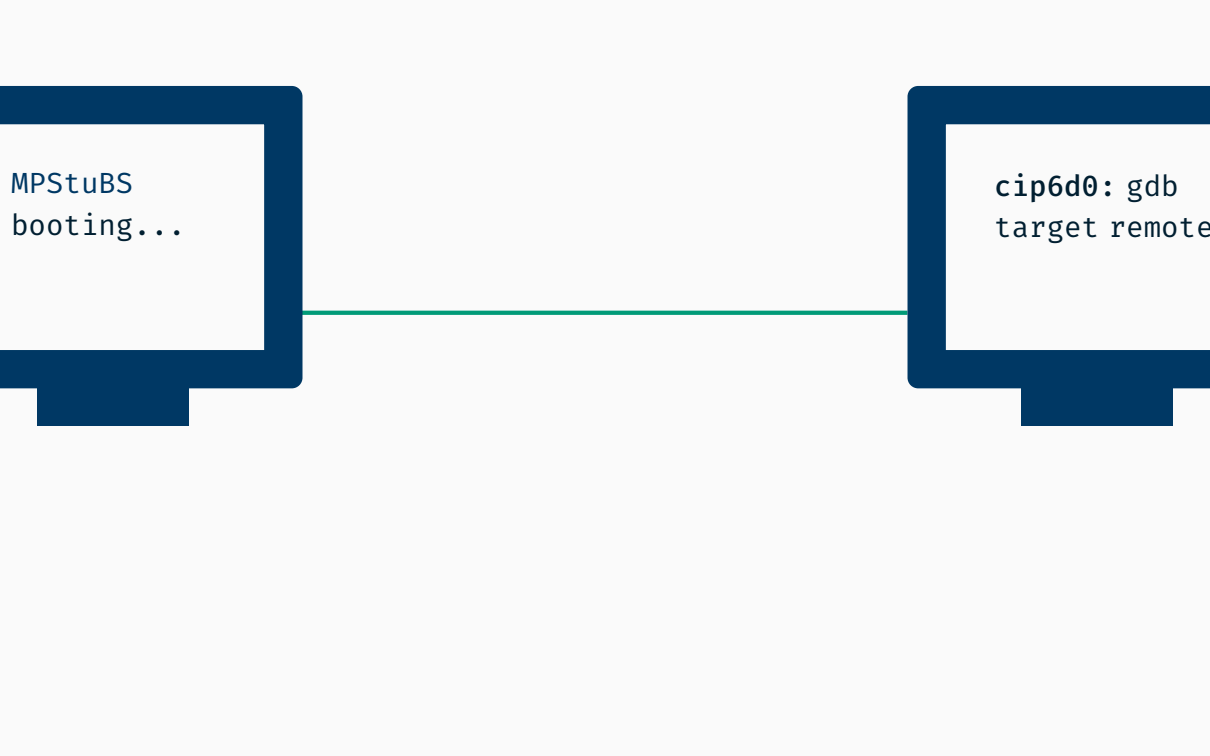


```
graph LR; A[MPStuBS booting...] --- B[cip6d0: gdb target remote];
```

MPStuBS
booting...

