OSCI Exercises

Interrupts

Winter term 2026/27

Manuel Vögele & Benedict Herzog

Bochum Operating Systems and System Software Group (BOSS)

RUHR UNIVERSITÄT BOCHUM

RUB

BOCHUM OPERATING SYSTEMS & SYSTEM SOFTWARE

Interrupts and Traps


CPU


CPU

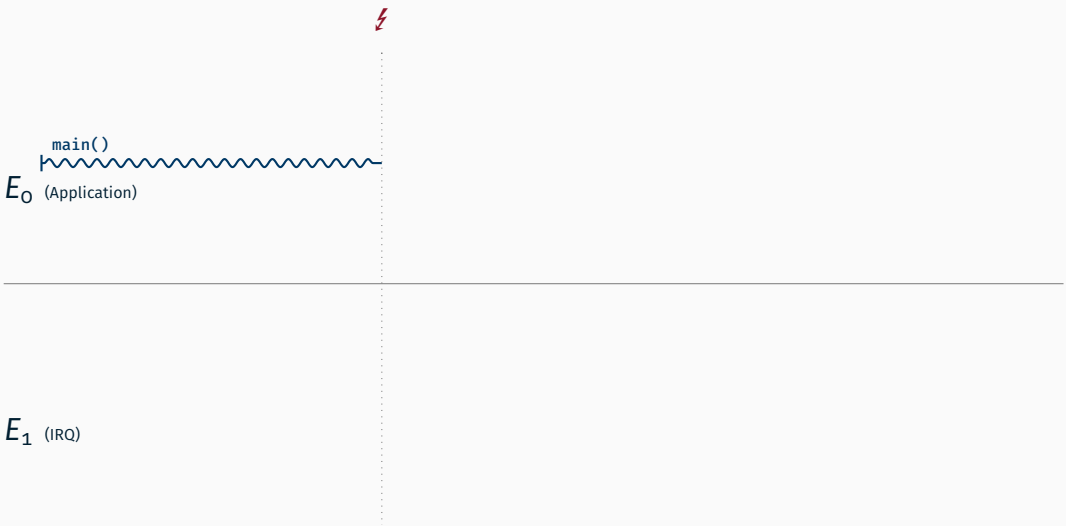

CPU

Interrupts (Application's View)

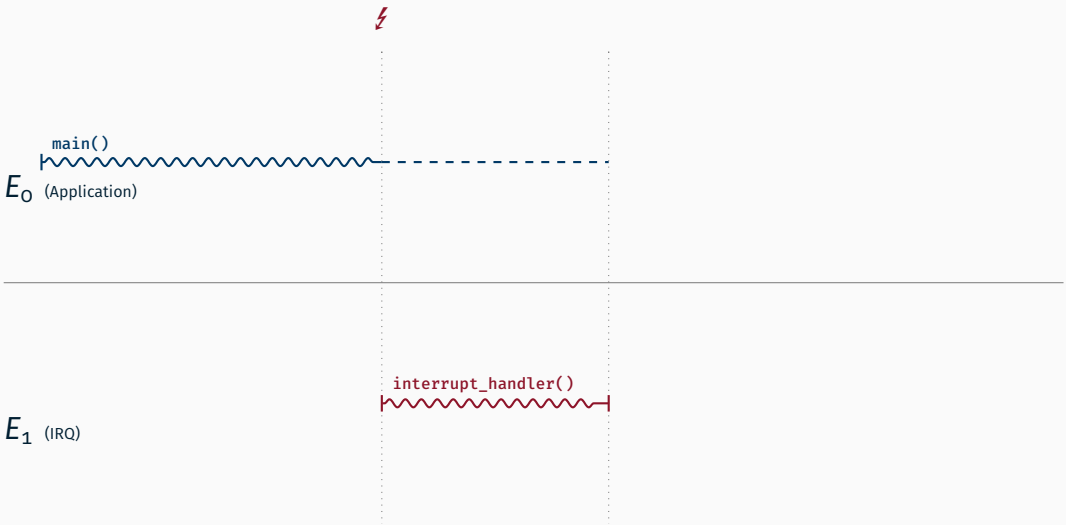


E_1 (IRQ)

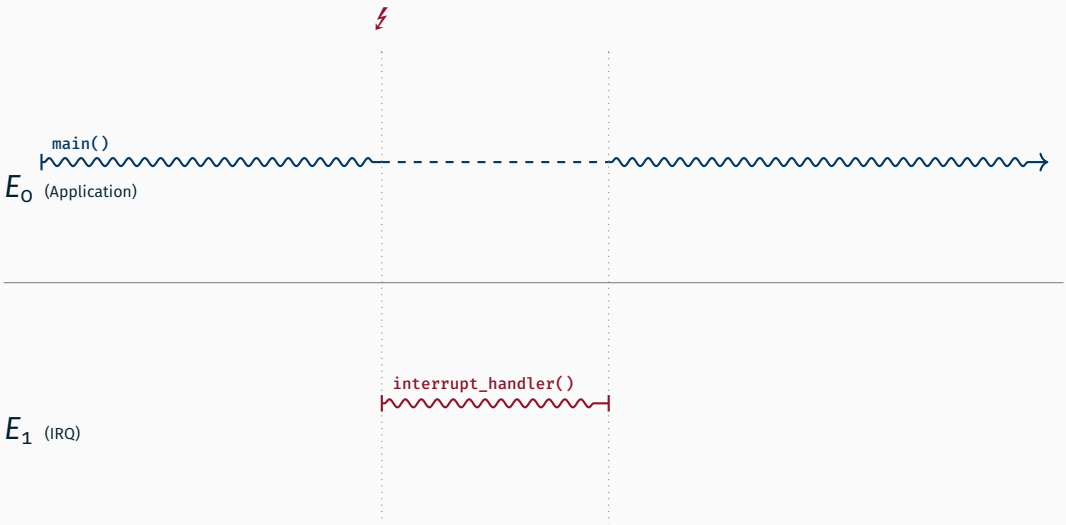
Interrupts (Application's View)



Interrupts (Application's View)



Interrupts (Application's View)

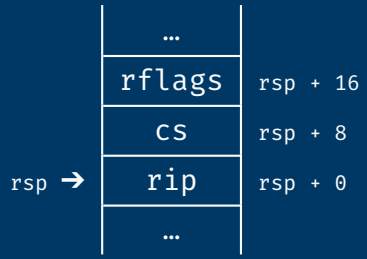


Minimal state to save?

Minimal state to save?

- CPU automatically saves

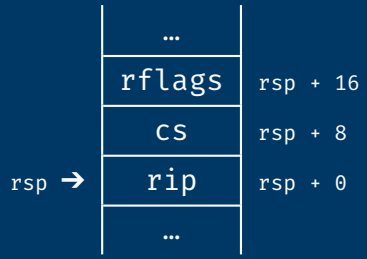
rip Instruction Pointer/Return Address
rflags State/Condition Codes
cs Current code segment



Minimal state to save?

- CPU automatically saves

rip	Instruction Pointer/Return Address
rflags	State/Condition Codes
cs	Current code segment



- Previous processor state can be restored by calling `iretq`

Interrupt Handling

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ


  iretq
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

```
// C++

void interrupt_handler()
{
    // Magic
}
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

```
// C++

void interrupt_handler()
{
    // Magic
}
```

```
01 voegele:~/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'
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

```
// C++

void interrupt_handler()
{
    // Magic
}
```

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

0000000000000000 <_Z17interrupt_handlerv>:
   0:   c3                      retq
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

```
// C++ (with C linkage)
extern "C"
void interrupt_handler()
{
    // Magic
}
```

Interrupt Handling

using a high level language

```
;; Assembler
interrupt_entry:
  ;; Handle IRQ
  ;; in high level language
  call interrupt_handler
  iretq
```

```
// C++ (with C linkage)
extern "C"
void interrupt_handler()
{
    // Magic
}
```

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

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

What about the remaining registers?

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code
- Saving context when calling functions

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code
- Saving context when calling functions
  1. Calling function saves all registers it uses (caller save)

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code
- Saving context when calling functions
  1. Calling function saves all registers it uses (caller save)
  2. Called function saves all registers it modifies (callee save)

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code
- Saving context when calling functions
  1. Calling function saves all registers it uses (caller save)
  2. Called function saves all registers it modifies (callee save)
  3. Some registers are caller saved, some registers are callee saved

What about the remaining registers?

