BS-Entwicklung mit Literate Programming

Foliensatz 7: Interrupts und Faults

Hans-Georg Eßer
TH Nürnberg

v1.1, 11.11.2014

Überblick

- Interrupts
  - Geräte erzeugen Interrupts (asynchron)
  - sind mit einem von zwei PICs verbunden
  - PICs leiten an CPU weiter
  - Interrupt-Handler: BS wird unterbrochen und reagiert
- Faults
  - ähnlich wie Interrupt-Handling
  - aber synchron: durch Code verursacht
  - z. B. Division durch 0, Page Fault
  - Fault-Handler: muss ggf. Prozess abbrechen

Chunk: ⟨constants⟩ (1)

- Wir definieren IRQ-Konstanten für die "interessanten" Interrupts
- Timer, Tastatur, seriell, Floppy, IDE

⟨constants⟩≡
```
  #define IRQ_TIMER       0
  #define IRQ_KBD         1
  #define IRQ_SLAVE       2      // Here the slave P↩
…IC connects to master
  #define IRQ_COM2        3
  #define IRQ_COM1        4
  #define IRQ_FDC         6
  #define IRQ_IDE        14      // primary IDE cont↩
…roller; secondary has IRQ 15
```

Chunk: ⟨function prototypes⟩ (1)

- Funktionen für Kommunikation mit Geräten
- Port-Nummern
- Bytes oder Wörter übertragen

⟨function prototypes⟩≡
```
  unsigned char  inportb (unsigned short port);
  unsigned short inportw (unsigned short port);
  void outportb (unsigned short port, unsigned cha↩
…r data);
  void outportw (unsigned short port, unsigned sho↩
…rt data);
```

Chunk: ⟨function implementations⟩ (1)

⟨function implementations⟩≡
```
  unsigned short inportw (unsigned short port) {
    unsigned short retval;
    asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
    return retval;
  }

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

Chunk: ⟨constants⟩ (2)

- Auch die PICs (programmable interrupt controllers) haben I/O-Ports
- Master und Slave
- je acht Eingänge, ein Ausgang
- Ausgang vom Slave geht in Eingang 2 vom Master
- Ausgang vom Master geht zur CPU

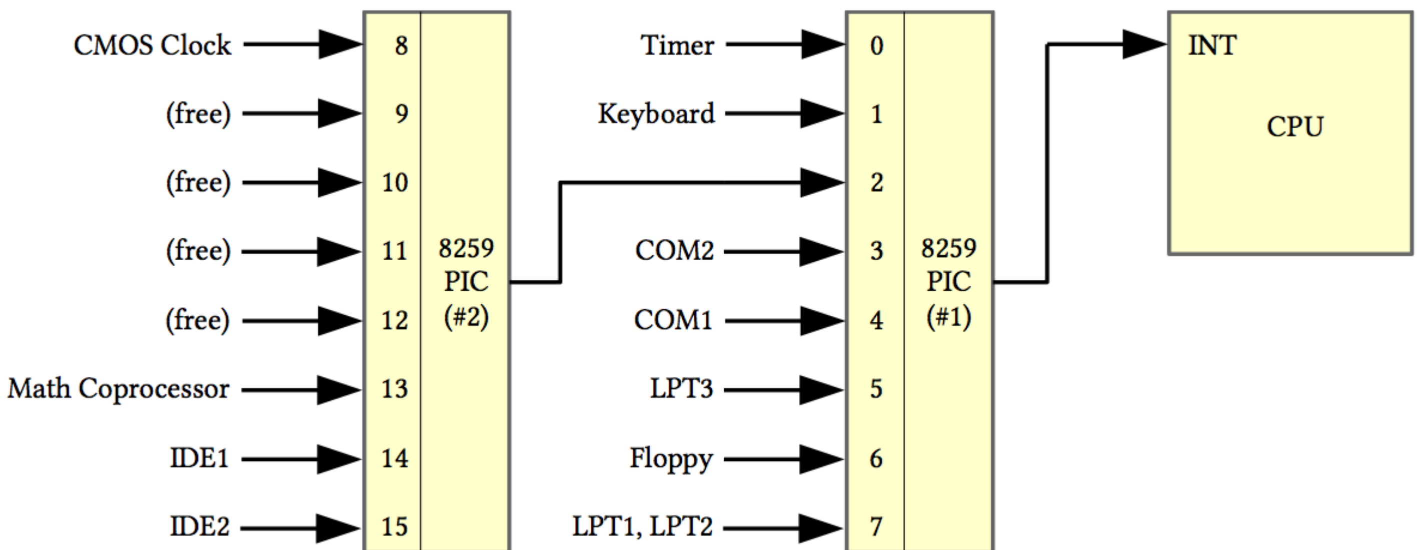
⟨*constants*⟩+≡

