

Betriebssysteme

SS 2015

Hans-Georg Eßer

Dipl.-Math., Dipl.-Inform.

Foliensatz V:

- Ulix: Interrupts und Faults
- Ulix: System Calls

v1.0, 2015/05/11
(mit Literate Programming)

Übersicht: BS Praxis und BS Theorie

A	Einführung	Software-Verwaltung	E
	Shell	Scheduler / Prioritäten	F
	Dateiverwaltung	Synchronisation	G
B	Filter	Speicherverwaltung	S
	C-Compiler	Dateisysteme, Zugriffsrechte	T
C	Prozesse / Jobs	Einführung Ulix	U
	Threads	Ulix: Interrupts, Faults	
D	Interrupts		
	System Calls	Zusammenfassung	H

Folien V

Ulix: Interrupts und Faults

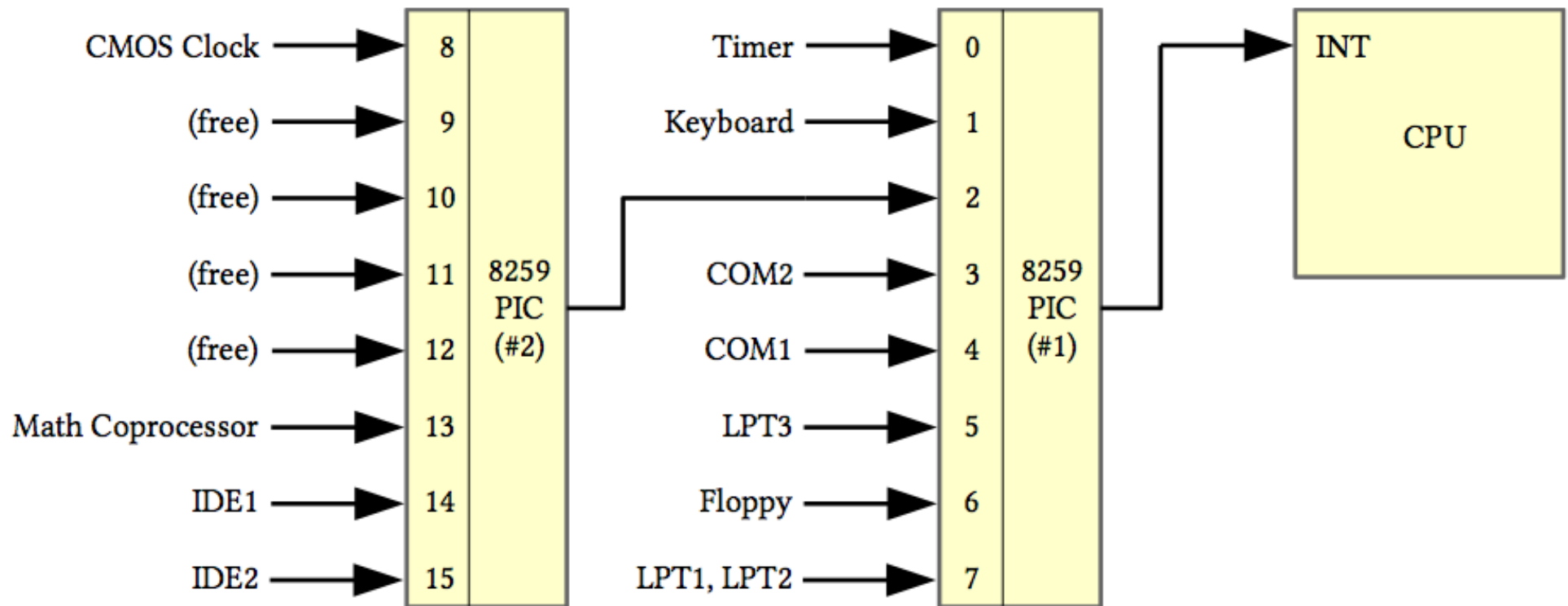
- Timer → u. a. Aufruf des Schedulers, Prozess- bzw. Thread-Wechsel
- Tastatur
- Festplatten- und Floppy-Controller
- serielle Schnittstelle

- offensichtliche Programmfehler
(z. B. Division durch 0, illegale Instruktion)
- falscher Speicherzugriff (Page Fault)
 - Seite ist gerade ausgelagert → wieder einlagern
 - Zugriff auf Adresse nicht erlaubt (Programm im User Mode, Zugriff auf Kernel-Speicher)
→ Programm abbrechen
 - Adresse existiert nicht → auch: abbrechen

Interrupts

Kaskadierte PICs

- Zwei PICs (Programmable Interrupt Controller)
- kaskadiert



```

⟨constants 6⟩≡
#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_SLAVE      2    // Here the slave PIC connects to master
#define IRQ_COM2        3
#define IRQ_COM1        4
#define IRQ_FDC         6
#define IRQ_IDE        14    // primary IDE controller; secondary has IRQ 15

