Übung zu Betriebssystembau

Interruptbehandlung

31. Oktober 2024

Alexander Krause

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

(Mit Material vom Lehrstuhl 4 der FAU)

Dann schlug ein Blitz ein …

Dann schlug ein Blitz ein …

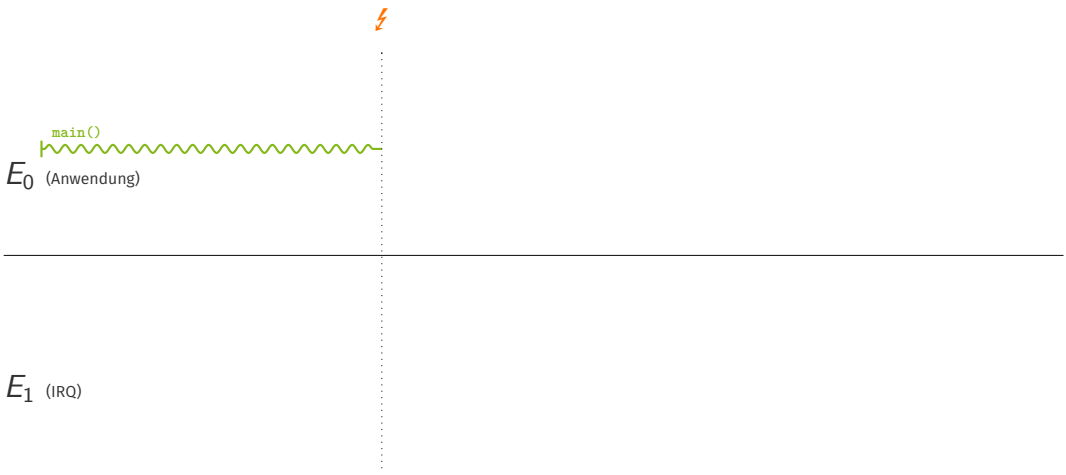
Dann schlug ein Blitz ein …

Unterbrechung aus Anwendungssicht

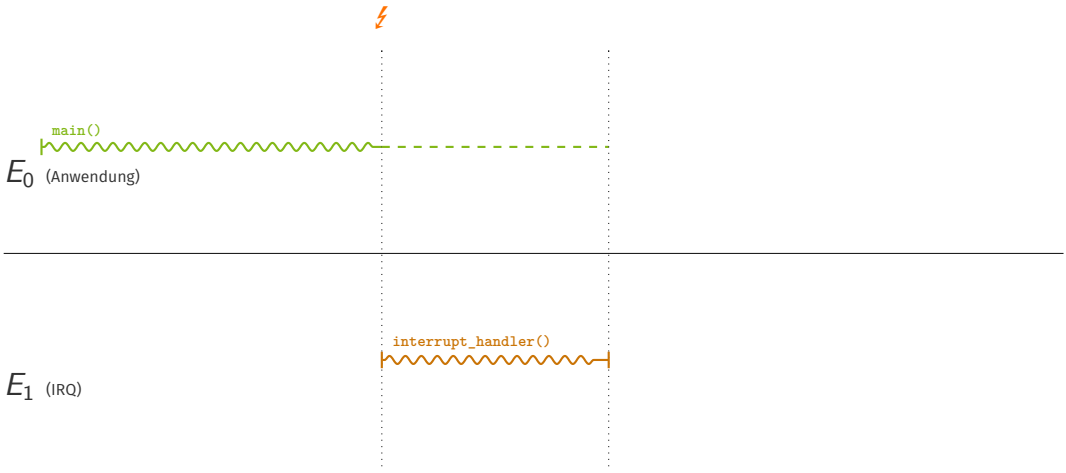
E_0 (Anwendung)

E_1 (IRQ)

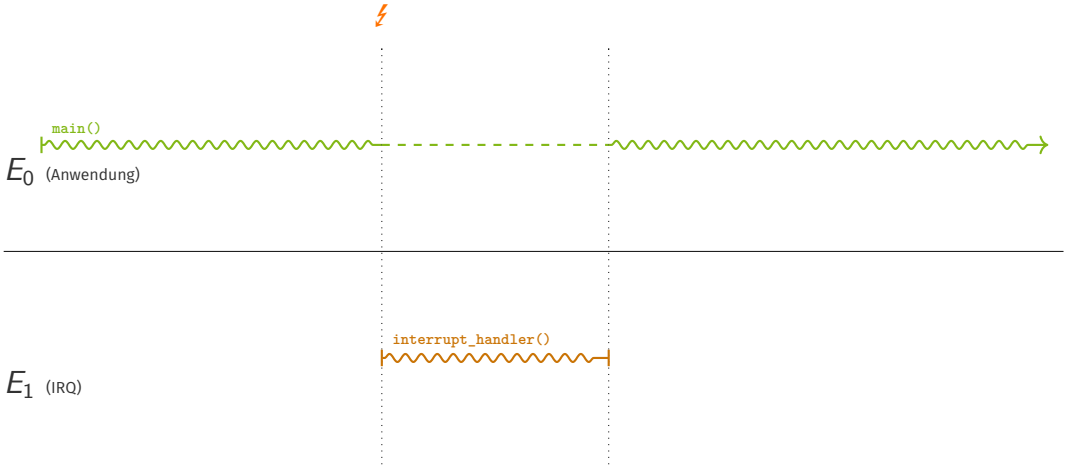
Unterbrechung aus Anwendungssicht



Unterbrechung aus Anwendungssicht



Unterbrechung aus Anwendungssicht



Zustandsicherung (1)

Was muss **mindestens** gesichert werden?

Zustandsicherung (1)

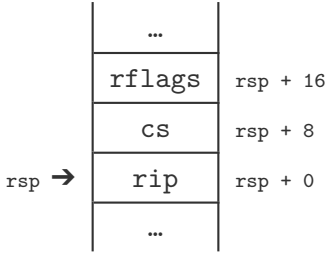
Was muss **mindestens** gesichert werden?

- CPU sichert automatisch

  rip Instruktionszeiger/Rücksprungadresse

  rflags Status/Condition Codes

  cs Aktuelles Code Segment

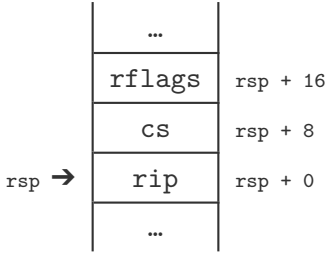


Zustandsicherung (1)

Was muss **mindestens** gesichert werden?

- CPU sichert automatisch

  rip Instruktionszeiger/Rücksprungadresse
  rflags Status/Condition Codes
  cs Aktuelles Code Segment



- Wiederherstellung des ursprünglichen Prozessorzustandes durch Befehl `iretq`

Unterbrechungsbehandlung

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

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

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

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

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

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

Unterbrechungsbehandlung

unter Verwendung einer Hochsprache

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

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

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Zustandssicherung (2)

Was ist mit den restlichen Registern?

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

Aufteilung der Register in zwei Gruppen

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

Aufteilung der Register in zwei Gruppen

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

Aufteilung der Register in zwei Gruppen

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

Unterbrechungsbehandlungen müssen flüchtige Register sichern!

Kontextsicherung in der Unterbrechungsbehandlung

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

Kontextsicherung in der Unterbrechungsbehandlung

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

Kontextsicherung in der Unterbrechungsbehandlung

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

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

Kontextsicherung in der Unterbrechungsbehandlung

```
;; 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.
}
```

Kontextsicherung in der Unterbrechungsbehandlung

```
;; 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

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

Parameterübergabe

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

Parameterübergabe

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

Kontextsicherung in der Unterbrechungsbehandlung

```
;; 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.
}
```

Kontextsicherung in der 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++
struct Context {
    uint64_t r11;
    // ...
    uint64_t rcx;
    uint64_t rax;



} __attribute__((packed));

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

Kontextsicherung in der 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++
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.
}
```

Dann schlug ein Blitz ein …

Dann schlug ein Blitz ein …

Interruptvektoren (x86/x64)

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

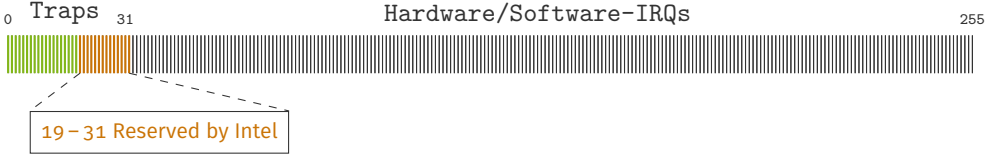
Interruptvektoren (x86/x64)