```
  // I/O Addresses of the two programmable interru↩
…pt controllers
  #define IO_PIC_MASTER_CMD    0x20  // Master (IRQ↩
…s 0-7), command register
  #define IO_PIC_MASTER_DATA   0x21  // Master, con↩
…trol register

  #define IO_PIC_SLAVE_CMD     0xA0  // Slave (IRQs↩
… 8-15), command register
  #define IO_PIC_SLAVE_DATA    0xA1  // Slave, cont↩
…rol register
```

PIC-Kaskade

Kaskadierung der beiden PICs



Chunk: ⟨remap the interrupts to 32..47⟩ (1)

- Wir müssen die Einstellungen der PICs ändern
- Interrupt-Nummern umbiegen
- Master: 0..7 auf 32..39
- Slave: 0..7 auf 40..47

⟨*remap the interrupts to 32..47*⟩≡
  ⟨*PIC: program/initialize the PICs*⟩
  ⟨*PIC: set the initial interrupt mask*⟩

Chunk: ⟨PIC: program/initialize the PICs⟩ (1)

⟨*PIC: program/initialize the PICs*⟩≡

```
  outportb (IO_PIC_MASTER_CMD,  0x11);  // ICW1: initialize; begin programming
  outportb (IO_PIC_SLAVE_CMD,   0x11);  // ICW1: dito, for PIC2
  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)
  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
  outportb (IO_PIC_MASTER_DATA, 0x01);  // ICW4 for PIC1 and PIC2: 8086 mode
  outportb (IO_PIC_SLAVE_DATA,  0x01);
```

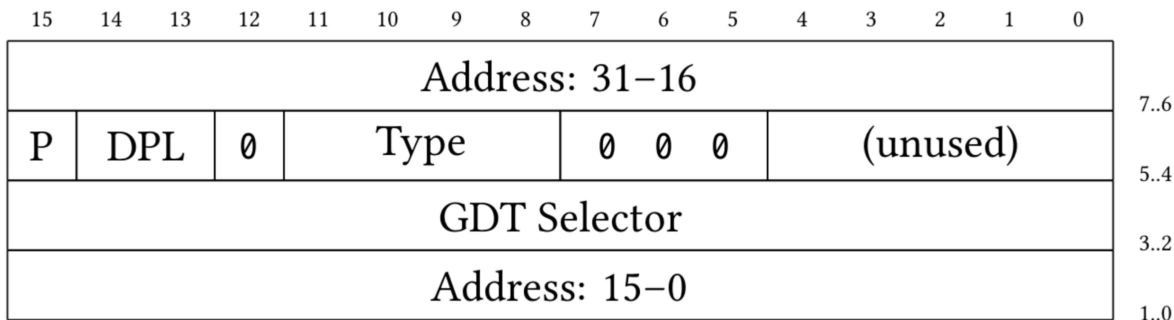
Chunk: ⟨PIC: set the initial interrupt mask⟩ (1)

- Interrupt-Maske legt fest, welche Interrupts beachtet werden
- anfangs: alles auf 0
- zwei `outportb`-Befehle notwendig (für zwei PICs)

⟨PIC: set the initial interrupt mask⟩≡
```
outportb (IO_PIC_MASTER_DATA, 0x00);  // PIC1: m↩
…ask 0
outportb (IO_PIC_SLAVE_DATA,  0x00);  // PIC2: m↩
…ask 0
```

Chunk: ⟨type definitions⟩ (1)

⟨type definitions⟩≡
```
struct idt_entry {
    unsigned int addr_low  : 16;   // lower 16 bits of address
    unsigned int gdtsel    : 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));
```



Chunk: ⟨type definitions⟩ (2)

- Wie bei GDT: wir zeigen mit Pointer auf IDT

⟨type definitions⟩+≡
```
struct idt_ptr {
    unsigned int limit   : 16;
    unsigned int base    : 32;
} __attribute__((packed));
```

Chunk: ⟨global variables⟩ (1)

- Platz für 256 Interrupt-Deskriptoren

⟨global variables⟩≡
```
struct idt_entry idt[256] = { 0 };
struct idt_ptr idtp;
```

Chunk: ⟨function prototypes⟩ (2)

- Funktion, die einen Eintrag der Tabelle füllt
- `gdtsel`: welches Segment verwenden (bei uns immer das Kernel-Code-Segment 0x08)
- `flags`: present, DPL (benötigter descriptor privilege level)
- `type`: 1110 (80386 32-bit interrupt gate)

⟨function prototypes⟩+≡
```
void fill_idt_entry (unsigned char num, unsigned↩
… long address,
    unsigned short gdtsel, unsigned char flags, ↩
…unsigned char type);
```

Chunk: ⟨function implementations⟩ (2)

⟨function implementations⟩+≡