```

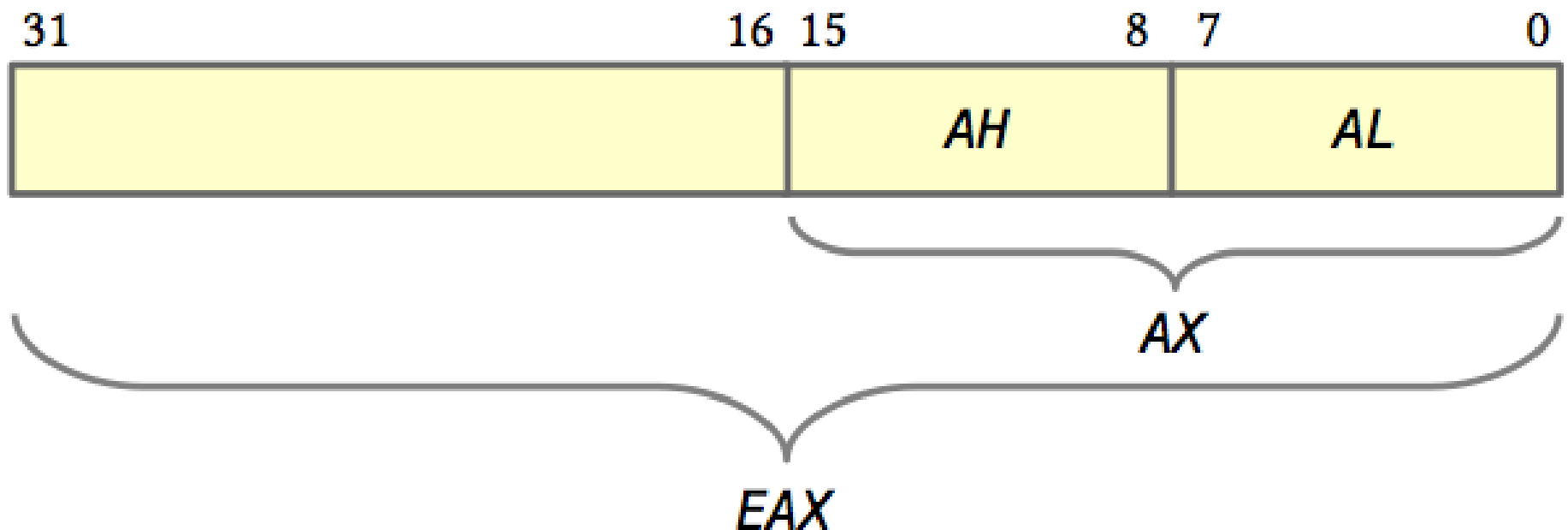
8▷

später: Interrupt-Handler für diese Interrupt-Nummern einrichten

- PICs initialisieren
 - über Kaskadierung informieren
 - Mapping der Interrupt-Nummern (jeweils 0–7) auf 32–39 bzw. 40–47 einrichten
- Für Zugriff auf PICs: in- und out-Befehle
 - Lesen und Schreiben von Ports
 - `inportb`, `outportb`: byte-weise (8 bit)
`inportw`, `outportw`: wort-weise (16 bit)
 - Funktionen nutzen Assembler-Befehle `inb`, `outb`, `inw`, `outw` (und diese nutzen CPU-Register)

32-Bit-Register (EAX, EBX etc.)

- Zur Erinnerung: Intel-Register-Struktur
EAX (32 bit), enthält *AX* (16 bit), *AL* (8 bit)



function implementations 7b ≡

12c▷

```

byte inportb (word port) {
    byte retval; asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
    return retval;
}

word inportw (word port) {
    word retval; asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
    return retval;
}

void outportb (word port, byte data) {
    asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
}

void outportw (word port, word data) {
    asm volatile ("outw %%ax, %%dx" : : "d" (port), "a" (data));
}

```

- Jeder der beiden PICs hat ein **Command Register** und ein **Control Register**, beschreibbar über Ports

<constants 6>+≡

<6 19a>

```
// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC_MASTER_CMD    0x20  // Master (IRQs 0-7), command register
#define IO_PIC_MASTER_DATA  0x21  // Master, control register

#define IO_PIC_SLAVE_CMD    0xA0  // Slave (IRQs 8-15), command register
#define IO_PIC_SLAVE_DATA   0xA1  // Slave, control register
```

Interrupt-Nummern remappen

Bildquellen:
siehe
Foliensatz D

CPU



PIC 1



PIC 2



erzeugt Faults (Nr. 0–31)

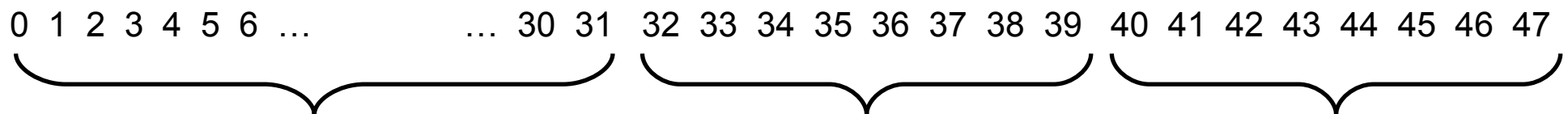
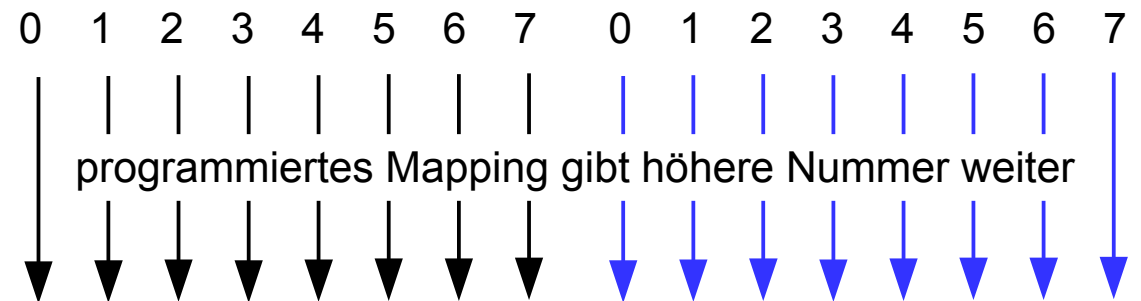
0 = Division by Zero

1 = Debug

2 = Non-maskable Interrupt

3 = Breakpoint

...