- Must be stored by the assembler code itself.
  - either in the assembly
  - or the compiler already generates the required code
- Saving context when calling functions
  1. Calling function saves all registers it uses (caller save)
  2. Called function saves all registers it modifies (callee save)
  3. Some registers are caller saved, some registers are callee saved
- Usually, 3 is being used
  - This behavior is compiler specific
  - To ensure compatibility of binary code there are conventions (for x64 there are two: *Microsoft* and *System V*)
  - → Calling convention is part of the *Application Binary Interface* (ABI)

Registers are separated into two groups

Registers are separated into two groups

- Volatile registers (*scratch registers, caller-save*)
  - Compiler assumes that suroutines will change the registers
  - Caller must save the registers if needed
  - For x64 (in the *System V* ABI) the registers `rax`, `rcx`, `rdx`, `rsi`, `rdi` and `r8` – `r11` are defined as volatile

Registers are separated into two groups

- Volatile registers (*scratch registers, caller-save*)
  - Compiler assumes that suroutines will change the registers
  - Caller must save the registers if needed
  - For x64 (in the *System V* ABI) the registers `rax`, `rcx`, `rdx`, `rsi`, `rdi` and `r8` – `r11` are defined as volatile
- Non-volatile registers (*non-scratch registers, callee-save*)
  - Compiler assumes that suprotines will leave the registers unchanged
  - Callee must save the register if needed
  - For x64 all other registers are defined as non-volatile: `rbx`, `rbp`, `rsp` und `r12` – `r15`

Registers are separated into two groups

- Volatile registers (*scratch registers, caller-save*)
  - Compiler assumes that suroutines will change the registers
  - Caller must save the registers if needed
  - For x64 (in the *System V* ABI) the registers `rax`, `rcx`, `rdx`, `rsi`, `rdi` and `r8` – `r11` are defined as volatile
- Non-volatile registers (*non-scratch registers, callee-save*)
  - Compiler assumes that suprotines will leave the registers unchanged
  - Callee must save the register if needed
  - For x64 all other registers are defined as non-volatile: `rbx`, `rbp`, `rsp` und `r12` – `r15`



Interrupt handlers need to save volatile registers as well!

Interrupt handling (saving context)

```
;; Assembler
interrupt_entry:




  call interrupt_handler




  iretq
```

Interrupt handling (saving context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11


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

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11


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

```
// C++


extern "C"
void interrupt_handler()
{
    // Magic
}
```

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11


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

```
// C++
struct Context {









} __attribute__((packed));

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

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11


  call interrupt_handler
  ;; restore
  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)
{
    // Magic
}
```

Passing parameters

Passing parameters

- via the stack
  - for x86 this was standard according to *C declaration*

Passing parameters

- via the stack
  - for x86 this was standard according to *C declaration*
- via registers
  - Advantage: faster
  - Downside: number of registers is limited
- combined
  - first parameters via registers, additional parameters via stack

Passing parameters

- via the stack
  - for x86 this was standard according to *C declaration*
- via registers
  - Advantage: faster
  - Downside: number of registers is limited
- combined
  - first parameters via registers, additional parameters via stack
  - according to x64 *System V* ABI:
    - Parameters will be passed via registers in this order `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
    - In user space the folling registers will be used next: `xmm0` – `xmm7`
    - Remaining parameters will be put onto the stack (last parameter will be pushed first)

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11


  call interrupt_handler
  ;; restore
  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)
{
    // Magic
}
```

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  call interrupt_handler
  ;; restore
  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)
{
    // Magic
}
```

Interrupt handling (context)

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  call interrupt_handler
  ;; restore
  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)
{
    // Magic
}
```


CPU


0-255
CPU

Interrupt Vector Table (x86/x64)



Interrupt Vector Table (x86/x64)



Interrupt Vector Table (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

Interrupt Vector Table (x86/x64)



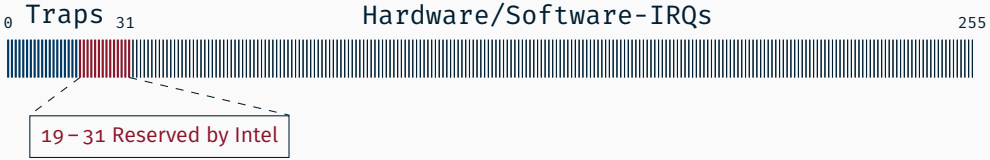
0	**Division-by-Zero**
1	Debug Exception
2	Non-Maskable Interrupt(NMI)
3	**Breakpoint (INT 3)**
4	Overflow Exception
5	Bound Exception
6	**Invalid Opcode**
7	FPU not Available
8	Double Fault
10	Invalid TSS
11	Segment not Present
12	Stack Exception
13	**General Protection Fault**
14	Page Fault
16	Floating-Point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

Interrupt Vector Table (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

Interrupt Vector Table (x86/x64)



Interrupt Vector Table (x86/x64)



32 – 255 entries for IRQs

- Triggered by software via `int <vec#>`
- Triggered by hardware from external devices

Interrupt Vector Table (x86/x64)



Can be masked through instructions

- `cli` (**cl**ear **i**nterrupt flag) Disable interrupt line
- `sti` (**s**e**t** **i**nterrupt flag) Enable interrupt line

Interrupt handling

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp


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

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

  )
{
    // Magic









}
```

Interrupt handling

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp


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

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

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

Interrupt handling

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp


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

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

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

Interrupt handling

```
;; Assembler
interrupt_entry:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp


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

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

Interrupt handling (for vector 6)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp


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

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

Interrupt handling (for vector 6)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, 6
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

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

Interrupt handling (for vector 6)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, 6
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

```
voegele:~/oostubs$ make
ASM   interrupt/handler.asm
CXX   interrupt/handler.cc
LD    .build/system64
```

Interrupt handling (linking)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, 6
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>
```

Interrupt handling (linking)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, 6
  call 0x10070f0 <interrupt_handler>
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>
```

Interrupt handling (linking)

```
Machine code


50
51

41 53

48 89 e7

be 06 00 00 00
e8 a6 6e 00 00

41 5b

59
58
48 cf
```

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>
```

Interrupt handling (linking)

```
Memory      Machine code
address

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>
```

Interrupt handling (linking)

```
Memory address      Machine code

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>

100 0230 <interrupt_entry_6>
```

Interrupt handling (linking)

```
;; Assembler
interrupt_entry_6:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, 6
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
```

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

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>

100 0230 <interrupt_entry_6>
```

Interrupt handling

```
%macro IRQ 1
align 8
interrupt_entry_%1:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, %1
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
%endmacro
```

```
voegele:~/oostubs$ make
ASM   interrupt/handler.asm
CXX   interrupt/handler.cc
LD    .build/system64
```

Memory addresses during **linking**:

```
100 70f0 <interrupt_handler>

100 0230 <interrupt_entry_6>
```

Interrupt handling

```
%macro IRQ 1
align 8
interrupt_entry_%1:
  ;; save context
  push rax
  push rcx
  ;; ...
  push r11
  ;; pointer to stack
  mov rdi, rsp
  ;; vector number
  mov rsi, %1
  call interrupt_handler
  ;; restore
  pop r11
  ;; ...
  pop rcx
  pop rax
  iretq
%endmacro
```

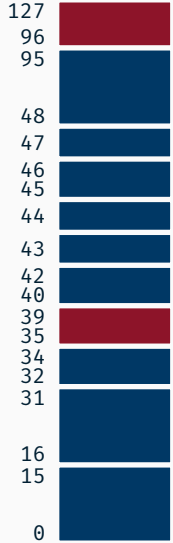
```
01 voegele:~/oostubs$ make
02 ASM   interrupt/handler.asm
03 CXX   interrupt/handler.cc
04 LD    .build/system64
```

Memory addresses during **linking**:

```
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>
```

How does the CPU know the address of the interrupt handler?

Interrupt Descriptor



Unused – must be `0`

Offset (high): *high* part of the jump target for interrhupt handling (z.B. `interrupt_entry_6`)

Present: Entry active `(1)` or inactive `(0)`

Descriptor Privilege Level

Storage Segment: `0` for interrupts and traps

Mode: `16` bit `(0)` or `32/64` bit `(1)`

Type: Task `(5)`, Interrupt `(6)` or Trap `(7)`?

Unused – must be `0`

Interrupt Stack Table

Selector: code segment, into which the interrupt switches (usually the kernel code segment)

Offset (low): *low* part of the jump target for interrupt handling

Interrupt Descriptor (Example)



Unused – must be `0`

Offset (high): *high* part of the jump target for interrhupt handling (z.B. `interrupt_entry_6`)

Present: Entry active (`1`) or inactive (`0`)

Descriptor Privilege Level

Storage Segment: `0` for interrupts and traps

Mode: 16 bit (`0`) or 32/64 bit (`1`)

Type: Task (`5`), Interrupt (`6`) or Trap (`7`)?

Unused – must be `0`

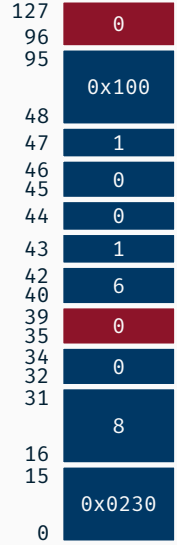
Interrupt Stack Table

Selector: code segment, into which the interrupt switches (usually the kernel code segment)

Offset (low): *low* part of the jump target for interrupt handling

for `100 0230` **<interrupt_entry_6>**

Interrupt Descriptor (Example)



Unused – must be `0`

Offset (high): *high* part of the jump target for interrhupt handling (z.B. `interrupt_entry_6`)

Present: Entry active (`1`) or inactive (`0`)

Descriptor Privilege Level

Storage Segment: `0` for interrupts and traps

Mode: 16 bit (`0`) or 32/64 bit (`1`)

Type: Task (`5`), Interrupt (`6`) or Trap (`7`)?

Unused – must be `0`

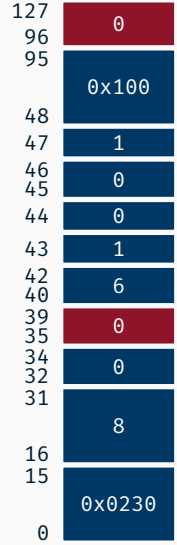
Interrupt Stack Table

Selector: code segment, into which the interrupt switches (usually the kernel code segment)

Offset (low): *low* part of the jump target for interrupt handling

for `100 0230` **<interrupt_entry_6>**

Interrupt Descriptor (Example)



Unused – must be `0`

Offset (high): *high* part of the jump target for interrhupt handling (z.B. `interrupt_entry_6`)

Present: Entry active (`1`) or inactive (`0`)

Descriptor Privilege Level

Storage Segment: `0` for interrupts and traps

Mode: 16 bit (`0`) or 32/64 bit (`1`)

Type: Task (`5`), Interrupt (`6`) or Trap (`7`)?

Unused – must be `0`

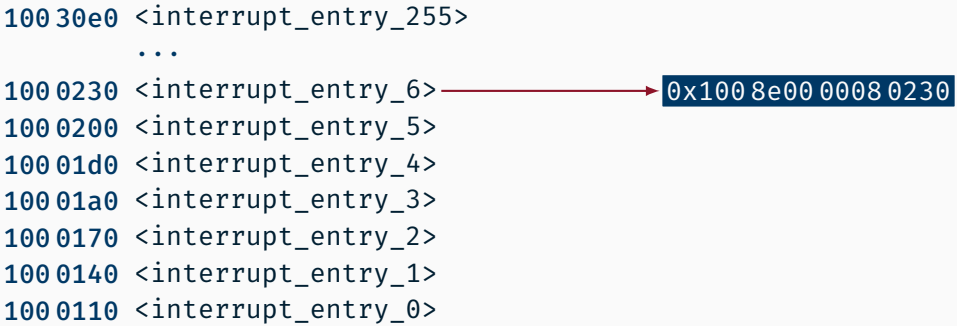
Interrupt Stack Table

Selector: code segment, into which the interrupt switches (usually the kernel code segment)

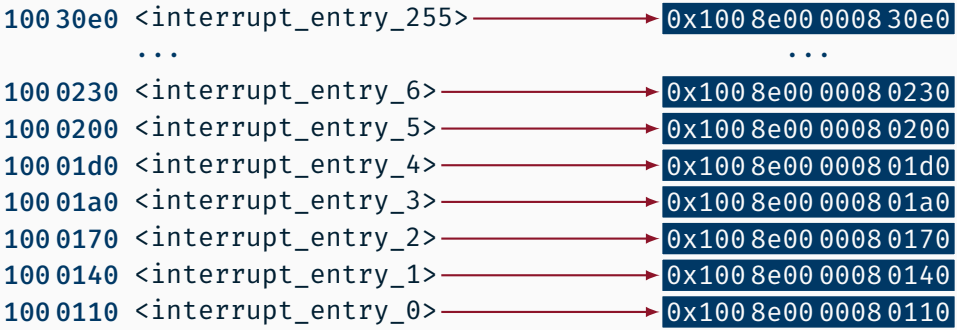
Offset (low): *low* part of the jump target for interrupt handling

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

Interrupt Descriptor Table (IDT)

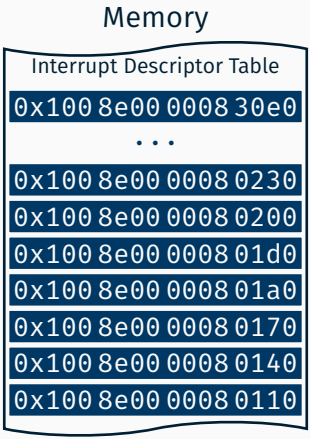


Interrupt Descriptor Table (IDT)



Interrupt Descriptor Table (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>
```

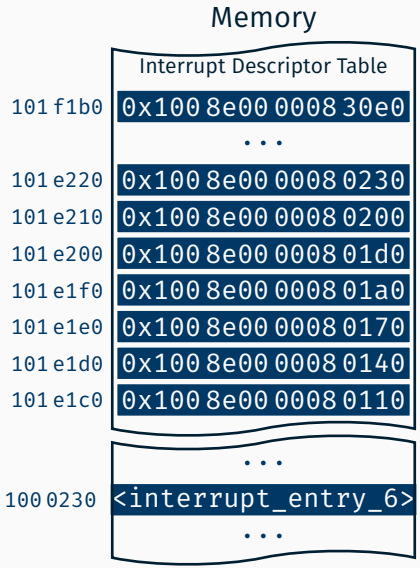


Interrupt Descriptor Table (IDT)

```
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>
```

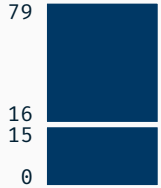


Interrupt Descriptor Table (IDT)



Interrupt Descriptor Table (IDT)

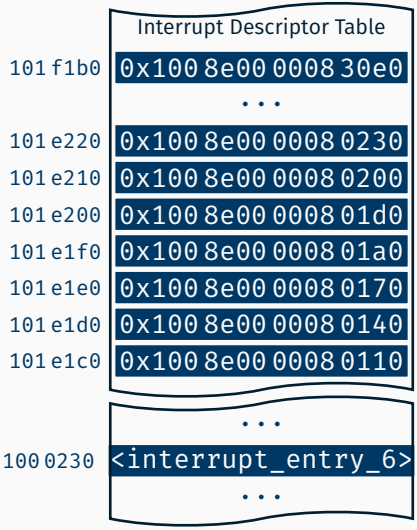
IDT register **`idtr`**



Base: start address of the IDT

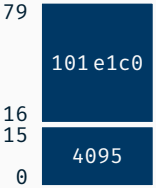
Limit: bytes *entries * 16 - 1*

Memory



Interrupt Descriptor Table (IDT)

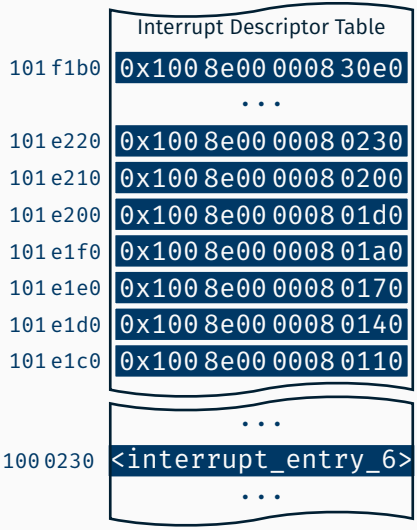
IDT register **`idtr`**



Base: start address of the IDT

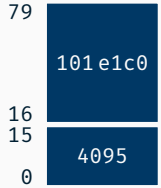
Limit: bytes *entries * 16 - 1*

Memory



Interrupt Descriptor Table (IDT)

IDT register **`idtr`**



Base: start address of the IDT

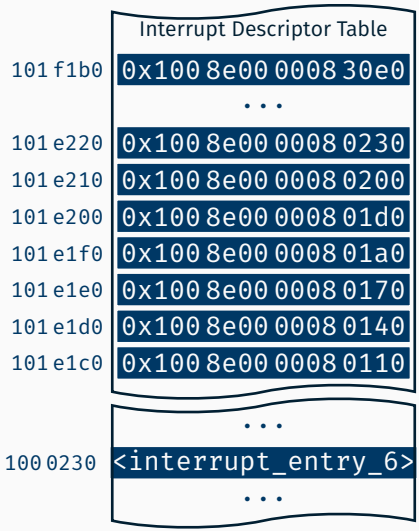
Limit: bytes *entries * 16 - 1*

Instructions:

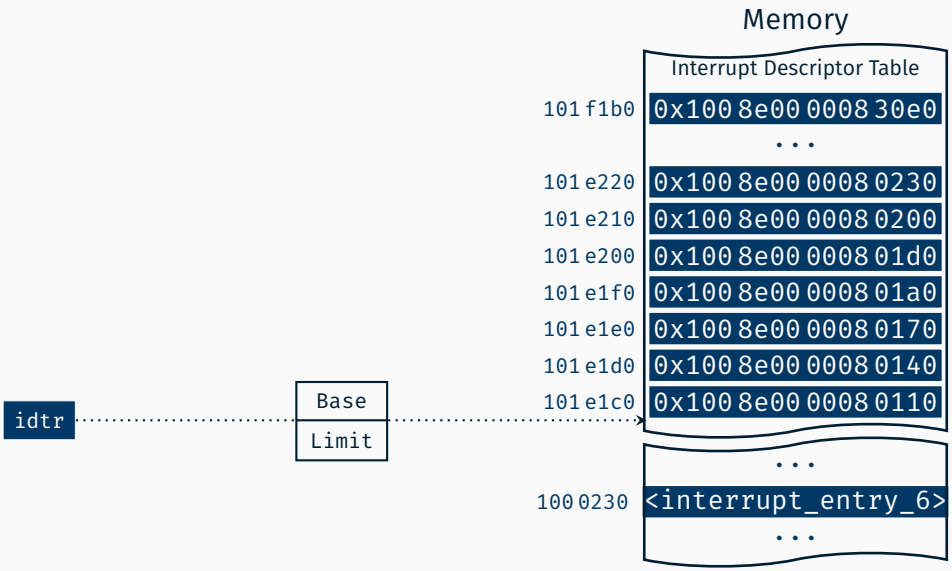
`lidt` load into register

`sidt` write from register

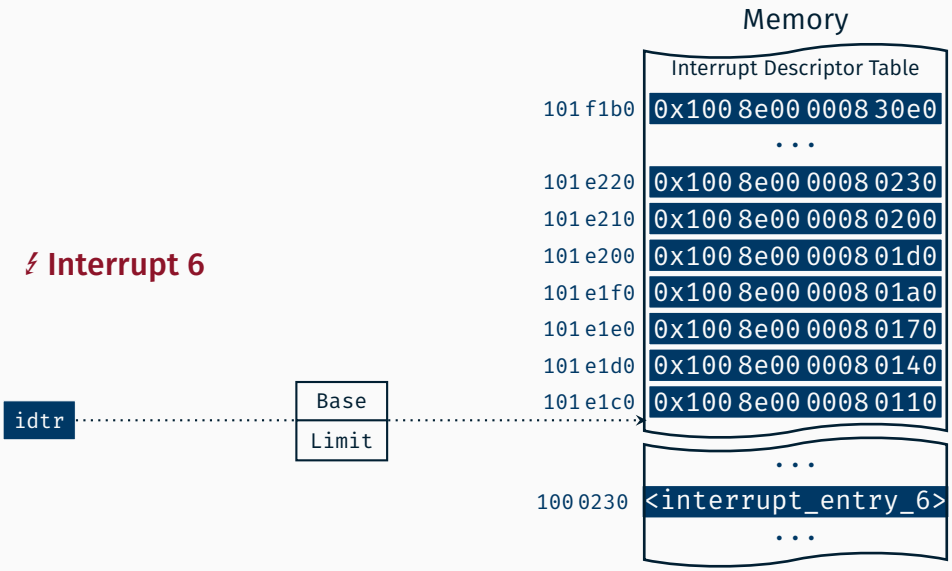
Memory



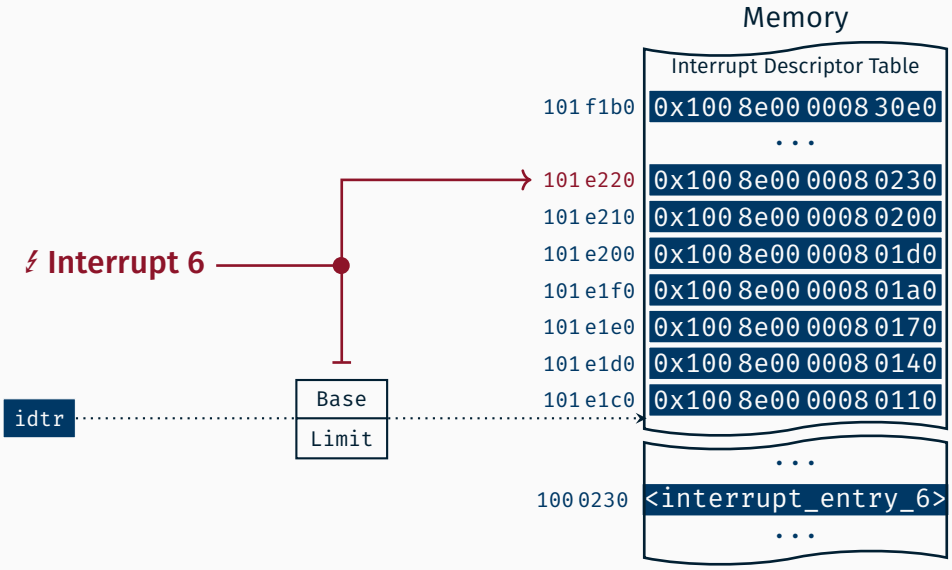
Interrupt Descriptor Table (IDT)



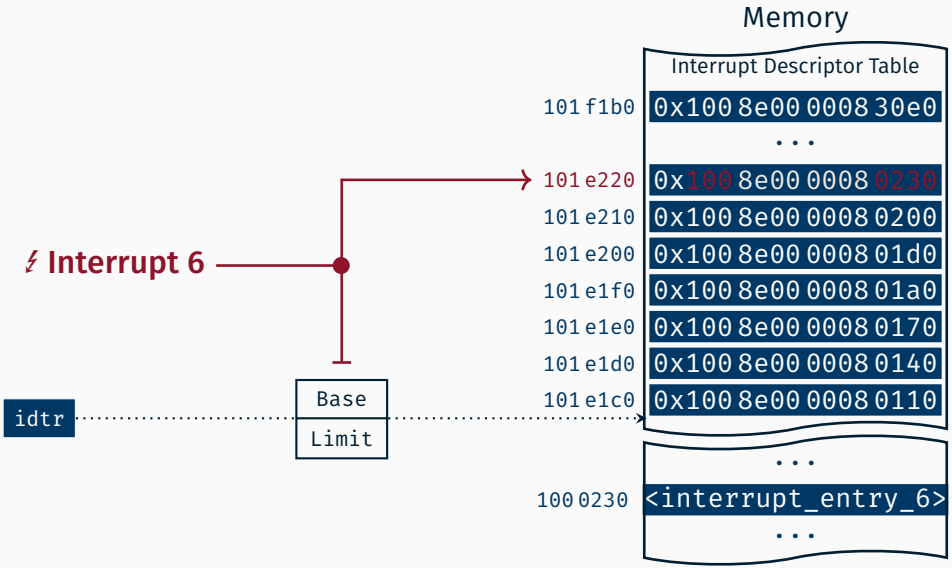
Interrupt Descriptor Table (IDT)



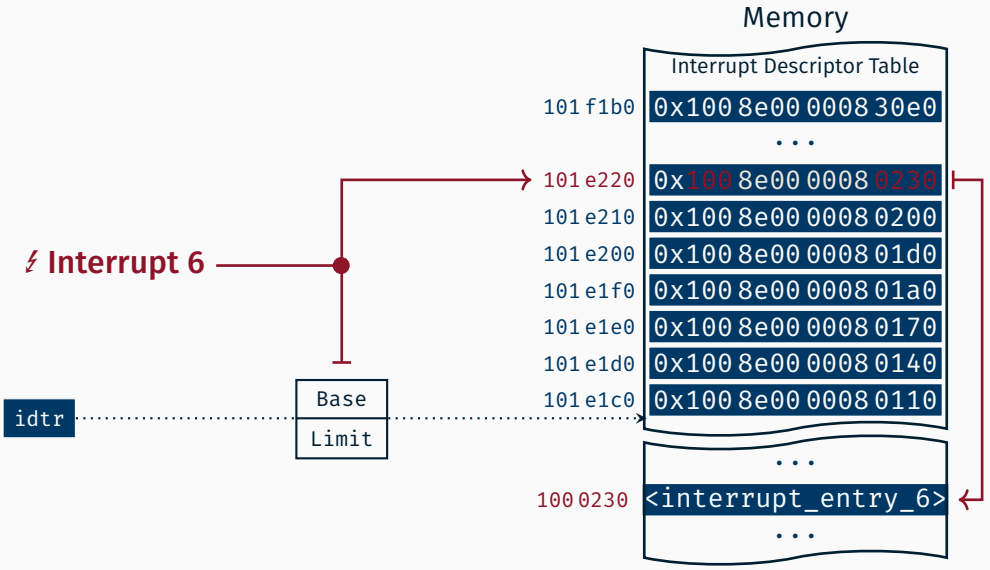
Interrupt Descriptor Table (IDT)