+ ->

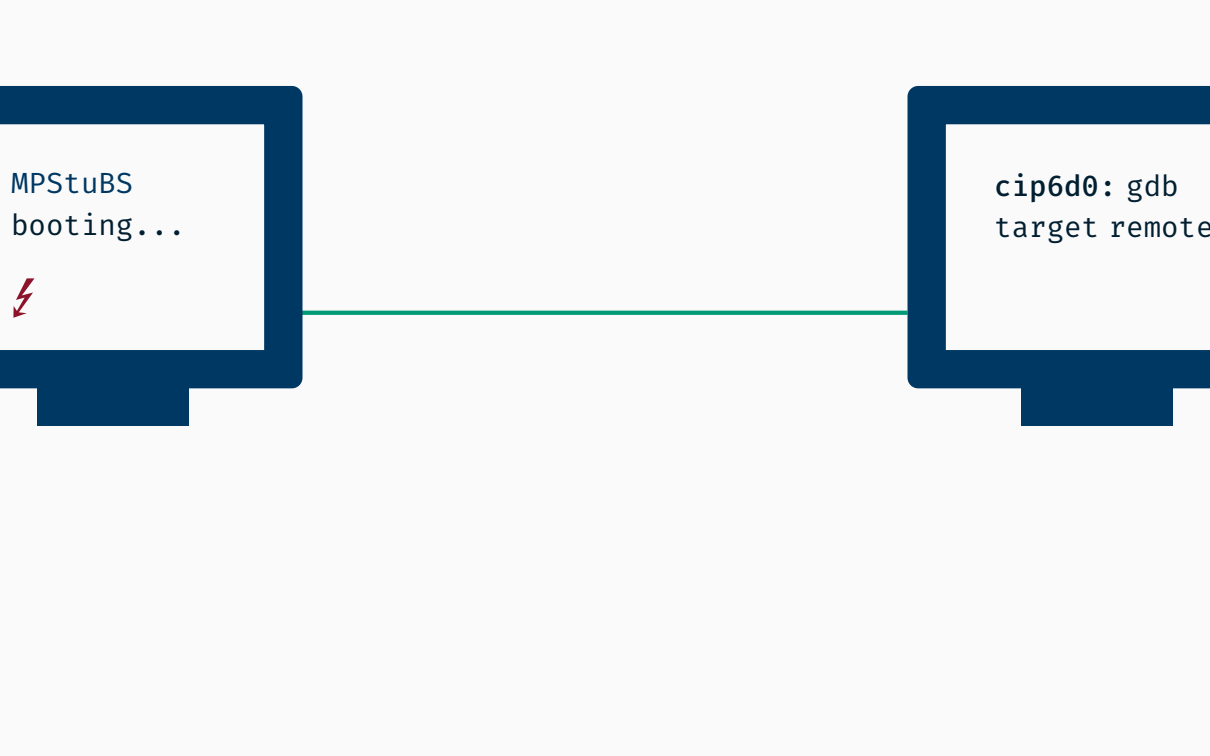
cip6d0: gdb
target remote



```
graph LR; A[MPStuBS booting...] --- B[cip6d0: gdb target remote];
```

MPStuBS
booting...

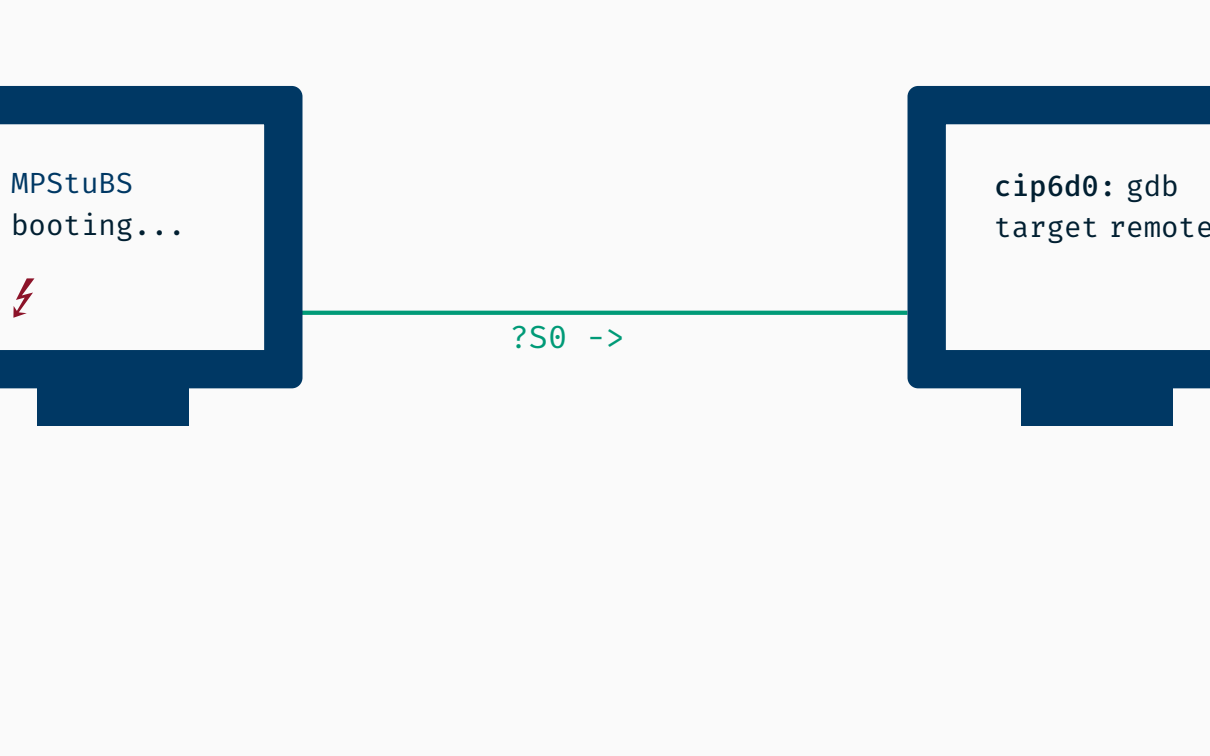
cip6d0: gdb
target remote



MPStuBS
booting...