- Ziel: Remapping der Interrupt-Nummern
 - Master: 0..7 → 32..39
 - Slave: 0..7 → 40..47
- Dazu: Senden von vier Kontrollsequenzen ICW1 bis ICW4 (Initialization Command Words) an jeden der beiden PICs
- ICW1: Programmierung initialisieren

⟨remap the interrupts to 32..47 9a⟩ ≡ (13a) 9b▷

```
outportb (IO_PIC_MASTER_CMD, 0x11); // ICW1: initialize; begin programming
outportb (IO_PIC_SLAVE_CMD, 0x11);  // ICW1: dito, for PIC2
```

- ICW2: Remapping festlegen durch Angabe des Offset; einmal 32 (0x20), einmal 40 (0x28)

⟨remap the interrupts to 32..47 9a⟩ + ≡ (13a) ◁9a 9c▷

```

outportb (IO_PIC_MASTER_DATA, 0x20); // ICW2 for PIC1: offset 0x20
                                           // (remaps 0x00..0x07 -> 0x20..0x27)
outportb (IO_PIC_SLAVE_DATA, 0x28); // ICW2 for PIC2: offset 0x28
                                           // (remaps 0x08..0x0f -> 0x28..0x2f)

```

- ICW3: Slave-Konfiguration

[9c]

⟨remap the interrupts to 32..47 9a⟩ + ≡ (13a) ◁9b 9d▷

```

outportb (IO_PIC_MASTER_DATA, 0x04); // ICW3 for PIC1: there's a slave on IRQ 2
                                           // (0b00000100 = 0x04)
outportb (IO_PIC_SLAVE_DATA, 0x02); // ICW3 for PIC2: your slave ID is 2

```

- ICW4: 8086 mode

⟨remap the interrupts to 32..47 9a⟩ + ≡ (13a) < 9c

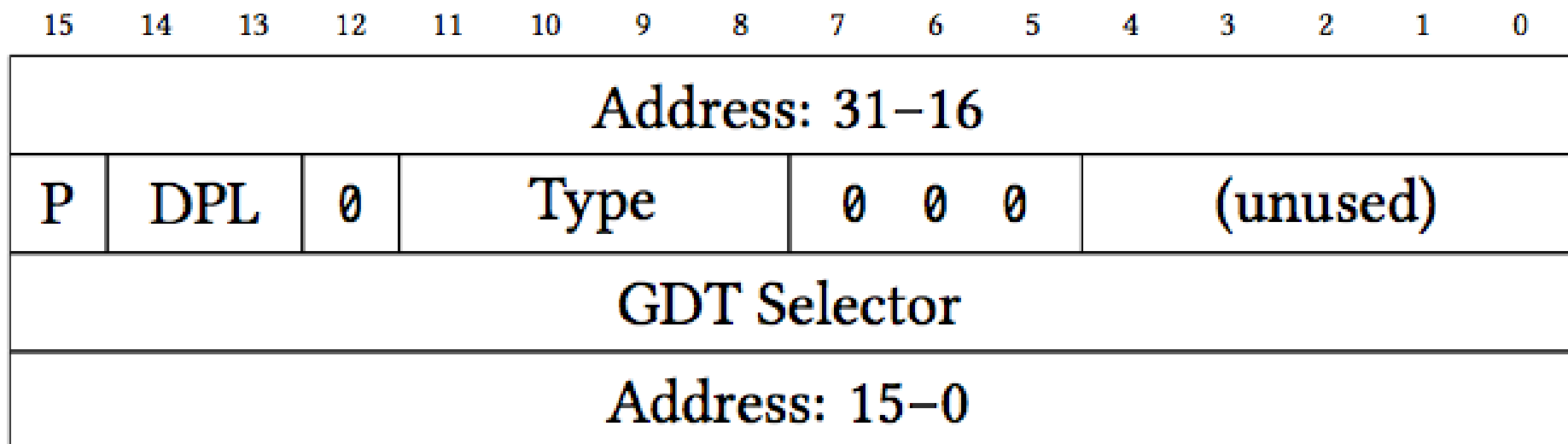
```
outportb (IO_PIC_MASTER_DATA, 0x01); // ICW4 for PIC1 and PIC2: 8086 mode
outportb (IO_PIC_SLAVE_DATA, 0x01);
```


- CPU muss Adressen der Interrupt-Handler kennen
- Intel-Architektur:
 - CPU-Register IDTR enthält Adresse eines **IDT Pointers**
 - IDT-Pointer speichert Adresse und Länge der **Interrupt Descriptor Table (IDT)**
 - IDT besteht aus mehreren **Interrupt Descriptors**