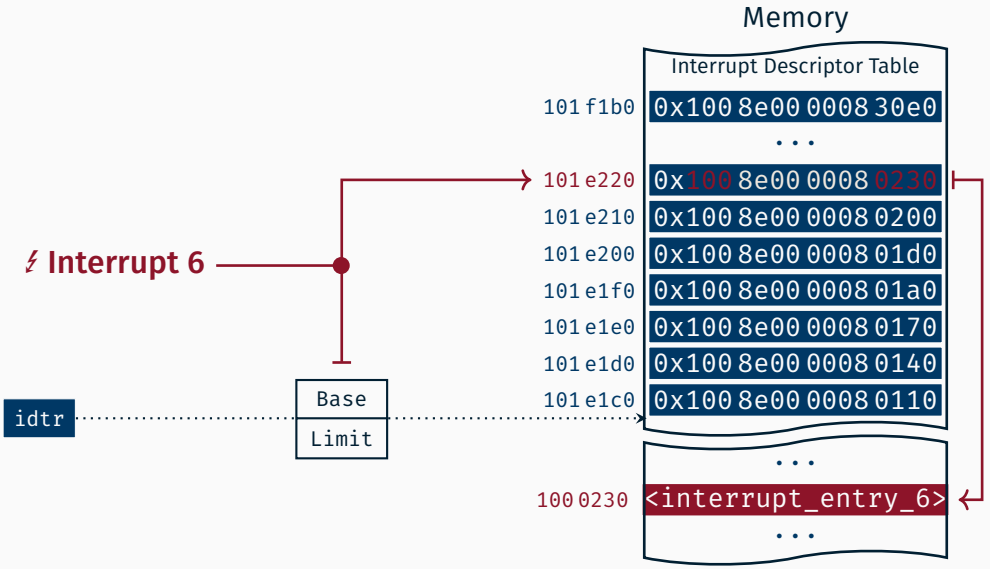
Interrupt Descriptor Table (IDT)



Interrupt Descriptor Table (IDT)

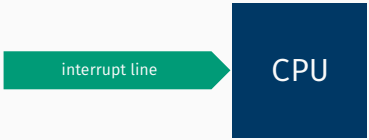


Interrupt Descriptor Table (IDT)

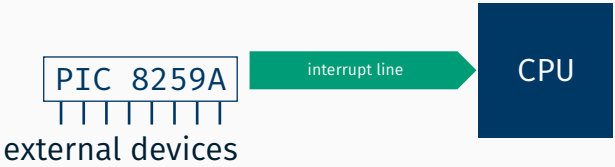


External interrupts

Interrupts from external devices on x86 CPUs



Interrupts from external devices on x86 CPUs



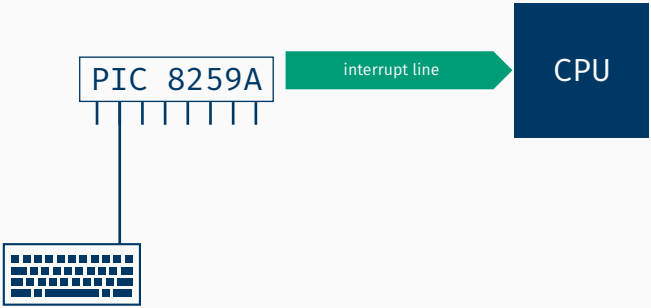
- Multiple devices can be attached through the Programmable Interrupt Controller

Interrupts from external devices on x86 CPUs



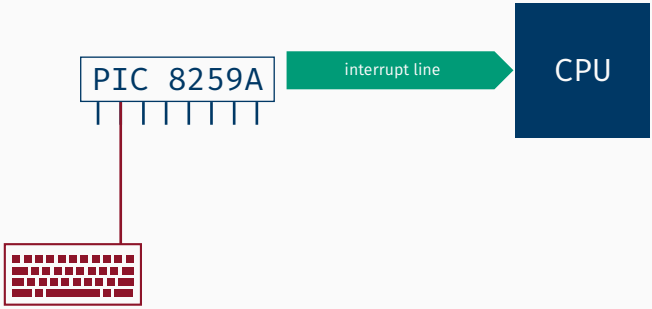
- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports

Interrupts from external devices on x86 CPUs



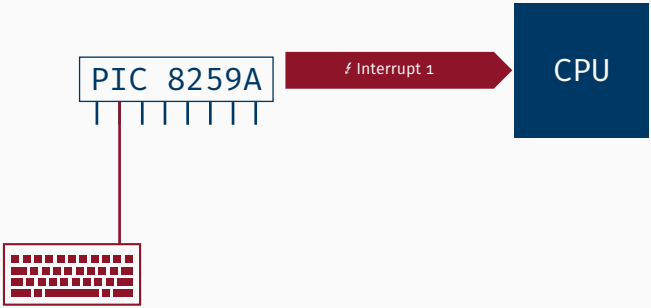
- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports

Interrupts from external devices on x86 CPUs



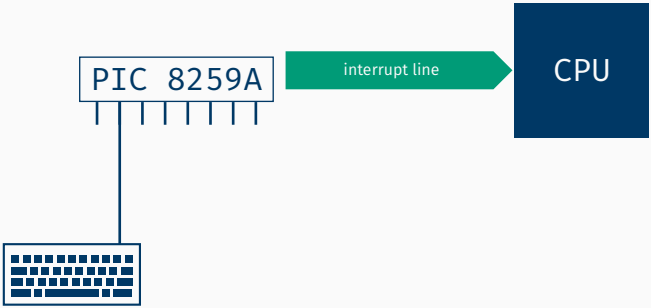
- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports

Interrupts from external devices on x86 CPUs



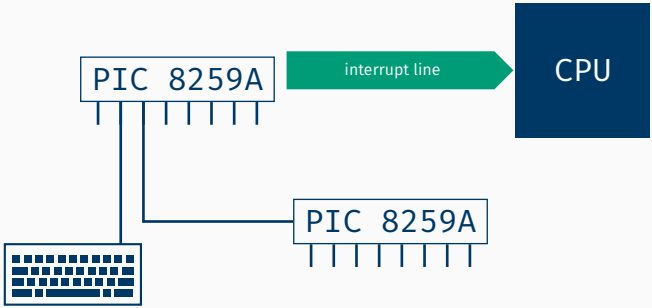
- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports

Interrupts from external devices on x86 CPUs



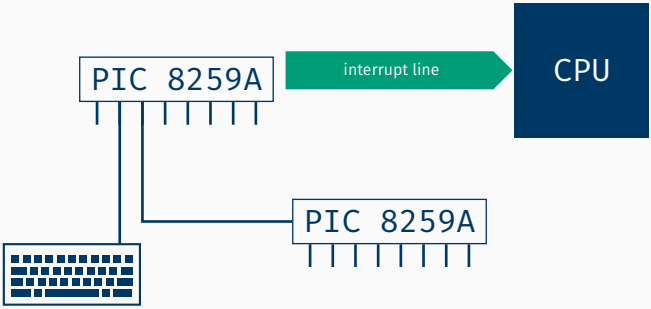
- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports

Interrupts from external devices on x86 CPUs



- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports
- Extension from 8 to 15 devices through dasy chaining

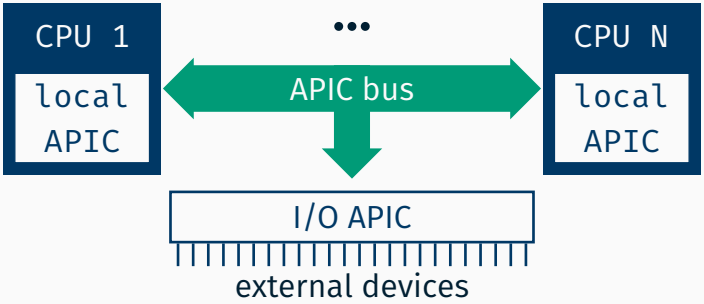
Interrupts from external devices on x86 CPUs



- Multiple devices can be attached through the Programmable Interrupt Controller
  - fixed priorities
  - interrupt vector numbers can be changed withen limits (multiple of 8)
  - configuration via I/O ports
- Extension from 8 to 15 devices through dasy chaining
- Cannot be used on multiprocessor systems