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

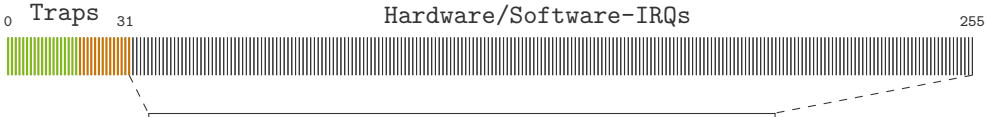
Interruptvektoren (x86/x64)

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

Unterbrechungsbehandlung

```
01
02  ;; Assembler
03  interrupt_entry:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11
12
13    call interrupt_handler
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

```
01
02  // C++
03  extern "C"
04  void interrupt_handler(
05      Context* c
06
07    )
08  {
09      // Magie:
10
11
12
13
14
15
16
17
18  }
19
```

Unterbrechungsbehandlung

```
01
02  ;; Assembler
03  interrupt_entry:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11
12
13    call interrupt_handler
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

```
01
02  // C++
03  extern "C"
04  void interrupt_handler(
05      Context* c,
06      uint32_t vector
07    )
08  {
09      // Magie:
10      switch (vector){
11          case KBD:
12              kbd.magic();
13              break;
14          case TMR:
15              tmr.magic();
16              break;
17      }
18  }
19
```

Unterbrechungsbehandlung für Vektor 6

```
01
02  ;; Assembler
03  interrupt_entry_6:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11    ;; Vektornummer
12    mov rsi, 6
13    call interrupt_handler
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

```
01
02 // C++
03 extern "C"
04 void interrupt_handler(
05     Context* c,
06     uint32_t vector
07   )
08 {
09     // Magie:
10     switch (vector){
11         case KBD:
12             kbd.magic();
13             break;
14         case TMR:
15             tmr.magic();
16             break;
17     }
18 }
19
```

Unterbrechungsbehandlung beim Binden

```
01
02  ;; Assembler
03  interrupt_entry_6:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11    ;; Vektornummer
12    mov rsi, 6
13    call 0x10070f0 <interrupt_handler>
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

```
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>
```

Unterbrechungsbehandlung beim Binden

```
01
02  ;; Assembler
03  interrupt_entry_6:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11    ;; Vektornummer
12    mov rsi, 6
13    call interrupt_handler
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

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

Speicheradressen beim **Binden**:

```
100 70f0 <interrupt_handler>

100 0230 <interrupt_entry_6>
```

Unterbrechungsbehandlung beim Binden

```
01
02  ;; Assembler
03  interrupt_entry_6:
04    ;; Kontext sichern
05    push rax
06    push rcx
07    ;; ...
08    push r11
09    ;; Pointer auf Stack
10    mov rdi, rsp
11    ;; Vektornummer
12    mov rsi, 6
13    call interrupt_handler
14    ;; wiederherstellen
15    pop r11
16    ;; ...
17    pop rcx
18    pop rax
19    iretq
20
```

```
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 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

Generische Unterbrechungsbehandlungen

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

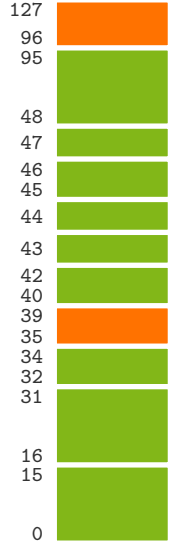
```
01 heinloth:~/oostubs$ make
02 ASM  interrupt/handler.asm
03 CXX  interrupt/handler.cc
04 LD   .build/system64
```

Speicheradressen beim **Binden**:

```
100 70f0 <interrupt_handler>
         ...
100 0230 <interrupt_entry_6>
         ...
100 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



Unused – muss `0` sein

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

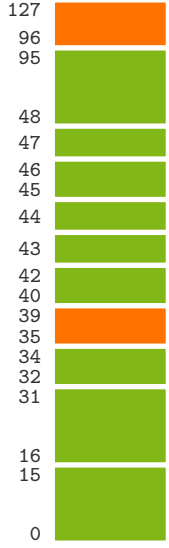
Unused – muss `0` sein

Interrupt Stack Table

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

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

Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6



Unused – muss `0` sein

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

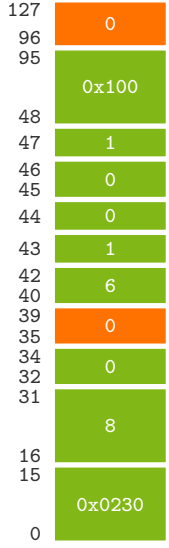
Interrupt Stack Table

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

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

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

Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6



Unused – muss 0 sein

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

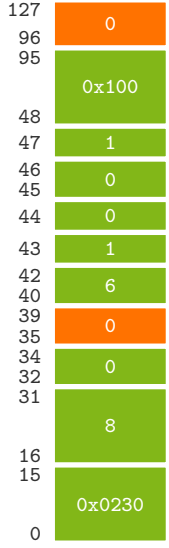
Interrupt Stack Table

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

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

für 100 0230 **<interrupt_entry_6>**

Interrupt Deskriptor – Beispiel für Unterbrechung Nr. 6



Unused – muss 0 sein

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

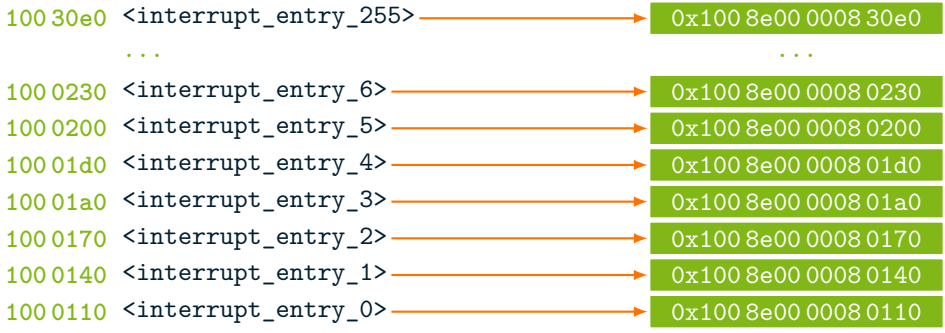
Offset (low): *unterer* Teil der Einsprungsadresse für die Interruptbehandlung

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

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>
```

Interrupt Deskriptor Tabelle (IDT)

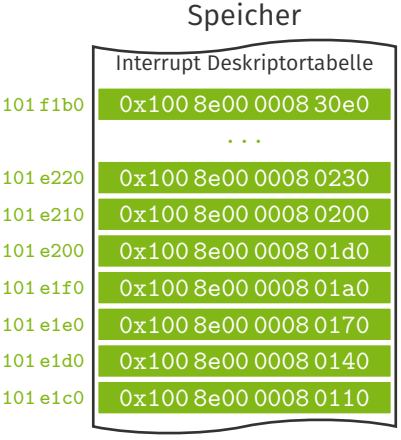


Interrupt Deskriptor Tabelle (IDT)

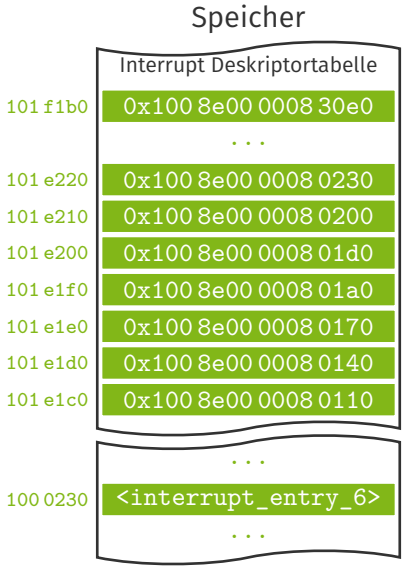
```
100 30e0 <interrupt_entry_255>
         ...
100 0230 <interrupt_entry_6>
100 0200 <interrupt_entry_5>
100 01d0 <interrupt_entry_4>
100 01a0 <interrupt_entry_3>
100 0170 <interrupt_entry_2>
100 0140 <interrupt_entry_1>
100 0110 <interrupt_entry_0>
```