⟨type definitions 11a⟩≡

11b▷

```
struct idt_entry {
    unsigned int addr_low  : 16;  // lower 16 bits of address
    unsigned int gdt_sel   : 16;  // use which GDT entry?
    unsigned int zeroes    :  8;  // must be set to 0
    unsigned int type      :  4;  // type of descriptor
    unsigned int flags     :  4;
    unsigned int addr_high : 16;  // higher 16 bits of address
} __attribute__((packed));
```



7..6

5..4

3..2

1..0

$\langle \textit{type definitions 11a} \rangle + \equiv$

$\triangleleft 11a$

```
struct idt_ptr {
    unsigned int limit    : 16;
    unsigned int base     : 32;
} __attribute__((packed));
```

- Globale Variablen:
 - Die Tabelle selbst (`idt`)
 - der **IDT Pointer** auf die Tabelle (`idt_ptr`)
 - Adresse von `idt_ptr` später in IDTR schreiben, im Chunk *⟨install the interrupt descriptor table⟩*

⟨global variables 12a⟩≡

```
struct idt_entry idt[256] = { { 0 } };
struct idt_ptr idtp;
```

12e▷

⟨function implementations 7b⟩+≡

◁7b 13d▷

```
void fill_idt_entry (byte num, unsigned long address,  
    word gdt sel, byte flags, byte type) {  
    if (num ≥ 0 && num < 256) {  
        idt[num].addr_low  = address & 0xFFFF; // address is the handler address  
        idt[num].addr_high = (address >> 16) & 0xFFFF;  
        idt[num].gdt sel   = gdt sel;           // GDT sel.: user mode or kernel mode?  
        idt[num].zeroes    = 0;  
        idt[num].flags     = flags;  
        idt[num].type      = type;  
    }  
}
```

- Funktion schreibt einen IDT-Eintrag
- hier wichtig: Interrupt-Nummer und Handler-Adresse (zu gdt sel, flags, type: → später)

$\langle \text{function prototypes 7a} \rangle + \equiv$ $\langle 12b \ 13b \rangle$

```
extern void irq0(), irq1(), irq2(), irq3(), irq4(), irq5(), irq6(), irq7();
extern void irq8(), irq9(), irq10(), irq11(), irq12(), irq13(), irq14(), irq15();
```

- `irq0` bis `irq15` sind die Handler-Funktionen
→ Implementation in der Assembler-Datei

```

<global variables 12a>+≡
void (*irqs[16])() = {
    irq0, irq1, irq2, irq3, irq4, irq5, irq6, irq7,    // store them in
    irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15 // an array
};
    
```

- einfacheres „Ansprechen“ über ein Array

```
<install the interrupt handlers 13a> ≡  
<remap the interrupts to 32..47 9a>  
set_irqmask (0xFFFF);           // initialize IRQ mask  
enable_interrupt (IRQ_SLAVE);    // IRQ slave  
  
for (int i = 0; i < 16; i++) {  
    fill_idt_entry (32 + i,  
                    (unsigned int)irqs[i],  
                    0x08,  
                    0b1110,      // flags: 1 (present), 11 (DPL 3), 0  
                    0b1110);    // type: 1110 (32 bit interrupt gate)  
}
```

- trägt die 16 Handler `irq0` bis `irq15` in die IDT an Positionen 32–47 ein (Mapping!)
- Argument `gdt_sel=0x08`: Kernel Mode

⟨function implementations 7b⟩+≡

◁12c 13f▷

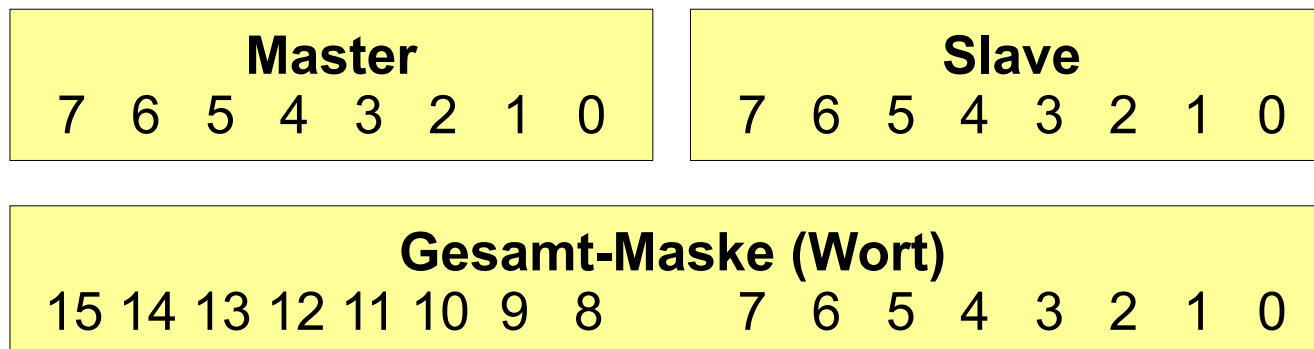
```
static void set_irqmask (word mask) {
    outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
    outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8) );
}
```

[13f]

⟨function implementations 7b⟩+≡

◁13d 14▷

```
word get_irqmask () {
    return inportb (IO_PIC_MASTER_DATA)
        + (inportb (IO_PIC_SLAVE_DATA) << 8);
}
```



- Idee: für jeden Interrupt-Handler, den wir einrichten, die Maske anpassen

<function implementations 7b>+≡

<13f 20a>

```
static void enable_interrupt (int number) {  
    set_irqmask (  
        get_irqmask ()           // the current value  
        & ~(1 << number)         // 16 one-bits, but bit "number" cleared  
    );  
}
```