cip6d0: gdb
target remote

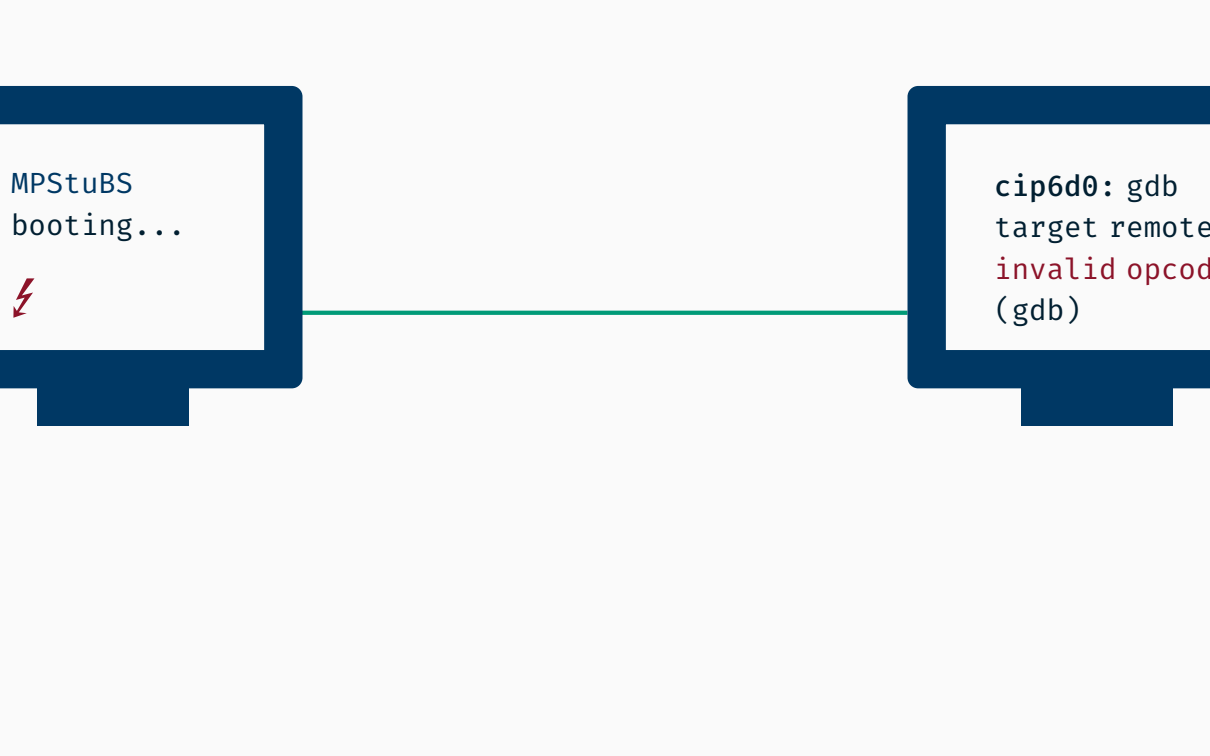


MPStuBS
booting...



?S0 ->

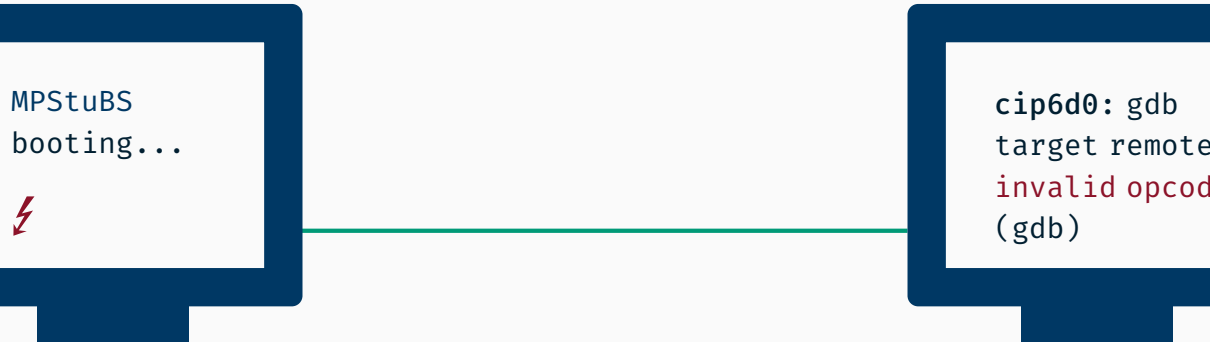
cip6d0: gdb
target remote



```
MPStuBS
booting...
```



```
cip6d0: gdb
target remote
invalid opcode
(gdb)
```