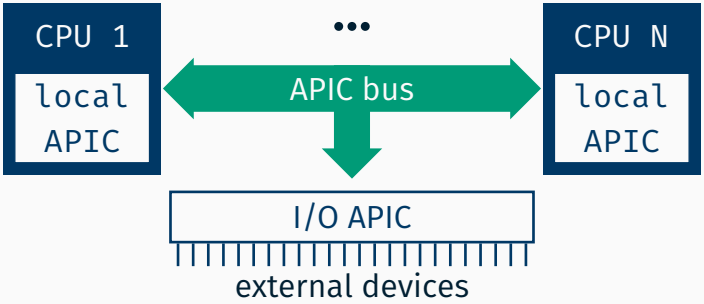
External interrupts with the APIC

The APIC Architecture



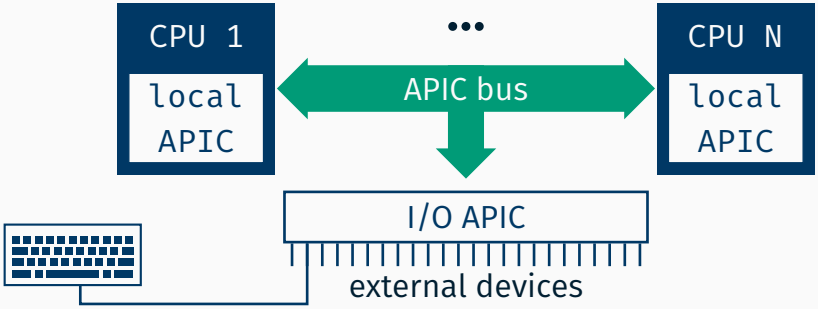
Separation into local APIC and I/O APIC

The APIC Architecture



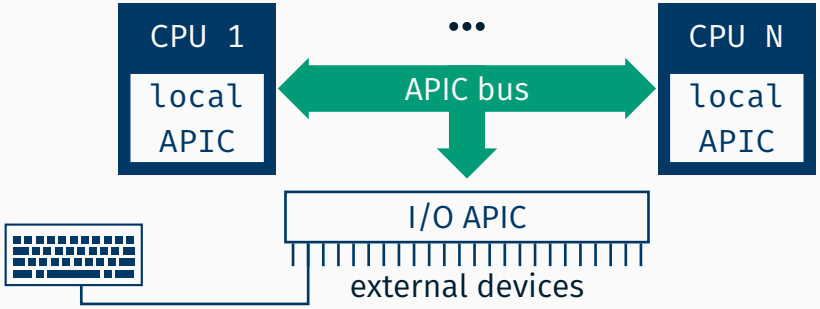
I/O APIC is used to attach the devices

The APIC Architecture



I/O APIC is used to attach the devices

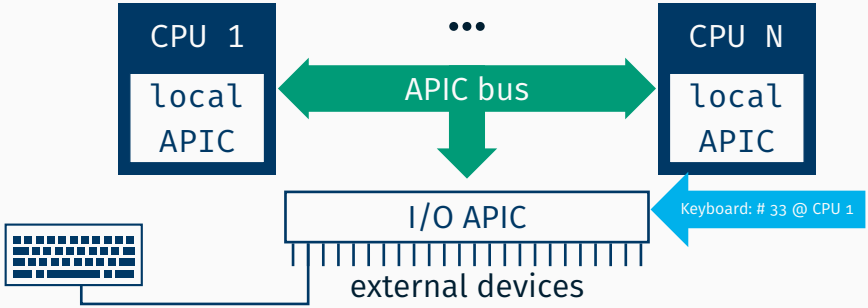
The APIC Architecture



I/O APIC is used to attach the devices

- Vector numbers can be assigned freely
- Interrupt sources can be activated and deactivated individually
- An interrupt can be assigned a destination processor in MP-Systems

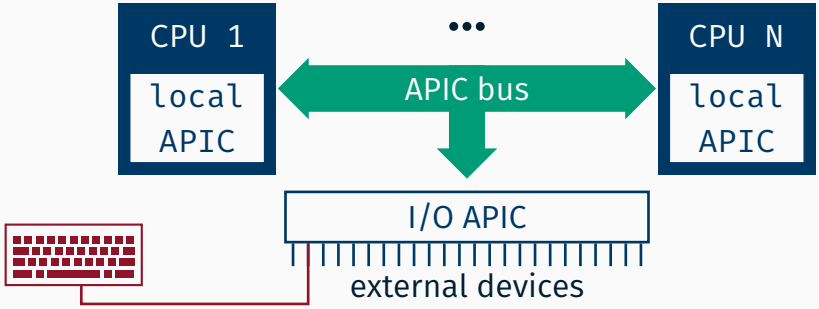
The APIC Architecture



I/O APIC is used to attach the devices

- Vector numbers can be assigned freely
- Interrupt sources can be activated and deactivated individually
- An interrupt can be assigned a destination processor in MP-Systems

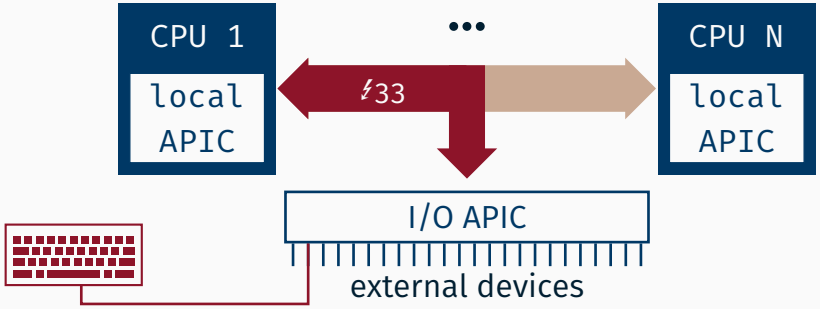
The APIC Architecture



I/O APIC is used to attach the devices

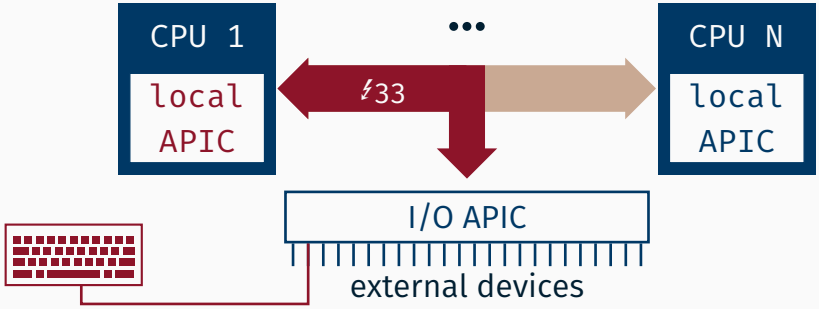
- Vector numbers can be assigned freely
- Interrupt sources can be activated and deactivated individually
- An interrupt can be assigned a destination processor in MP-Systems

The APIC Architecture



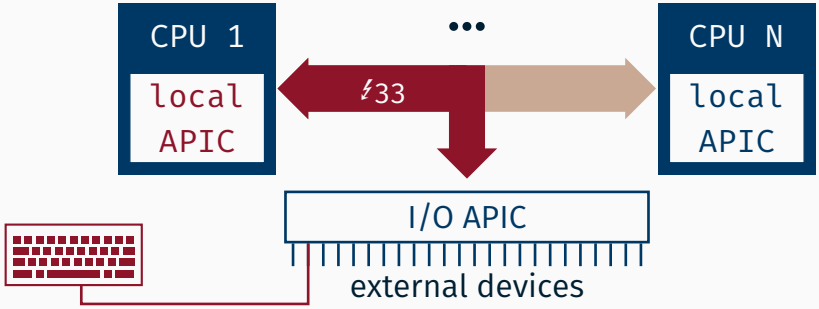
Interrupts become messages on the APIC bus

The APIC Architecture



Received by local APIC

The APIC Architecture



Received by local APIC

- Connects one CPU with the APIC bus
- Reads messages from the APIC bus and interrupts the CPU
- Must acknowledge interrupts (ACK)

Programming the Intel I/O APIC

Access to the internal registers via memory mapped I/O

Programming the Intel I/O APIC

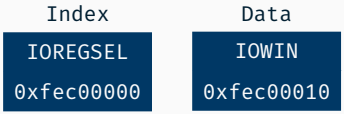
Access to the internal registers via memory mapped I/O

- However no direct mapping of internal registers to memory addresses

Programming the Intel I/O APIC

Access to the internal registers via memory mapped I/O

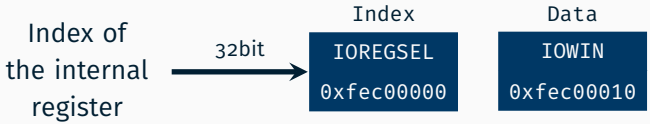
- However no direct mapping of internal registers to memory addresses
- *Diversion* through an index and data register



Programming the Intel I/O APIC

Access to the internal registers via memory mapped I/O

- However no direct mapping of internal registers to memory addresses
- *Diversion* through an index and data register



Programming the Intel I/O APIC

Access to the internal registers via memory mapped I/O

- However no direct mapping of internal registers to memory addresses
- *Diversion* through an index and data register



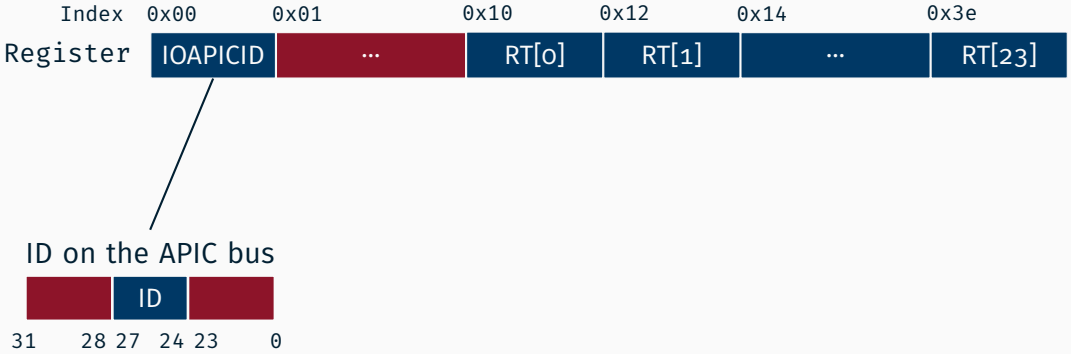
Programming the Intel I/O APIC (2)