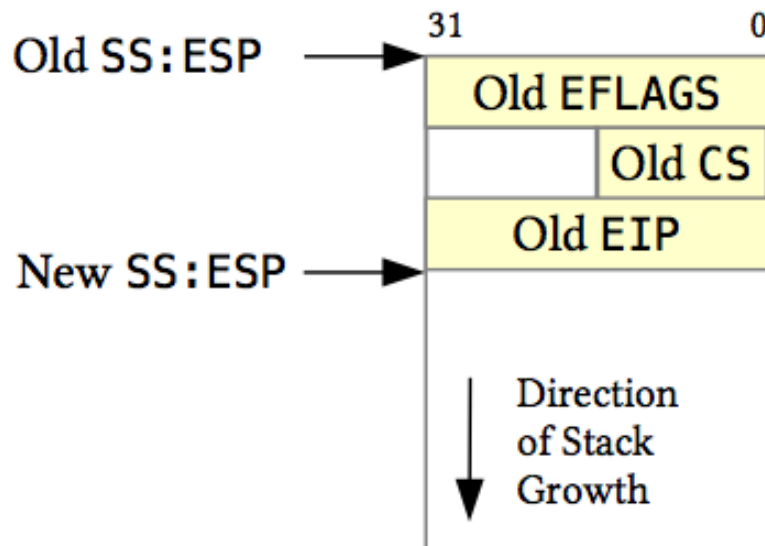
Stack-Einsatz beim Interrupt (1)

- Jeder Prozess besitzt zwei Stacks – für User Mode und Kernel Mode
 - Interrupt tritt im User Mode auf
→ Wechsel in Kernel Mode **und** Wechsel zum Kernel Mode Stack. Alte Stack-Adresse (und alten Modus) sichern!
 - Interrupt tritt im Kernel Mode auf
→ kein Wechsel, aktuellen Stack weiter verwenden
 - Diese Unterscheidung nimmt die CPU automatisch vor. Adresse des Kernel-Stacks steht in einer anderen Datenstruktur (**TSS**, ignorieren wir hier)

Stack-Einsatz beim Interrupt (2)

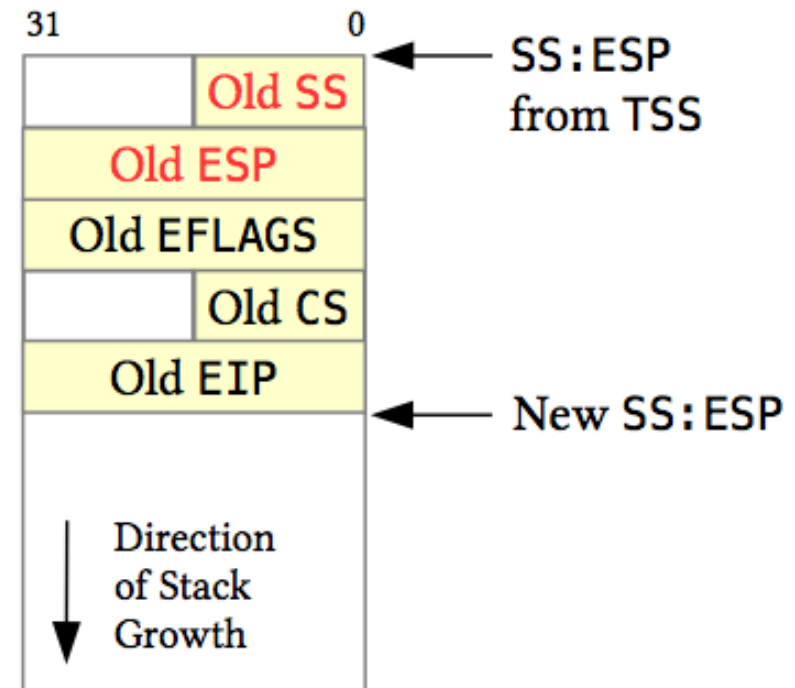
CPU im Kernel Mode (Ring 0)

→ keine Änderung des Privilege Levels, alten Stack weiter nutzen



CPU im User Mode (Ring 3)

→ Änderung des Privilege Levels auf Ring 0 (Kernel), neuen (Kernel-) Stack nutzen



- Alle Interrupt-Handler haben die Signatur
`void handler_function (context_t *r);`
- Dabei ist `context_t` eine Struktur, die alle wichtigen Register enthält
- Beim Interrupt sichert die CPU automatisch einige (wenige) Register auf den Stack; weitere müssen wir selbst (im Handler) sichern