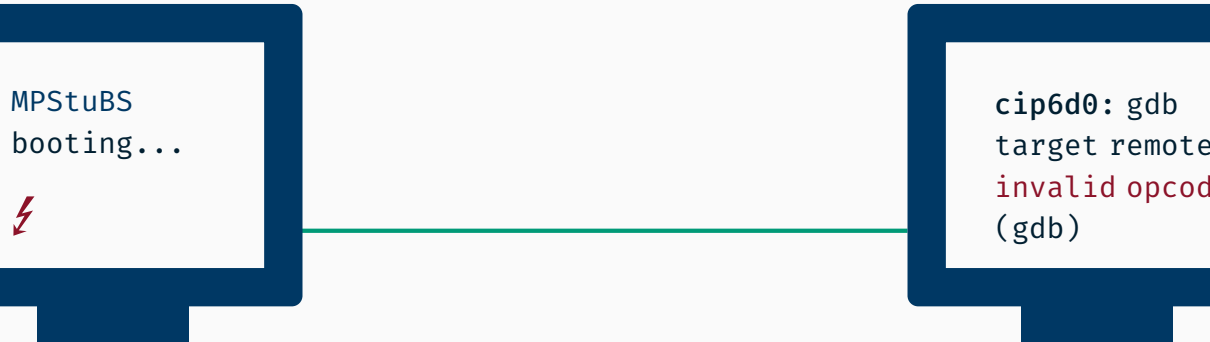


```
MPStuBS
booting...
```



```
cip6d0: gdb
target remote
invalid opcode
(gdb)
```

- Protokoll ist bereits implementiert

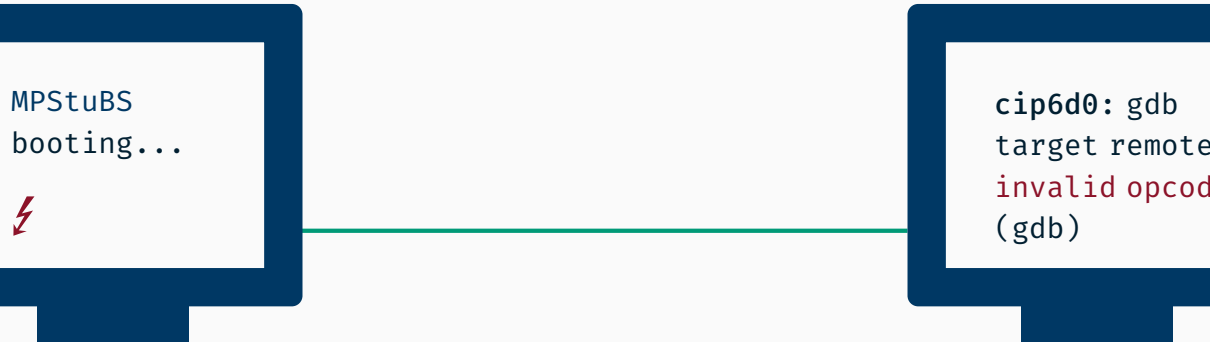


```
MPStuBS
booting...
```



```
cip6d0: gdb
target remote
invalid opcode
(gdb)
```

- Protokoll ist bereits implementiert
- verwendet eigene Unterbrechungsbehandlung für Traps