Internal registers of the I/O APIC



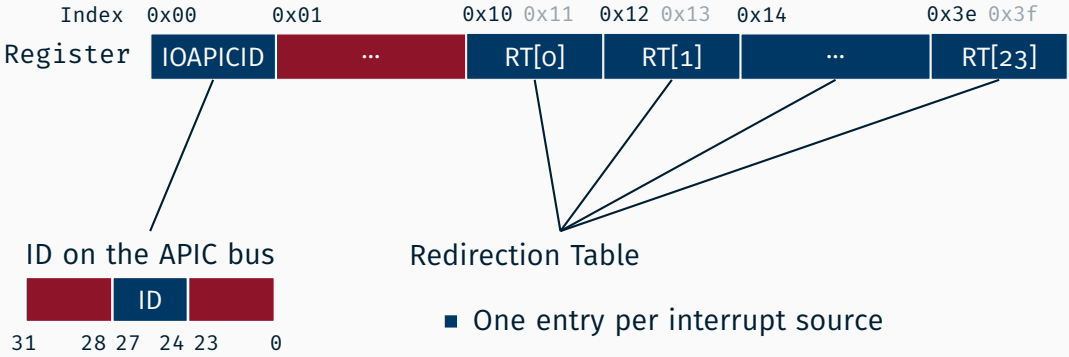
Programming the Intel I/O APIC (2)

Internal registers of the I/O APIC



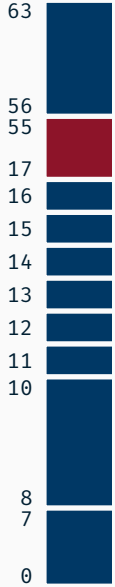
Programming the Intel I/O APIC (2)

Internal registers of the I/O APIC



- One entry per interrupt source
- Allows individual configuration of interrupt source
- Two internal registers per entry (64 bit)

Structure of a Redirection Table Entry



Destination Field: Destination address of the IRQ
- with `Dest. Mode == Physical` APIC ID of dst. CPU
- with `Dest. Mode == Logical` Group of dst. CPUs

reserved

Interrupt-Mask: Interrupt active (`0`) or inactive (`1`)
Trigger Mode: Edge (`0`) or level (`1`) trigger
Remote IRR: Type of the received acknowledgement
Interrupt Polarity: Active high (`0`) or active low (`1`)
Delivery Status: Interrupt message in flight?
Destination Mode: Physical (`0`) or logical (`1`) mode
Delivery Mode: e.g.
- `0` **Fixed** – Send signal to all destination processors
- `1` **Lowest Priority** – CPU with the lowest priority

Interrupt Vector: Number in the vector table (32 – 255)

Implementation in StuBS

Programming the APIC System

Programming the APIC System

Which device is attached where?

Programming the APIC System

Which device is attached where?

- This can differ from computer to computer!

Programming the APIC System

Which device is attached where?

- This can differ from computer to computer!
- Can be looked up in the system configuration, usually **ACPI** (Advanced Configuration and Power Interface)

Programming the APIC System

Which device is attached where?

- This can differ from computer to computer!
- Can be looked up in the system configuration, usually **ACPI** (Advanced Configuration and Power Interface)
- In our case `APIC` provides the relevant information

Programming the APIC System

Which device is attached where?

- This can differ from computer to computer!
- Can be looked up in the system configuration, usually **ACPI** (Advanced Configuration and Power Interface)
- In our case `APIC` provides the relevant information

  `APIC::getIOAPICSlot` reurns the index of the given device in the redirection table (see `enum Device` in `machine/apic.h`)

Programming the APIC System

Which device is attached where?

- This can differ from computer to computer!
- Can be looked up in the system configuration, usually **ACPI** (Advanced Configuration and Power Interface)
- In our case `APIC` provides the relevant information

  `APIC::getIOAPICSlot` reurns the index of the given device in the redirection table (see `enum Device` in `machine/apic.h`)

  `APIC::getIOAPICID` returns the ID of the I/O APICs

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs
- **Goal:** uniform distribution of the interrupts across all CPUs

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs
- **Goal:** uniform distribution of the interrupts across all CPUs
  - Set the priority of the processors in the local APIC to `0`

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs
- **Goal:** uniform distribution of the interrupts across all CPUs
  - Set the priority of the processors in the local APIC to `0`
  - In the I/O APIC use `Lowest Priority` Delivery Mode

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs
- **Goal:** uniform distribution of the interrupts across all CPUs
  - Set the priority of the processors in the local APIC to `0`
  - In the I/O APIC use `Lowest Priority` Delivery Mode
  - Use the `Logical Destination Mode`; up to 8 CPUs addressable

Programming the APIC System

Addressing APIC Messages in StuBS

- Multiple relevant factors
  - `Destination Mode`, `Destination Field` and `Delivery Mode` in the I/O APIC
  - Processor priority in the local APICs of the CPUs
- **Goal:** uniform distribution of the interrupts across all CPUs
  - Set the priority of the processors in the local APIC to `0`
  - In the I/O APIC use `Lowest Priority` Delivery Mode
  - Use the `Logical Destination Mode`; up to 8 CPUs addressable
  - `Destination Field`: bit mask with set bit per active CPU

Redirection Table Entries in StuBS



Destination Field: Destination address of the IRQ
- with `Dest. Mode == Physical` APIC ID of dst. CPU
- with `Dest. Mode == Logical` Group of dst. CPUs

reserved

Interrupt-Mask: Interrupt active (`0`) or inactive (`1`)
Trigger Mode: Edge (`0`) or level (`1`) trigger
Remote IRR: Type of the received acknowledgement
Interrupt Polarity: Active high (`0`) or active low (`1`)
Delivery Status: Interrupt message in flight?
Destination Mode: Physical (`0`) or logical (`1`) mode
Delivery Mode: e.g.
- `0` **Fixed** – Send signal to all destination processors
- `1` **Lowest Priority** – CPU with the lowest priority

Interrupt Vector: Number in the vector table (32 – 255)

(RO: Read Only)

Putting it all together: Keyboard Interrupt in StuBS

Preparation

Preparation

- Intitalize **I/O APIC**

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**
  - Determine keyboard slot, locate the corresponding entry in the **Redirection Table** and activate it
  - Clear keyboard buffer

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**
  - Determine keyboard slot, locate the corresponding entry in the **Redirection Table** and activate it
  - Clear keyboard buffer
- Create **interrupt handler**
  - Write trampoline `interrupt_entry` with call to `interrupt_handler` [already done in the handout]

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**
  - Determine keyboard slot, locate the corresponding entry in the **Redirection Table** and activate it
  - Clear keyboard buffer
- Create **interrupt handler**
  - Write trampoline `interrupt_entry` with call to `interrupt_handler` [already done in the handout]
  - Register in the **Interrupt Deskriptor Table** (**IDT**) and load it into the register `idtr` [already done in the handout]

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**
  - Determine keyboard slot, locate the corresponding entry in the **Redirection Table** and activate it
  - Clear keyboard buffer
- Create **interrupt handler**
  - Write trampoline `interrupt_entry` with call to `interrupt_handler` [already done in the handout]
  - Register in the **Interrupt Deskriptor Table** (**IDT**) and load it into the register `idtr` [already done in the handout]
  - Event handling in `interrupt_handler` via **Plugbox**

Preparation

- Intitalize **I/O APIC**
  - Set **I/O APIC ID**
  - Intitialize entires in the **Redirection Table** (disable)
- Configure **Keyboard**
  - Register in the **Plugbox**
  - Determine keyboard slot, locate the corresponding entry in the **Redirection Table** and activate it
  - Clear keyboard buffer
- Create **interrupt handler**
  - Write trampoline `interrupt_entry` with call to `interrupt_handler` [already done in the handout]
  - Register in the **Interrupt Deskriptor Table** (**IDT**) and load it into the register `idtr` [already done in the handout]
  - Event handling in `interrupt_handler` via **Plugbox**
- Enable interrupts with `Core::Interrupt::enable()`

Event Flow

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus**

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU
4. **CPU** executes interrupt handler

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU
4. **CPU** executes interrupt handler
   4.1 Select the corresponding entry in the **Interrupt Deskriptor Table** via register `idtr` and jump to the trampoline

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU
4. **CPU** executes interrupt handler
   4.1 Select the corresponding entry in the **Interrupt Deskriptor Table** via register `idtr` and jump to the trampoline
   4.2 Trampoline `interrupt_entry_33` stores registers and calls `interrupt_handler` with parameter `vector = 33`

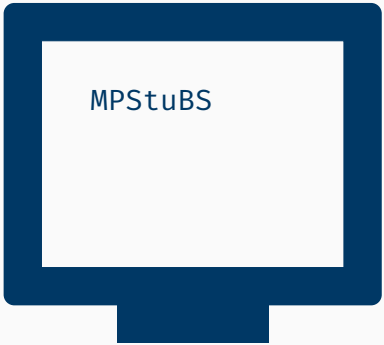
Event Flow

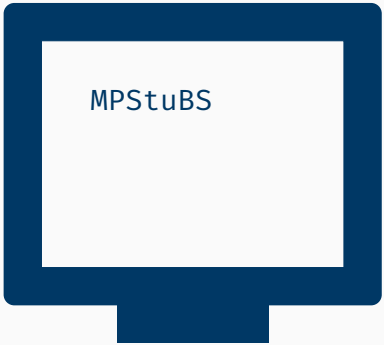
1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   - 2.1 Based on the **Redirection Table** an action is selected
   - 2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU
4. **CPU** executes interrupt handler
   - 4.1 Select the corresponding entry in the **Interrupt Deskriptor Table** via register `idtr` and jump to the trampoline
   - 4.2 Trampoline `interrupt_entry_33` stores registers and calls `interrupt_handler` with parameter `vector = 33`
   - 4.3 `interrupt_handler` uses **Plugbox** to handle the interrupt