```
void fill_idt_entry (unsigned char num, unsigned long address,
    unsigned short gdtsel, unsigned char flags, unsigned char 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].gdtsel    = gdtsel;           // GDT sel.: user mode or kernel mode?
    idt[num].zeroes    = 0;
    idt[num].flags     = flags;
    idt[num].type      = type;
  }
}
```

Chunk: ⟨function prototypes⟩ (3)

- Interrupt-Handler-Funktionen `irq0` bis `irq15` in `start.asm` definiert

⟨function prototypes⟩+≡

```
  extern void irq0(), irq1(), irq2(),  irq3(),  ir↩
…q4(),  irq5(),  irq6(),  irq7();
  extern void irq8(), irq9(), irq10(), irq11(), ir↩
…q12(), irq13(), irq14(), irq15();
```

Chunk: ⟨install the interrupt handlers⟩ (1)

⟨install the interrupt handlers⟩≡

```
⟨install the IDT⟩
⟨install the fault handlers⟩
⟨remap the interrupts to 32..47⟩
set_irqmask (0xFFFF);           // initialize IRQ mask
enable_interrupt (IRQ_SLAVE);   // IRQ slave

// flags: 1 (present), 11 (DPL 3), 0; type: 1110 (32 bit interrupt gate)
fill_idt_entry (32, (unsigned int)irq0,  0x08, 0b1110, 0b1110);
fill_idt_entry (33, (unsigned int)irq1,  0x08, 0b1110, 0b1110);
fill_idt_entry (34, (unsigned int)irq2,  0x08, 0b1110, 0b1110);
fill_idt_entry (35, (unsigned int)irq3,  0x08, 0b1110, 0b1110);
fill_idt_entry (36, (unsigned int)irq4,  0x08, 0b1110, 0b1110);
fill_idt_entry (37, (unsigned int)irq5,  0x08, 0b1110, 0b1110);
fill_idt_entry (38, (unsigned int)irq6,  0x08, 0b1110, 0b1110);
fill_idt_entry (39, (unsigned int)irq7,  0x08, 0b1110, 0b1110);
fill_idt_entry (40, (unsigned int)irq8,  0x08, 0b1110, 0b1110);
fill_idt_entry (41, (unsigned int)irq9,  0x08, 0b1110, 0b1110);
fill_idt_entry (42, (unsigned int)irq10, 0x08, 0b1110, 0b1110);
fill_idt_entry (43, (unsigned int)irq11, 0x08, 0b1110, 0b1110);
fill_idt_entry (44, (unsigned int)irq12, 0x08, 0b1110, 0b1110);
fill_idt_entry (45, (unsigned int)irq13, 0x08, 0b1110, 0b1110);
fill_idt_entry (46, (unsigned int)irq14, 0x08, 0b1110, 0b1110);
fill_idt_entry (47, (unsigned int)irq15, 0x08, 0b1110, 0b1110);
```

Chunk: ⟨function prototypes⟩ (4)

- Funktionen, die die Interrupt-Maske ändern
- `set_irqmask`: legt alle 16 Bits fest
- `get_irqmask`: liest alle 16 Bits aus
- `enable_interrupt`: schaltet einen bestimmten IRQ ein

⟨function prototypes⟩+≡