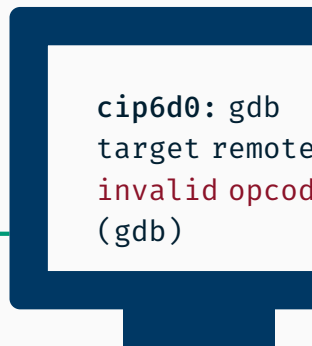


```
MPStuBS
booting...
```

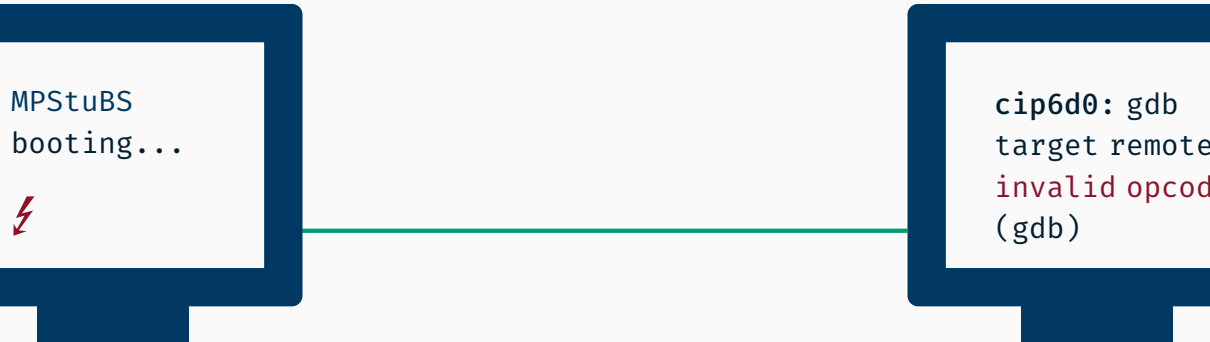


```
cip6d0: gdb
target remote
invalid opcode
(gdb)
```

- Protokoll ist bereits implementiert
 - verwendet eigene Unterbrechungsbehandlung für Traps
- modifiziert die IDT



- Protokoll ist bereits implementiert
- verwendet eigene Unterbrechungsbehandlung für Traps
→ modifiziert die IDT
- nur die serielle Schnittstelle (von Aufgabe 1) wird benötigt



```
MPStuBS
booting...
```



```
cip6d0: gdb
target remote
invalid opcode
(gdb)
```

- Protokoll ist bereits implementiert
- verwendet eigene Unterbrechungsbehandlung für Traps
- modifiziert die IDT
- nur die serielle Schnittstelle (von Aufgabe 1) wird benötigt
- aber ist freiwillig

Aufgabe 2

Lernziele

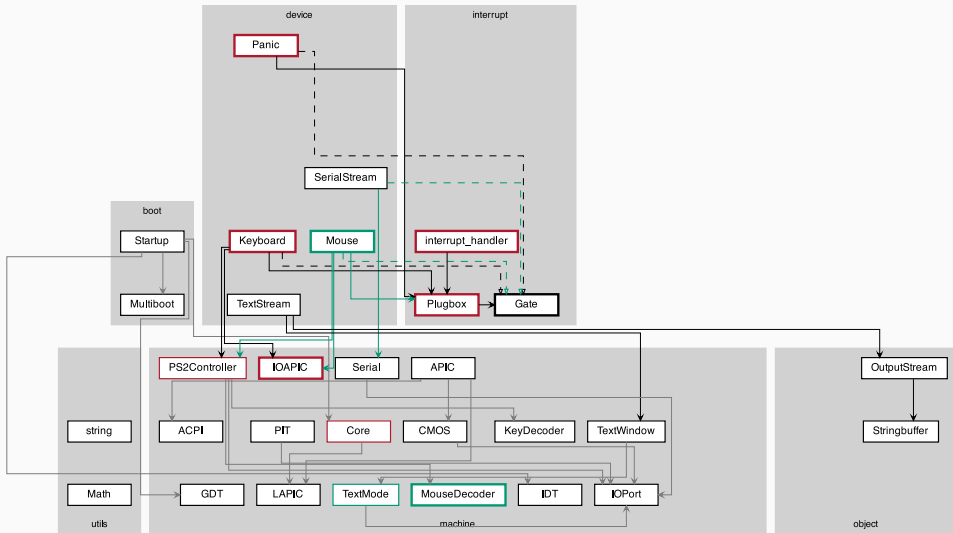
- Behandlung asynchroner Ereignisse
- Problematik und Schutz kritischer Abschnitte

Lernziele

- Behandlung asynchroner Ereignisse
- Problematik und Schutz kritischer Abschnitte

Aufgabe

- Konfiguration externer Geräte über I/O APIC
- Treiber für **Tastatur** und (*optional*) **Maus**
- Wechselseitiger Ausschluss (**Ticket-/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



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.



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.

(weitere Details siehe FAQ auf der Webseite)

Fragen?

Abgabe der 2. Aufgabe bis Freitag, 25. November
Nächste Woche ist wieder ein Seminar (22. November)