Event Flow

1. **Key press** – keyboard processor (inside the keyboard) transmits this serially to the **PS/2 controller**
2. **PS/2 controller** activates interrupt line to **I/O APIC**
   2.1 Based on the **Redirection Table** an action is selected
   2.2 Message on **APIC bus** with inerrupt no. 33 and dest.-CPU
3. corresponding **LAPIC** receives Message from **APIC bus** and interrupts CPU
4. **CPU** executes interrupt handler
   4.1 Select the corresponding entry in the **Interrupt Deskriptor Table** via register `idtr` and jump to the trampoline
   4.2 Trampoline `interrupt_entry_33` stores registers and calls `interrupt_handler` with parameter `vector = 33`
   4.3 `interrupt_handler` uses **Plugbox** to handle the interrupt
5. **LAPIC** acknowledges the interrupt

Remote debugging with GDB


MPStuBS


MPStuBS

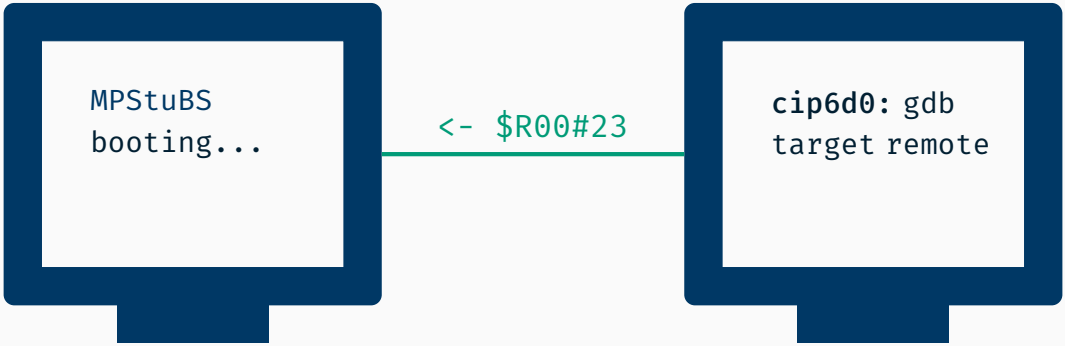

cip6d0:

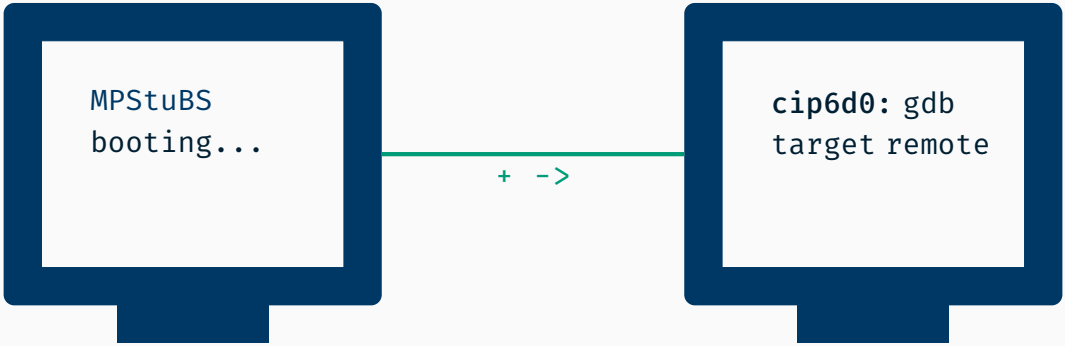

MPStuBS
cip6d0:


MPStuBS
cip6d0: gdb


MPStuBS
booting...
cip6d0: gdb

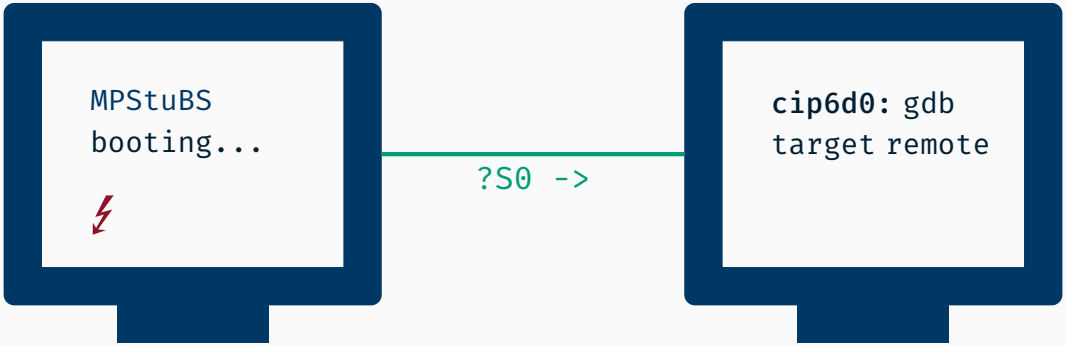

MPStuBS
booting...
cip6d0: gdb
target remote


MPStuBS
booting...
<- $R00#23
cip6d0: gdb
target remote


MPStuBS
booting...
+ ->
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 opcod
(gdb)



- Protocol is already implemented



- Protocol is already implemented
- uses custom interrupt handling for traps



- Protocol is already implemented
- uses custom interrupt handling for traps

→ modifies the IDT



- Protocol is already implemented
- uses custom interrupt handling for traps

→ modifies the IDT

- only the serial interface (implemented in task 1) is needed



- Protocol is already implemented
- uses custom interrupt handling for traps

→ modifies the IDT

- only the serial interface (implemented in task 1) is needed
- optional

Task 2

Goals

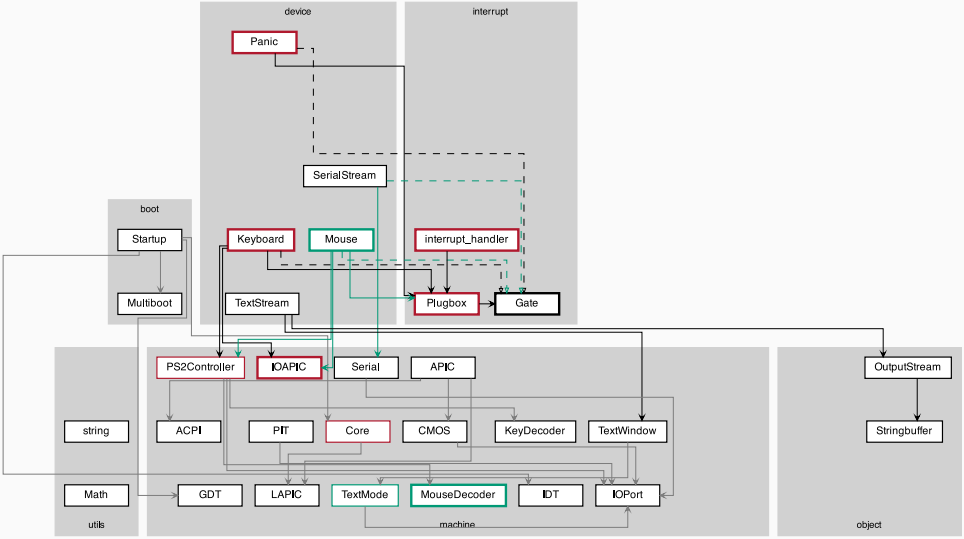
- Handling of asynchronous events
- Protection of critical sections

Goals

- Handling of asynchronous events
- Protection of critical sections

Task

- Configuration of external devices via I/O APIC
- Driver for **keyboard** and (*optional*) **mouse**
- Mutual exclusion (**Ticket-**/**Spinlock**)
- *Optional:* **GDB Stub**


device
interrupt
Panic
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

- **Problem:** Keyboard interrupts in KVM only on core 1



KVM might use Vector Hashing

- **Problem:** Keyboard interrupts in KVM only on core 1



KVM might use Vector Hashing

- **Problem:** Keyboard interrupts in KVM only on core `1`
- **Solution:** edit `/etc/modprobe.d/kvm_options.conf`, add entry **`options kvm vector_hashing=N`** and reboot the system.



KVM might use Vector Hashing

- **Problem:** Keyboard interrupts in KVM only on core `1`
- **Solution:** edit `/etc/modprobe.d/kvm_options.conf`, add entry **`options kvm vector_hashing=N`** and reboot the system.

(further details in the FAQ on the website)

Questions?

Submission of task 2 until Monday, 16th November
Next week is the GDB Seminar (10th November)