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>
```

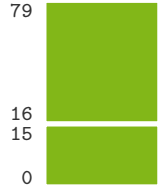


Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

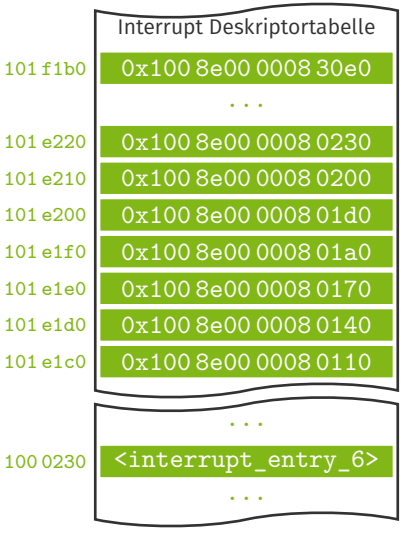
IDT-Register `idtr`



Basis: Startadresse der IDT

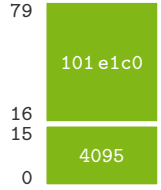
Limit: Bytes *Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`

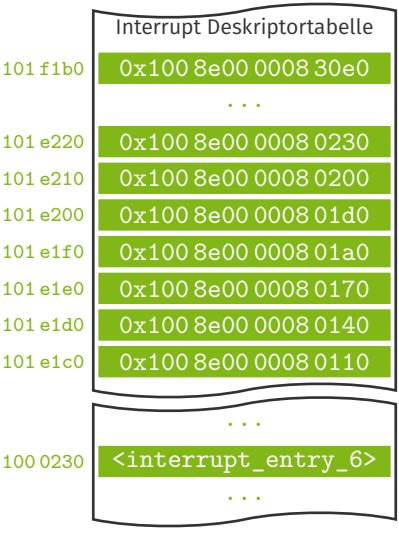


Basis: Startadresse der IDT

Limit: Bytes

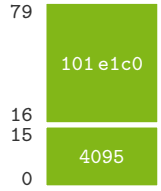
*Einträge * 16 - 1*

Speicher



Interrupt Deskriptor Tabelle (IDT)

IDT-Register `idtr`



Basis: Startadresse der IDT

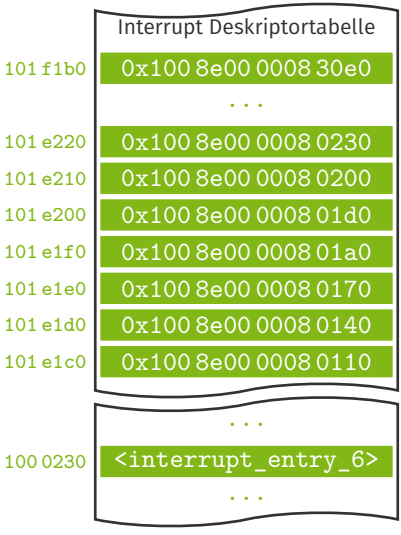
Limit: Bytes *Einträge * 16 - 1*

Instruktionen:

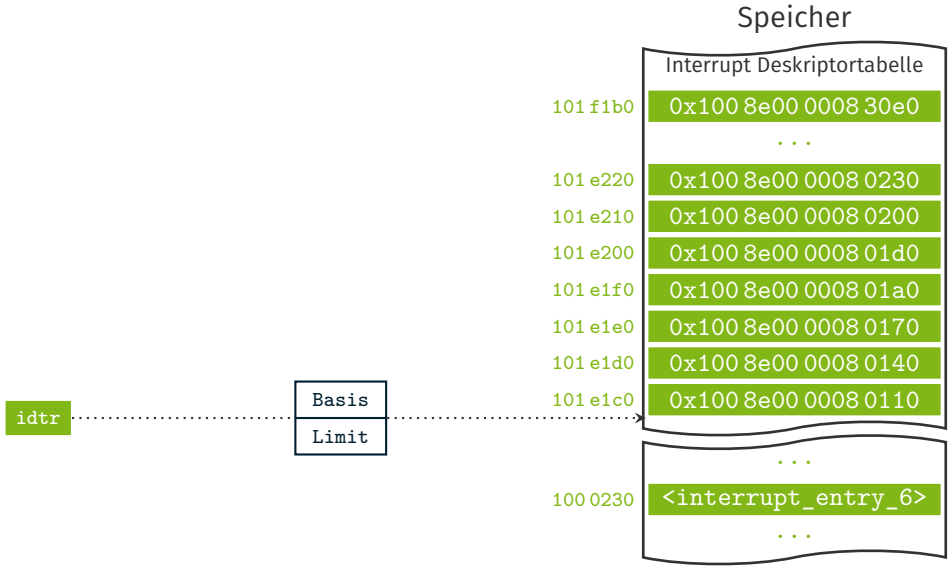
`lidt` in Register laden

`sidt` aus Register lesen

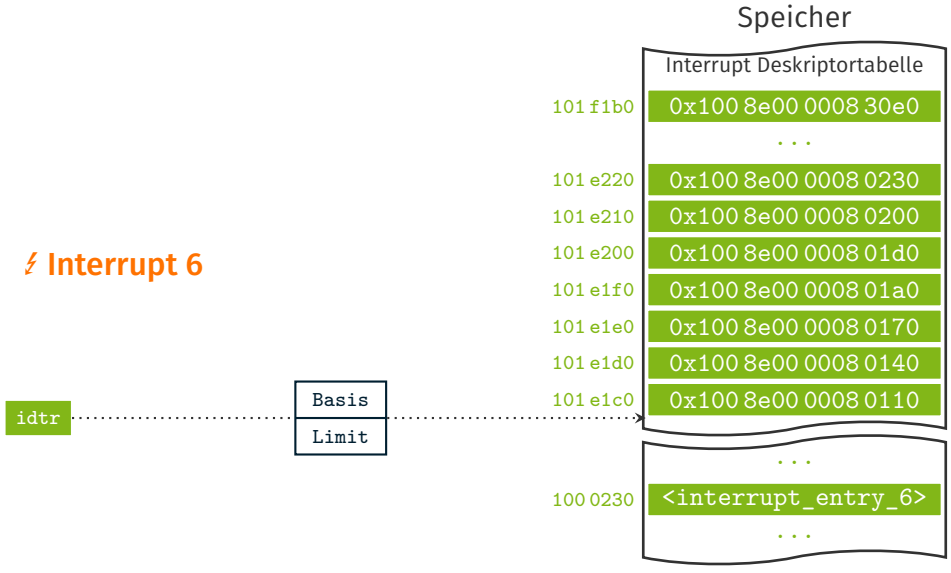
Speicher



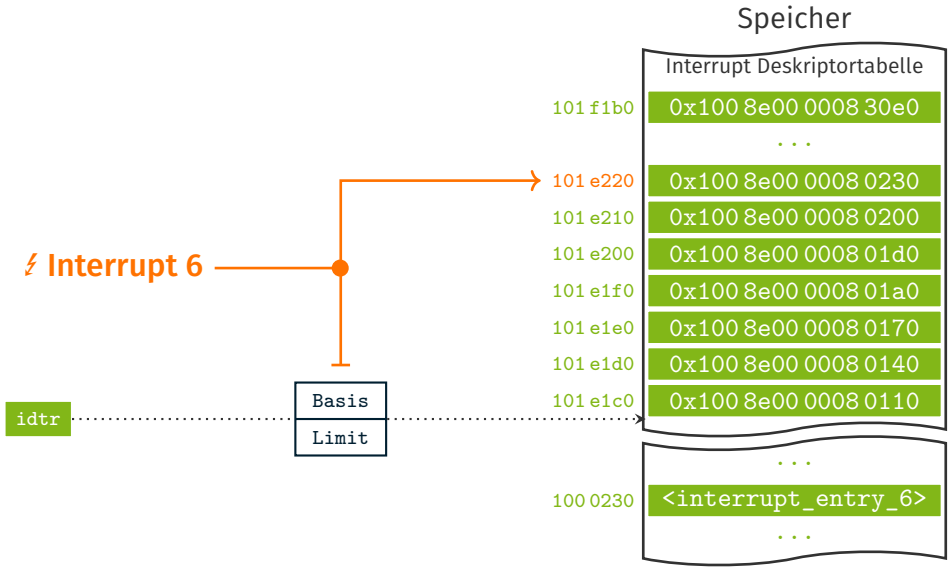
Interrupt Deskriptor Tabelle (IDT)



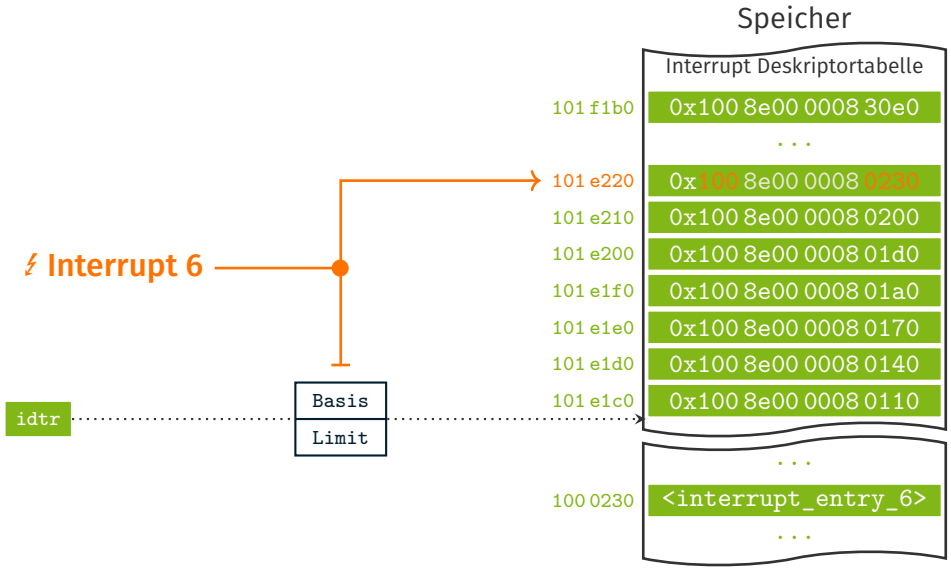
Interrupt Deskriptor Tabelle (IDT)



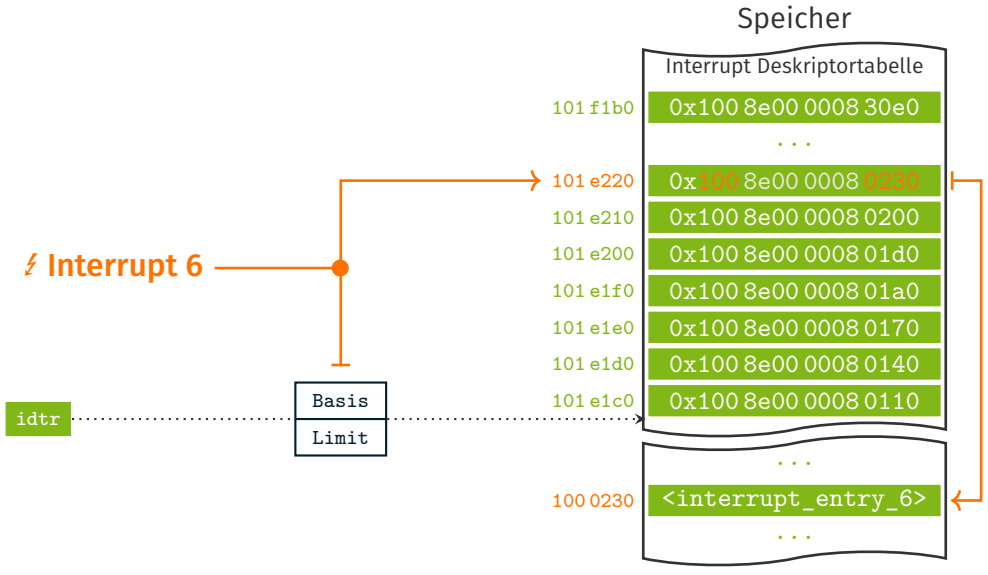
Interrupt Deskriptor Tabelle (IDT)



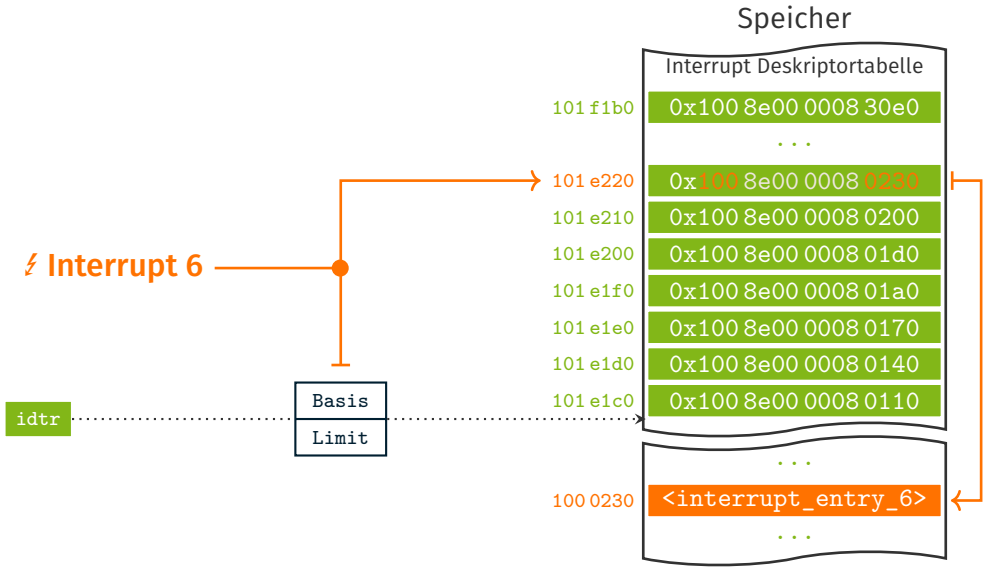
Interrupt Deskriptor Tabelle (IDT)



Interrupt Deskriptor Tabelle (IDT)

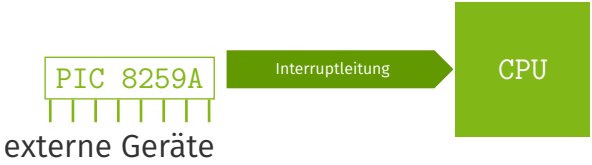


Interrupt Deskriptor Tabelle (IDT)



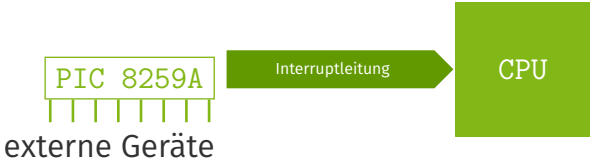
Unterbrechungen durch externe Geräte bei x86-CPUs

Unterbrechungen durch externe Geräte bei x86-CPUs



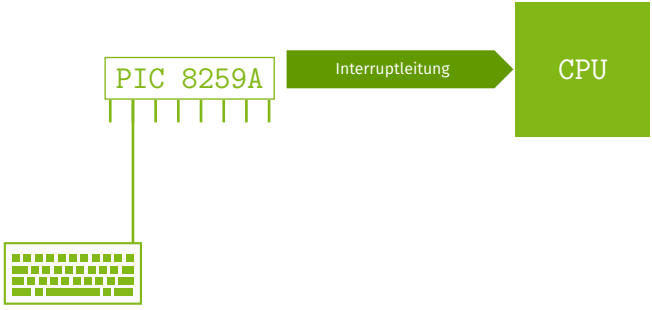
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller