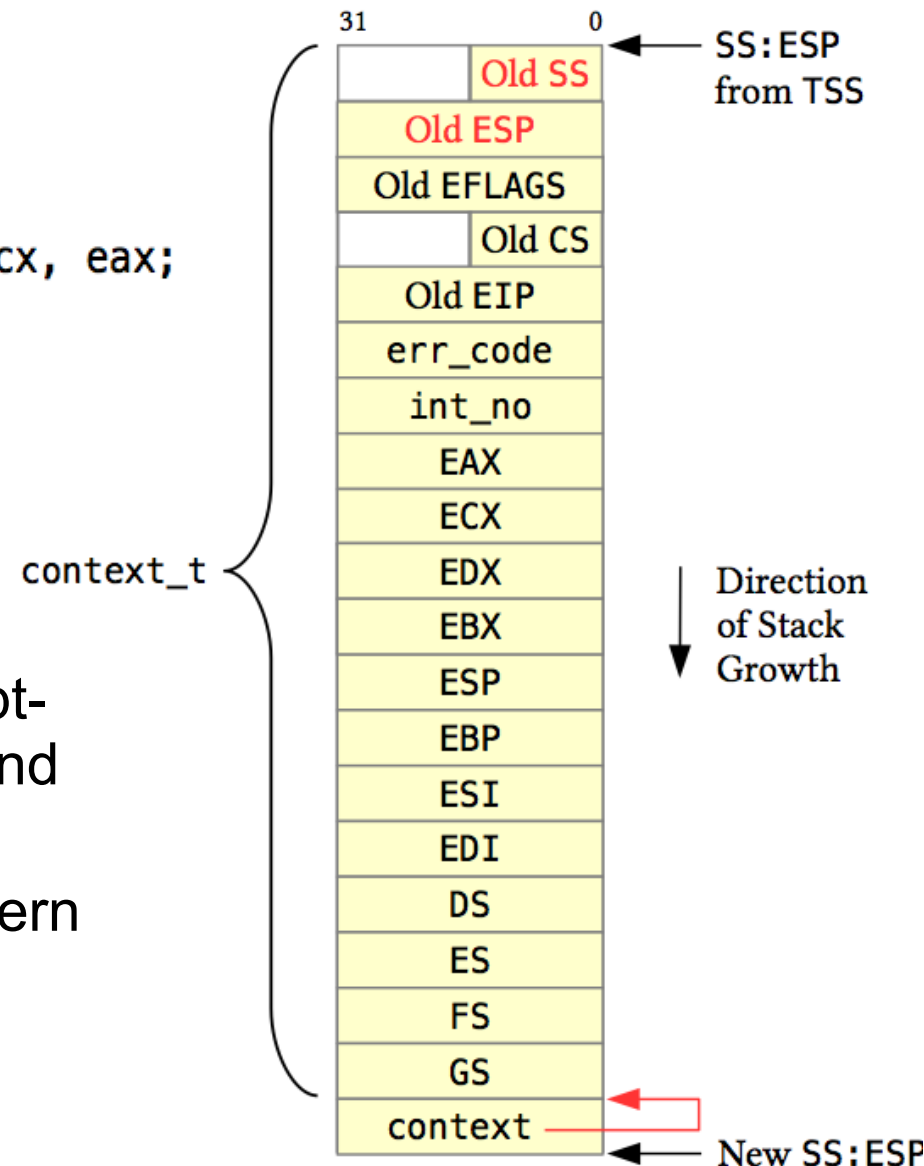
```

<public type definitions 16a>≡
typedef struct {
    unsigned int gs, fs, es, ds;
    unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
    unsigned int int_no, err_code;
    unsigned int eip, cs, eflags, useresp, ss;
} context_t;
    
```

Diesen Teil kopiert die CPU bei der Interrupt-Verarbeitung automatisch auf den Stack (und nimmt ihn bei `iret` wieder runter).

Um den Rest müssen wir uns selbst kümmern

→ *<push registers onto the stack>*,
<pop registers from the stack> sowie
 Interrupt-Nummer und Fehlercode



⟨push registers onto the stack 16b⟩≡

(17b 18 24b)

```
pusha
push ds
push es
push fs
push gs
push esp ; pointer to the context_t
```

(pusha / popa: *EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI*)

[17a]

⟨pop registers from the stack 17a⟩≡

(17b 18 24b)

```
pop esp
pop gs
pop fs
pop es
pop ds
popa
```

```

<irq15 example 17b>≡
  push byte 0           ; error code
  push byte 15          ; interrupt number
  <push registers onto the stack 16b>
  call irq_handler      ; call C function
  <pop registers from the stack 17a>
  add esp, 8            ; for errcode, irq no.
  iret

```

Diesen Code bräuchten wir jetzt 16 mal ...

⟨start.asm 18⟩≡

21a▷

```
global irq0, irq1, irq2, irq3, irq4, irq5, irq6, irq7
global irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15
```

```
%macro irq_macro 1
    push byte 0          ; error code (none)
    push byte %1         ; interrupt number
    jmp irq_common_stub ; rest is identical for all handlers
%endmacro
```

```
extern irq_handler      ; defined in the C source file
```

```
irq_common_stub:      ; this is the identical part
    ⟨push registers onto the stack 16b⟩
    call irq_handler   ; call C function
    ⟨pop registers from the stack 17a⟩
    add esp, 8
    iret
```

```
irq0:  irq_macro 32
irq1:  irq_macro 33
irq2:  irq_macro 34
irq3:  irq_macro 35
...
```

```

<constants 6>+≡                                     <8
    #define END_OF_INTERRUPT    0x20

<global variables 12a>+≡                             <12e 24c>
    void *interrupt_handlers[16] = { 0 };

<function implementations 7b>+≡                     <14 20c>
    void irq_handler (context_t *r) {
        int number = r->int_no - 32;                  // interrupt number
        void (*handler)(context_t *r);               // type of handler functions

        if (number ≥ 8) {
            outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT); // notify slave PIC
        }
        outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);  // notify master PIC (always)

        handler = interrupt_handlers[number];
        if (handler != NULL) handler (r);
    }