Unterbrechungen durch externe Geräte bei x86-CPUs



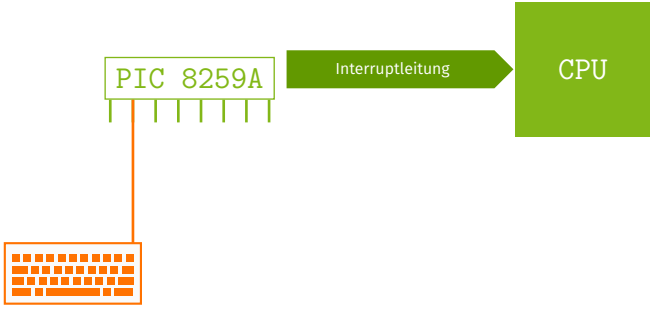
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



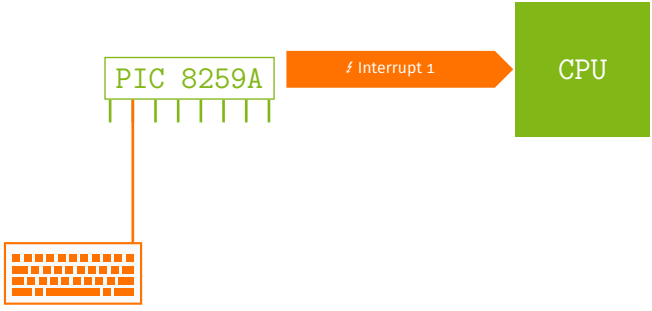
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



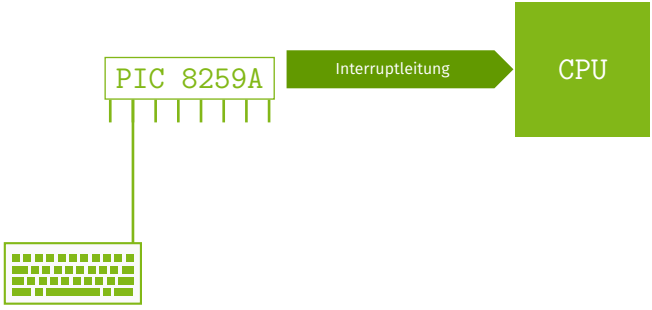
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



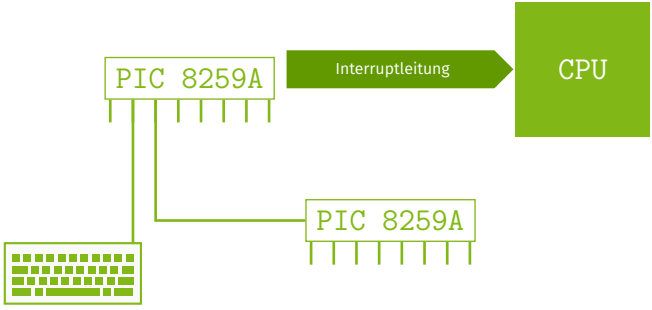
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



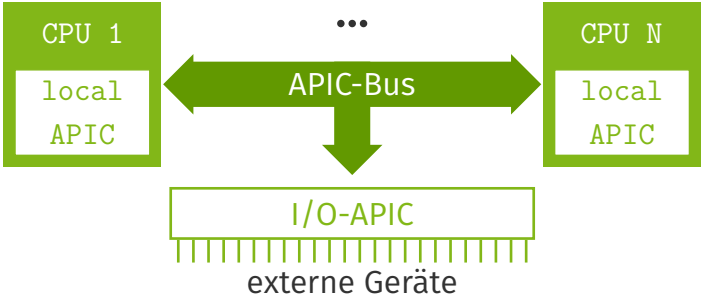
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports

Unterbrechungen durch externe Geräte bei x86-CPUs



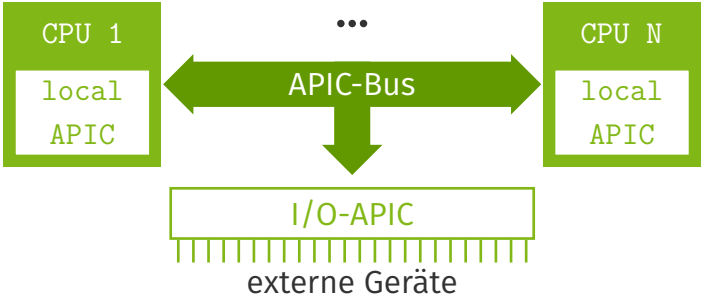
- Anschluss von mehreren Geräten durch Programmable Interrupt Controller
  - feste Prioritätenreihenfolge
  - Interruptvektornummer bedingt änderbar (Vielfaches von 8)
  - Konfiguration über I/O-Ports
- Erweiterung von 8 auf 15 Geräte durch Kaskadierung

Aufbau der APIC-Architektur



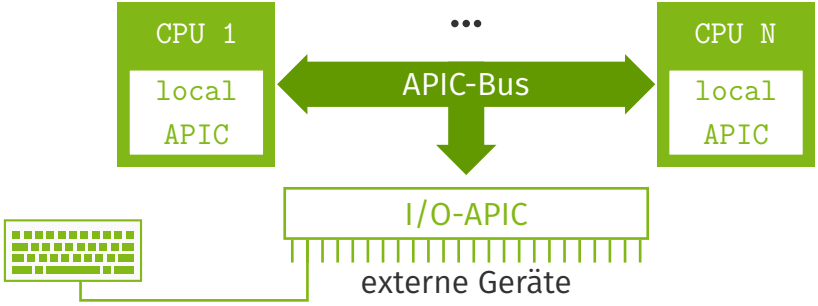
Aufteilung in lokalen APIC und I/O APIC

Aufbau der APIC-Architektur



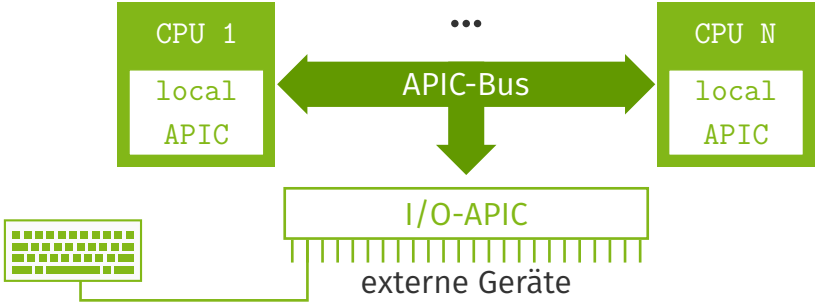
I/O-APIC dient zum Anschluss der Geräte

Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

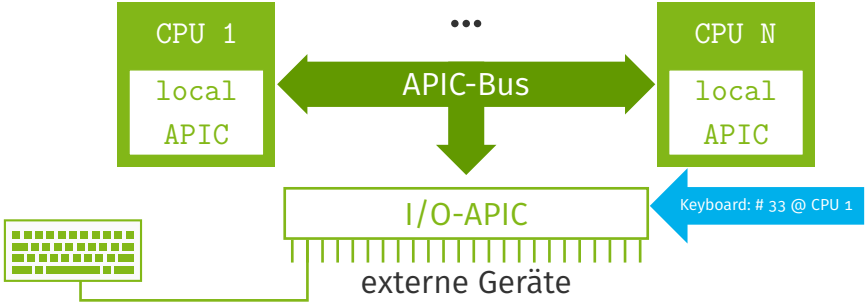
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

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

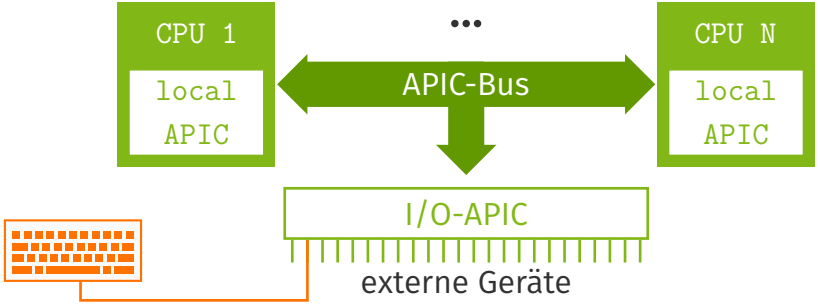
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

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

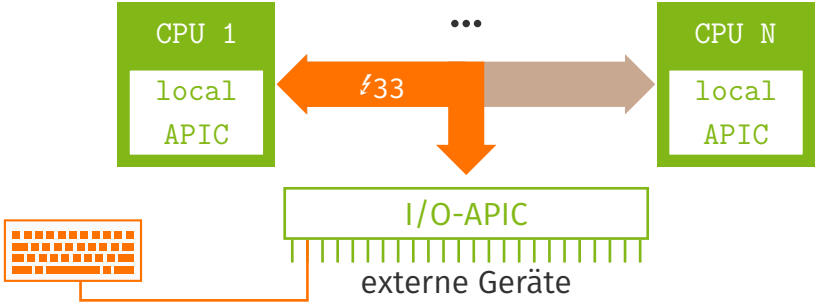