```

⟨function implementations 7b⟩+≡

◁20a 25a▷

```
void install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
    if (irq ≥ 0 && irq < 16)
        interrupt_handlers[irq] = handler;
}
```

Während Kernel-Initialisierung:

```
install_interrupt_handler (IRQ_IDE,    ide_handler);
install_interrupt_handler (IRQ_FDC,    floppy_handler);
install_interrupt_handler (IRQ_COM2,   serial_hard_disk_handler);
install_interrupt_handler (IRQ_KBD,    keyboard_handler);
install_interrupt_handler (IRQ_TIMER,  timer_handler);
```

```

<install the interrupt descriptor table 20d>≡
    idtp.limit = (sizeof (struct idt_entry) * 256) - 1;    // must do -1
    idtp.base  = (int) &idt;
    idt_load ();

```

[20e]

```

<function prototypes 7a>+≡
    extern void idt_load ();

```

<20b 21b>

[21a]

```

<start.asm 18>+≡
    extern idtp                ; defined in the C file
    global idt_load
    idt_load:    lidt [idt]
                ret

```

<18 22a>

Faults

Analog zu `irq0 ... irq15`:

32 Fault-Handler-Funktionen `fault0 ... fault31`
(und `fault128` für System Call Handler → später)

⟨function prototypes 7a⟩+≡

◁20e 24d▷

extern void

```
fault0(), fault1(), fault2(), fault3(), fault4(), fault5(), fault6(),  
fault7(), fault8(), fault9(), fault10(), fault11(), fault12(), fault13(),  
fault14(), fault15(), fault16(), fault17(), fault18(), fault19(), fault20(),  
fault21(), fault22(), fault23(), fault24(), fault25(), fault26(), fault27(),  
fault28(), fault29(), fault30(), fault31(), fault128();
```

(Diese Funktionen sind wieder in Assembler geschrieben.)

Auch die Fault-Handler landen in der IDT; diesmal beschleunigtes Eintragen via Makro:

```
<macro definitions 21c>≡
#define FILL_IDT(i) fill_idt_entry (i, (unsigned int)fault##i, 0x08, 0b1110, 0b1110)
```

[21d]

```
<install the fault handlers 21d>≡
FILL_IDT( 0); FILL_IDT( 1); FILL_IDT( 2); FILL_IDT( 3); FILL_IDT( 4); FILL_IDT( 5);
FILL_IDT( 6); FILL_IDT( 7); FILL_IDT( 8); FILL_IDT( 9); FILL_IDT(10); FILL_IDT(11);
FILL_IDT(12); FILL_IDT(13); FILL_IDT(14); FILL_IDT(15); FILL_IDT(16); FILL_IDT(17);
FILL_IDT(18); FILL_IDT(19); FILL_IDT(20); FILL_IDT(21); FILL_IDT(22); FILL_IDT(23);
FILL_IDT(24); FILL_IDT(25); FILL_IDT(26); FILL_IDT(27); FILL_IDT(28); FILL_IDT(29);
FILL_IDT(30); FILL_IDT(31); FILL_IDT(128);
```

```

<start.asm 18>+≡
global fault0, fault1, fault2, fault3, fault4, fault5, fault6, fault7
global fault8, fault9, fault10, fault11, fault12, fault13, fault14, fault15
global fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23
global fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31
    <21a 22b>

```

Zweierlei Faults: ohne / mit Fehlercode

Wenn die CPU beim Fault nicht selbst einen Fehlercode auf den Stack schreibt, dann schreiben wir eine 0

```

fault5:  push byte 0    ; error code
         push byte 5
         jmp  fault_common_stub

```

```

fault8:  ; no error code
         push byte 8
         jmp  fault_common_stub

```


<start.asm 18>+≡

```
%macro fault_macro_0 1
    push byte 0 ; error code
    push byte %1
    jmp fault_common_stub
%endmacro
```

<22a 22c>

```
%macro fault_macro_no0 1
    ; don't push error code
    push byte %1
    jmp fault_common_stub
%endmacro
```

⟨start.asm 18⟩+≡

◁22b 24a▷

```

fault0:  fault_macro_0      0      ; Divide by Zero
fault1:  fault_macro_0      1      ; Debug
fault2:  fault_macro_0      2      ; Non Maskable Interrupt
fault3:  fault_macro_0      3      ; INT 3
fault4:  fault_macro_0      4      ; INTO
fault5:  fault_macro_0      5      ; Out of Bounds
fault6:  fault_macro_0      6      ; Invalid Opcode
fault7:  fault_macro_0      7      ; Coprocessor not available
fault8:  fault_macro_no0    8      ; Double Fault
fault9:  fault_macro_0      9      ; Coprocessor Segment Overrun
fault10: fault_macro_no0    10     ; Bad TSS
fault11: fault_macro_no0    11     ; Segment Not Present
fault12: fault_macro_no0    12     ; Stack Fault
fault13: fault_macro_no0    13     ; General Protection Fault
fault14: fault_macro_no0    14     ; Page Fault
fault15: fault_macro_0      15     ; (reserved)
fault16: fault_macro_0      16     ; Floating Point
fault17: fault_macro_0      17     ; Alignment Check
fault18: fault_macro_0      18     ; Machine Check
fault19: fault_macro_0      19     ; (reserved)

```

(... und weiter bis fault31;
alle „reserved“)

```

<start.asm 18>+≡
extern fault_handler

```

◁22c 24b▷

fault_handler ist wieder eine C-Funktion ...

[24b]

```

<start.asm 18>+≡
fault_common_stub:
    <push registers onto the stack 16b>
    call fault_handler    ; call C function
    <pop registers from the stack 17a>
    add esp, 8           ; for errcode, irq no.
    iret

```

◁24a

fault_common_stub: vergleiche irq_common_stub
(identisch bis auf call fault_handler st ↔ att call
irq_handler)

<global variables 12a>+≡

<19b

```
char *exception_messages[] = {
    "Division By Zero",      "Debug",           // 0, 1
    "Non Maskable Interrupt", "Breakpoint",      // 2, 3
    "Into Detected Overflow", "Out of Bounds",   // 4, 5
    "Invalid Opcode",        "No Coprocessor",  // 6, 7
    "Double Fault",          "Coprocessor Segment Overrun", // 8, 9
    "Bad TSS",               "Segment Not Present", // 10, 11
    "Stack Fault",           "General Protection Fault", // 12, 13
    "Page Fault",            "Unknown Interrupt",  // 14, 15
    "Coprocessor Fault",     "Alignment Check",    // 16, 17
    "Machine Check",         // 18
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
    "Reserved", "Reserved", "Reserved" // 19..31
};
```

```

<function implementations 7b>+≡
void fault_handler (context_t *r) {
    if (r->int_no == 14) {
        page_fault_handler (r); return;
    }

    memaddress fault_address = (memaddress)(r->eip);

    if (r->int_no < 32) {
        <fault handler: display status information 26a>

        if ( fault_address < 0xc0000000 ) {
            <fault handler: terminate process 26b>

            <disable scheduler>
            <enable interrupts 25b>
            printf ("\n");
            asm ("jmp kernel_shell");
        }
    }
}
    <20c
    // fault 14 is a page fault
    // user mode
    // error inside the kernel

```

$\langle \text{enable interrupts 25b} \rangle \equiv$ (25a)
`asm ("sti"); // set interrupt flag`

$\langle \text{disable interrupts 25c} \rangle \equiv$
`asm ("cli"); // clear interrupt flag`

System Calls

Vorgehensweise:

- Anwendung trägt **System-Call-Nummer** in das Register *EAX* ein
- Parameter für den System Call: in weitere Register (*EBX*, *ECX*, *EDX*)
- Software-Interrupt auslösen → durch CPU-Instruktion `int 0x80`
- CPU reagiert auf `int`-Instruktion wie auf einen HW-Interrupt und ruft Interrupt-Handler auf; der ruft den richtigen System-Call-Handler auf

⟨*constants* 96a⟩+≡ (28a) ◁174a 216b▷

```
#define MAX_SYSCALLS 1024           // max syscall number: 1023
```

⟨*global variables* 76b⟩+≡ (28a) ◁178b 189b▷

```
void *syscall_table[MAX_SYSCALLS];
```

⟨*function implementations* 84b⟩+≡ (28a) ◁180 185c▷

```
void install_syscall_handler (int syscallno, void *syscall_handler) {
    if (syscallno ≥ 0 && syscallno < MAX_SYSCALLS)
        syscall_table[syscallno] = syscall_handler;
};
```

Beispiel für Eintragung in Tabelle

⟨*syscall entry example* 185b⟩≡

```
install_syscall_handler (__NR_write, sys_write);
```

⟨function implementations 84b⟩+≡ (28a) ◁185a 185e▷

```

void syscall_handler (context_t *r) {
    void (*handler) (context_t*); // handler is a function pointer
    int number = r->eax;
    if (number != __NR_get_errno) set_errno (0); // default: no error
    handler = syscall_table[number];
    if (handler != 0) handler (r);
    else
        printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
                "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
}

```

Assembler-Funktion sieht wie bei den anderen Interrupt-Handlern aus:

```

<start.asm 71>+≡
[section .text]
    extern syscall_handler
    global fault128

fault128: push byte 0          ; put 128 on the stack so it looks the same
          push byte -128       ; as it does after a hardware interrupt
          <push registers onto the stack 126b>
          call syscall_handler
          <pop registers from the stack 127a>
          add esp, 8           ; undo the two "push byte" commands from the start_
          iret
    
```

◁182 196c▷

- Bibliotheksfunktion `syscall4()` kopiert Argumente `eax`, `ebx`, `ecx`, `edx` in die gleichnamigen CPU-Register,
- führt dann `int 0x80` aus
- und gibt den Inhalt des Registers *EAX* zurück

<ulib function implementations 158d>+≡ (32b) <158d 187a>

```
inline int syscall4 (int eax, int ebx, int ecx, int edx) {
    int result;
    asm ( "int $0x80" : "=a" (result) : "a" (eax), "b" (ebx), "c" (ecx), "d" (edx) );
    return result;
}
```

- analog: `syscall3`, `syscall2`, `syscall1`

Beispiel: read()

- System Call installieren (Kernel-Initialisierung)

```
install_syscall_handler (__NR_read,      syscall_read);
```

- System Call Handler für read (im Kernel)

```
#define eax_return(retval) { r->eax = (unsigned int)((retval)); return; }

void syscall_read (context_t *r) {
    // ebx: fd, ecx: *buf, edx: nbytes
    eax_return ( u_read (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) ); };
```

- Bibliotheksfunktion read (User Mode)

```
int read (int fd, void *buf, size_t nbytes) {
    return syscall4 (__NR_read, fd, (uint)buf, nbytes); }
```