Aufbau der APIC-Architektur



I/O-APIC dient zum Anschluss der Geräte

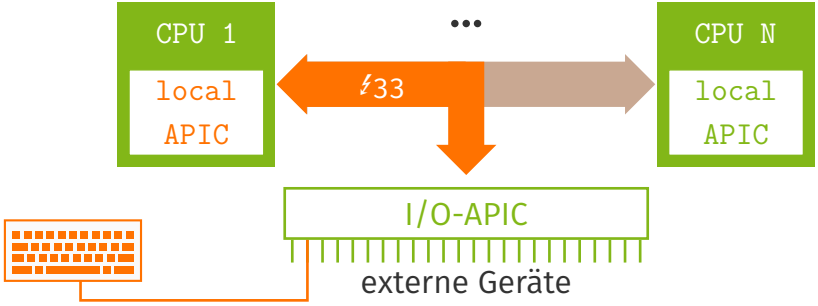
- Zuweisung von beliebigen Vektornummern
- Aktivieren und Deaktivieren von einzelnen Interruptquellen
- Zuweisung von Zielprozessoren für einzelnen Interrupts in MP-Systemen

Aufbau der APIC-Architektur



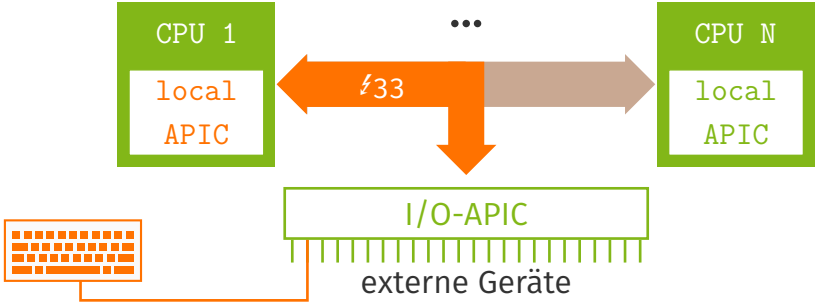
Interrupts werden zu Nachrichten auf dem APIC-Bus

Aufbau der APIC-Architektur



Empfang durch Local APIC

Aufbau der APIC-Architektur



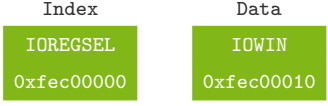
Empfang durch Local APIC

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

Programmierung des Intel I/O-APIC

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

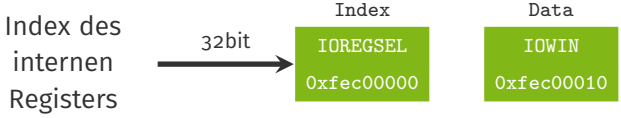
- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister



Programmierung des Intel I/O-APIC

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

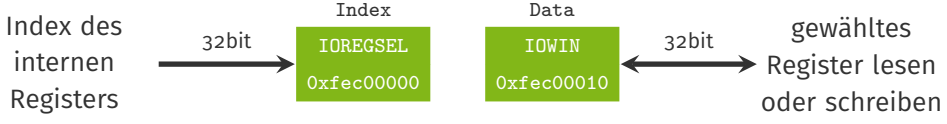
- Jedoch keine direkte Abbildung von internen Registern auf Speicheradressen
- *Umweg* über ein Index- und Datenregister



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

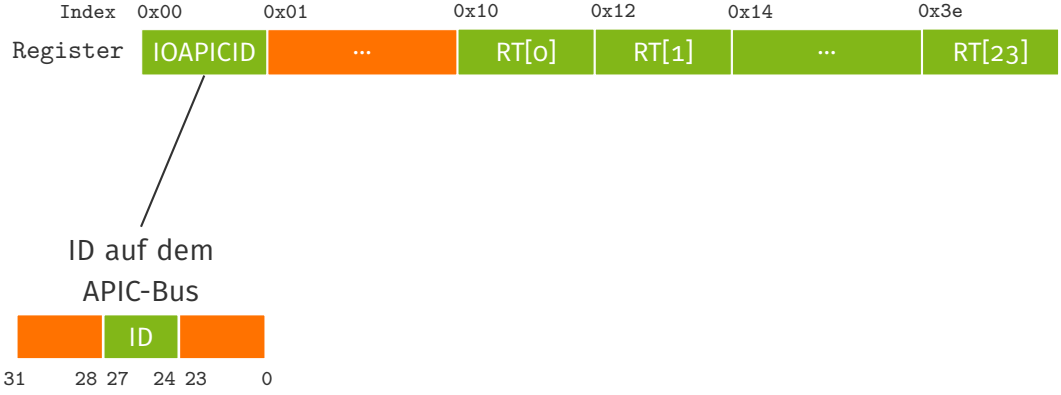


Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs

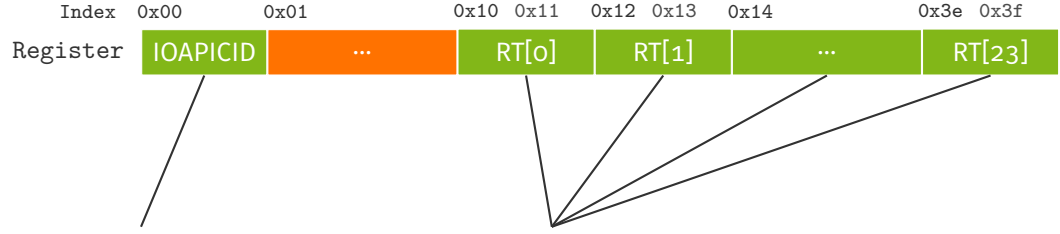
Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



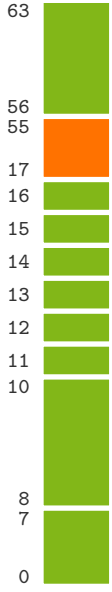
Programmierung des Intel I/O-APIC (2)

Interne Register des I/O-APICs



- 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

Programmierung des APIC Systems

Wo ist welches Gerät angeschlossen?

Programmierung des APIC Systems

Wo ist welches Gerät angeschlossen?

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

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)

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)
- Bei uns stellt `APIC` die relevanten Teile dieser Informationen bereit

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)
- 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`)

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)
- 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

Programmierung des APIC Systems

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

Programmierung des APIC Systems

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

Programmierung des APIC Systems

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

Programmierung des APIC Systems

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

Programmierung des APIC Systems

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

Programmierung des APIC Systems

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

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)

Vorbereitung

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, den 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, den Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen `interrupt_entry` schreiben ✓

Vorbereitung

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden, den Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen `interrupt_entry` schreiben ✓
  - **IDT** füllen und laden ✓

Vorbereitung

- **I/O APIC** initialisieren
  - **I/O APIC ID** setzen
  - Einträge in **Redirection Table** initialisieren (deaktivieren)
- **Keyboard** konfigurieren
  - Anmelden bei der **Plugbox**
  - Tastaturslot herausfinden, den Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen `interrupt_entry` schreiben ✓
  - **IDT** füllen und laden ✓
  - 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, den Eintrag in der **Redirection Table** konfigurieren und aktivieren
  - Tastaturbuffer leeren
- **Interruptbehandlung** erstellen
  - Einsprungsroutinen `interrupt_entry` schreiben ✓
  - **IDT** füllen und laden ✓
  - Ereignisbehandlung in `interrupt_handler` mittels **Plugbox**
- Interrupts mit `Core::Interrupt::enable()` aktivieren

Ablauf

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**
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**
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

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**
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

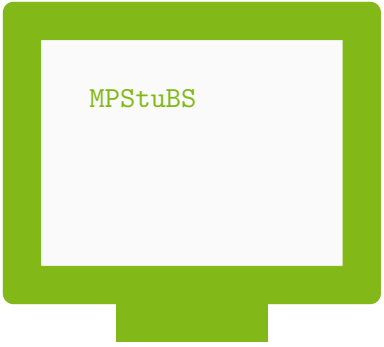
Ablauf

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**
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

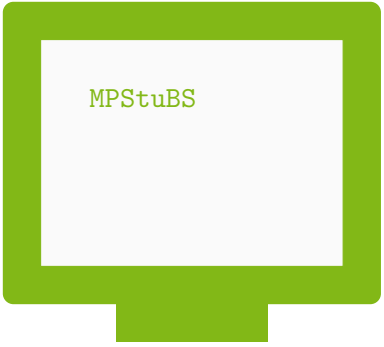
Ablauf

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**
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

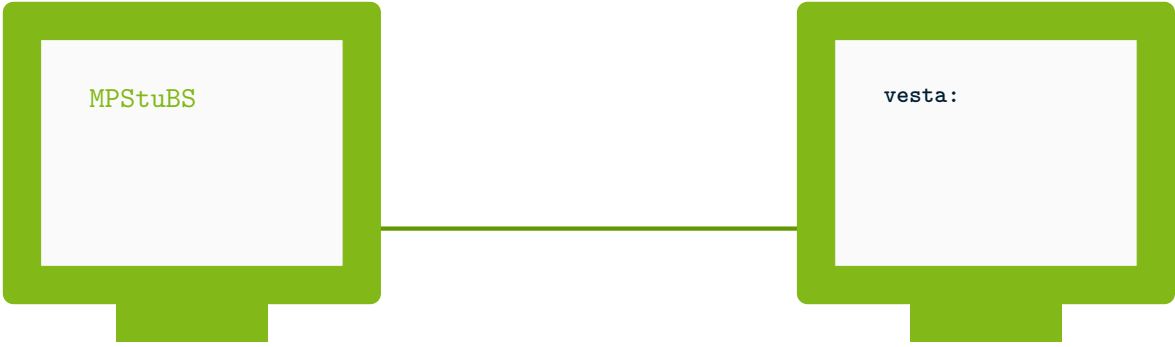
Remote GDB



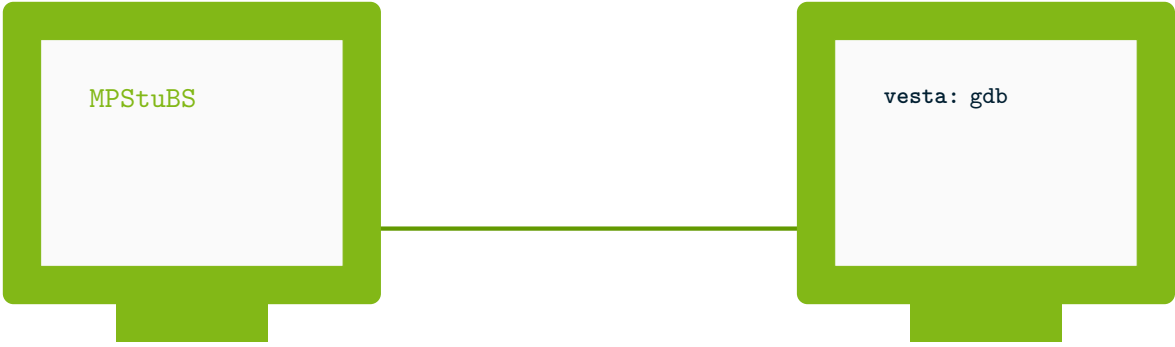
Remote GDB



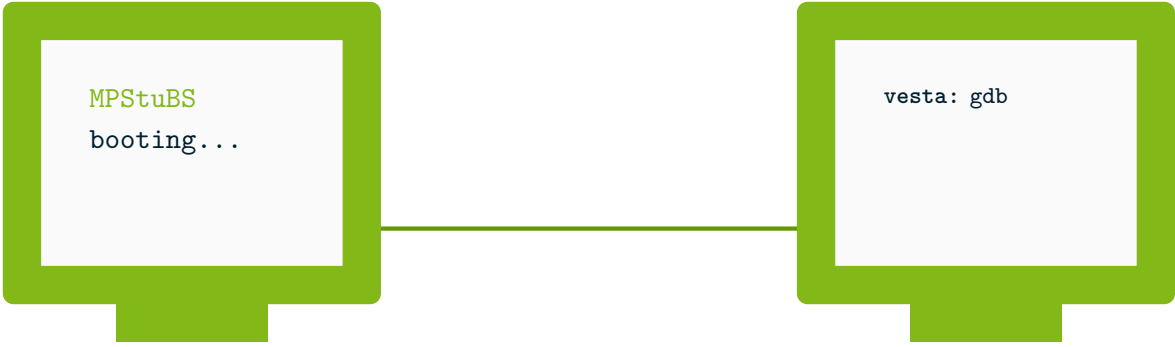
Remote GDB



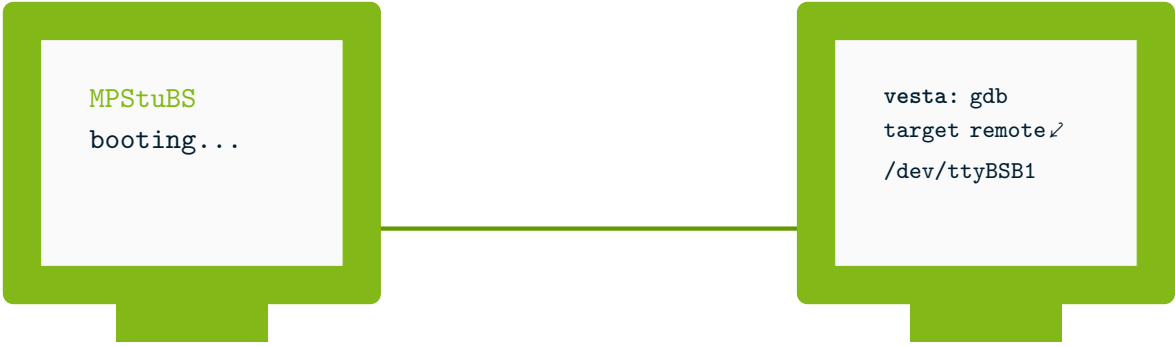
Remote GDB



Remote GDB

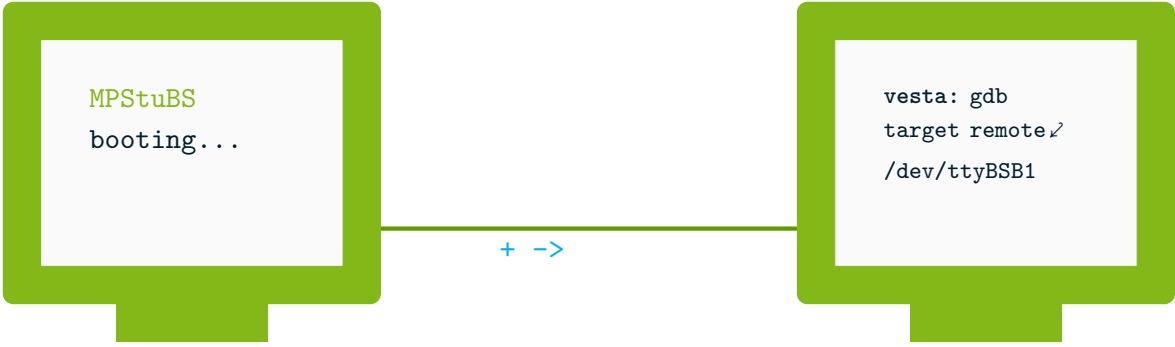


Remote GDB

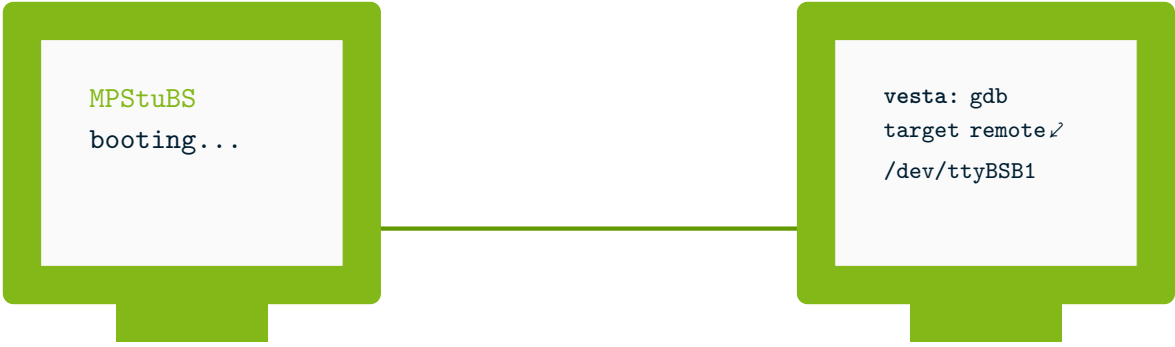


Remote GDB

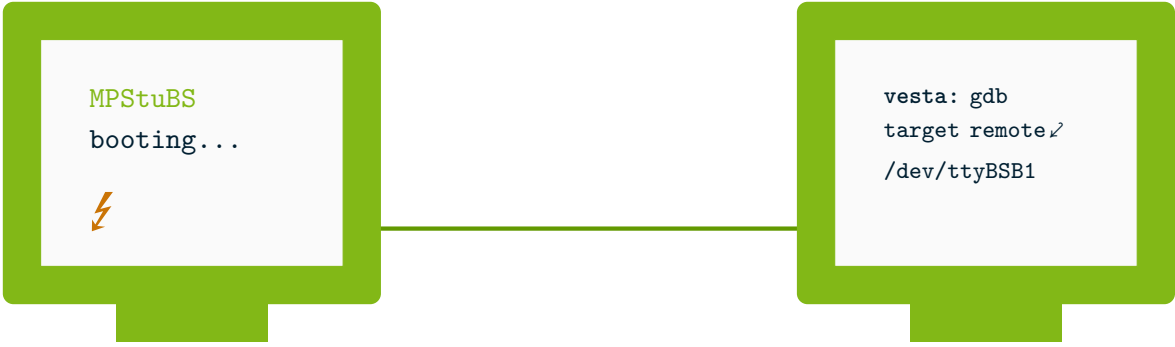
Remote GDB



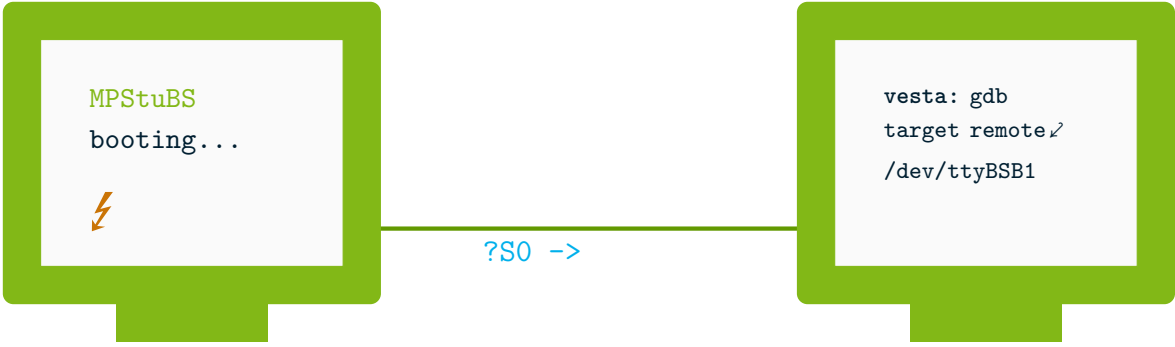
Remote GDB



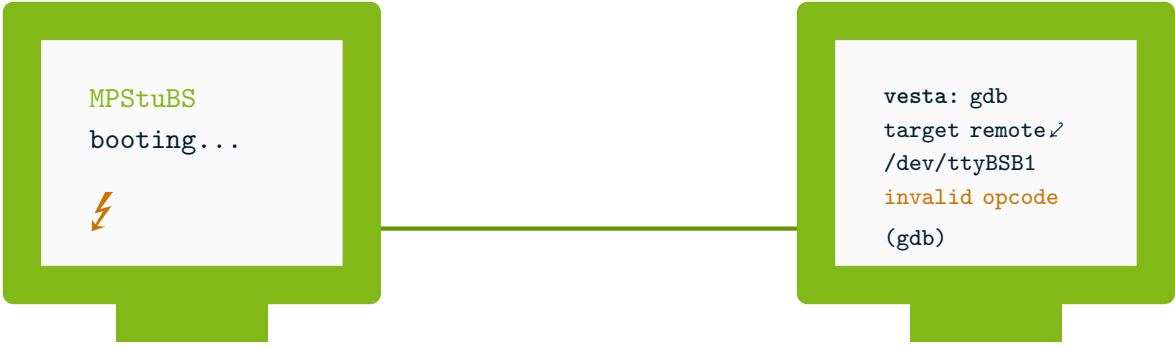
Remote GDB



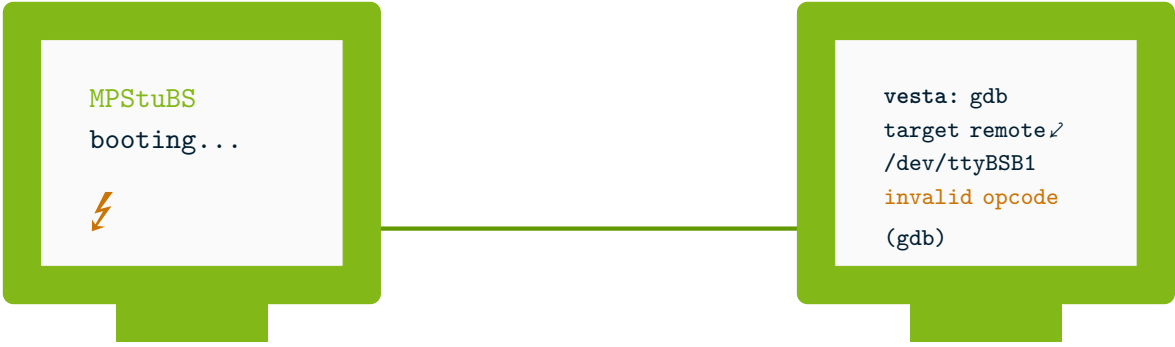
Remote GDB



Remote GDB

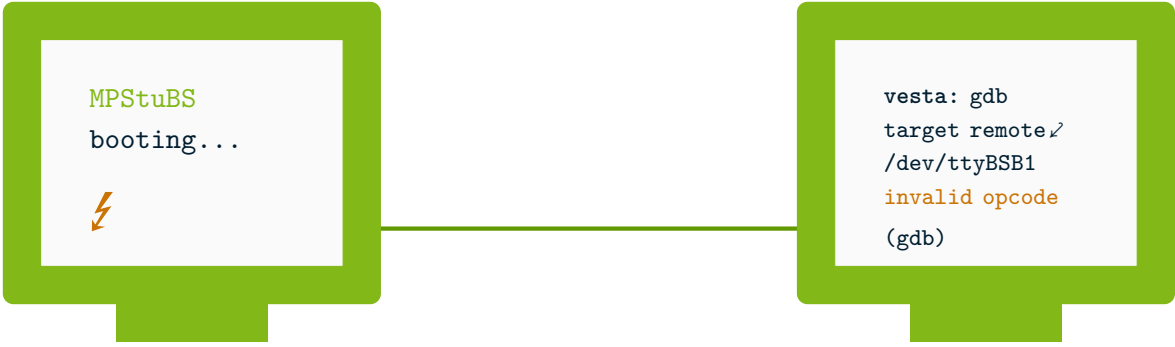

MPStuBS
booting...
vesta: gdb
target remote↙
/dev/ttyBSB1
invalid opcode
(gdb)

Remote GDB



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

Remote GDB



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

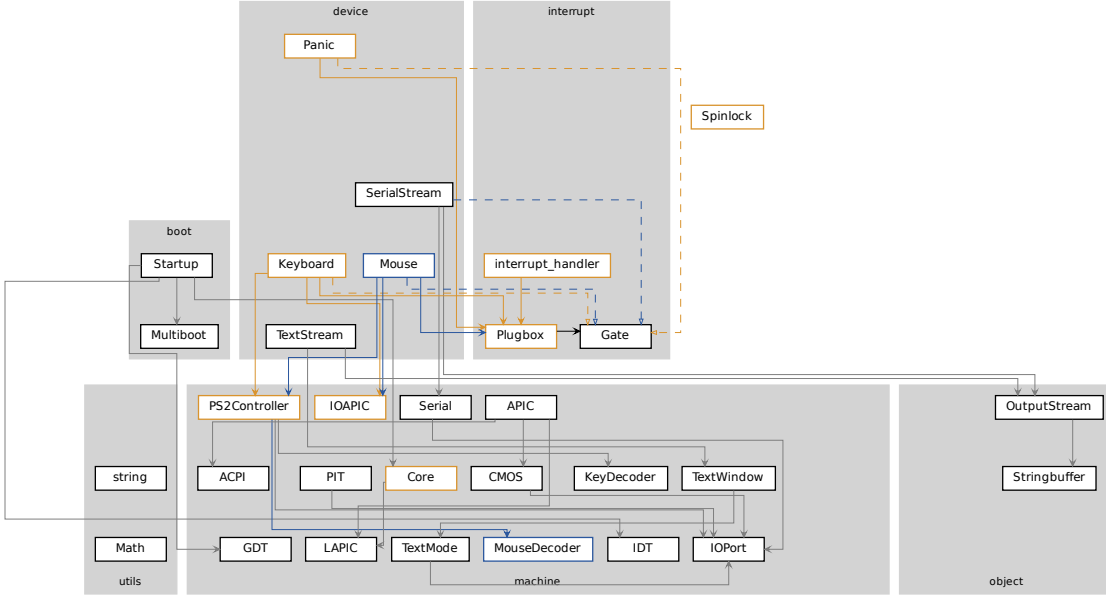
Aufgabe 2

Lernziele

- Behandlung asynchroner Ereignisse
- Problematik und Schutz kritischer Abschnitte

Aufgaben

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


device
interrupt
Panic
Spinlock
SerialStream
boot
Startup
Keyboard
Mouse
interrupt_handler
Multiboot
TextStream
Plugbox
Gate
PS2Controller
IOAPIC
Serial
APIC
OutputStream
string
ACPI
PIT
Core
CMOS
KeyDecoder
TextWindow
Stringbuffer
Math
GDT
LAPIC
TextMode
MouseDecoder
IDT
IOPort
utils
machine
object

Vergnügen mit KVM

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

Vergnügen mit KVM

KVM nutzt ggf. Vector Hashing

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

(weitere Details siehe FAQ auf der Webseite)

Gibt es noch Fragen?

Abgabe der 2. Aufgabe bis Mittwoch, 13. November