```
static void set_irqmask (unsigned short mask);
static void enable_interrupt (int number);
unsigned short get_irqmask ();
```

Chunk: ⟨function implementations⟩ (3)

⟨function implementations⟩+≡
```
static void set_irqmask (unsigned short mask) {
  outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
  outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8)  );
}

unsigned short get_irqmask () {
  return inportb (IO_PIC_MASTER_DATA)
      + (inportb (IO_PIC_SLAVE_DATA) << 8);
}

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

Chunk: ⟨kernel main: initialize system⟩ (1)

- Interrupts während der Kernel-Initialisierung aktivieren
- gleich hinter Paging und GDT

⟨kernel main: initialize system⟩≡
  ⟨install the interrupt handlers⟩

Chunk: ⟨type definitions⟩ (3)

- Interrupt-Handler sollen Zugriff auf Register-Inhalte bekommen
- werden alle
  `void handler (context_t *r)`

⟨type definitions⟩+≡
```
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;
```

Chunk: ⟨start.asm⟩ (1)

⟨*start.asm*⟩≡

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

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

irq0:  irq_macro 32
irq1:  irq_macro 33
irq2:  irq_macro 34
irq3:  irq_macro 35
irq4:  irq_macro 36
irq5:  irq_macro 37
irq6:  irq_macro 38
irq7:  irq_macro 39
irq8:  irq_macro 40
irq9:  irq_macro 41
irq10: irq_macro 42
irq11: irq_macro 43
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
irq15: irq_macro 47

extern irq_handler           ; defined in the C source file

irq_common_stub:             ; this is the identical part
       pusha
       push ds
       push es
       push fs
       push gs
       push esp  ; pointer to the context_t
       call irq_handler      ; call C function
       pop esp
       pop gs
       pop fs
       pop es
       pop ds
       popa
       add esp, 8
       iret
```

Chunk: ⟨function implementations⟩ (4)

- allgemeiner Handler muss Interrupt bestätigen, ggf. an beide PICs (sonst kommen keine neuen)
- ruft dann den eingetragenen Handler auf

⟨function implementations⟩+≡

```
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

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

Chunk: ⟨constants⟩ (3)

⟨constants⟩+≡

```
#define END_OF_INTERRUPT  0x20
```

Chunk: ⟨global variables⟩ (2)

- Platz für die Adressen von 16 spezifischen Handlern

⟨global variables⟩+≡

```
void *interrupt_handlers[16] = { 0 };
```

Chunk: ⟨function prototypes⟩ (5)

- Eintragen eines Handlers in die Tabelle
- (keine Funktion fürs Löschen; einfach `interrupt_handlers[n] = NULL`)

⟨function prototypes⟩+≡

```
void install_interrupt_handler (int irq, void (*↩
…handler)(context_t *r));
```

Chunk: ⟨function implementations⟩ (5)

⟨function implementations⟩+≡

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

Chunk: ⟨install the IDT⟩ (1)

- Vorbereiten des Zeigers auf die IDT

⟨install the IDT⟩≡

```
idtp.limit = (sizeof (struct idt_entry) * 256) -↩
… 1;   // must do -1
idtp.base  = (int) &idt;
idt_load ();
```

Chunk: ⟨function prototypes⟩ (6)

- idt_load ist in start.asm

⟨function prototypes⟩+≡
```
extern void idt_load ();
```

Chunk: ⟨start.asm⟩ (2)

- ähnlich wie gdt_flush

⟨start.asm⟩+≡
```
extern idtp  ; defined in the C file
global idt_load
idt_load:    lidt [idtp]
             ret
```

```
extern gp   ; defined in the C file
global gdt_flush

gdt_flush: lgdt [gp]
           mov ax, 0x10
           mov ds, ax
           mov es, ax
           mov fs, ax
           mov gs, ax
           mov ss, ax
           jmp 0x08:flush2
flush2:    ret
```

```
extern idtp   ; defined in the C file
global idt_load

idt_load: lidt [idtp]
          ret
```

Chunk: ⟨function prototypes⟩ (7)

- Fault-Handler: gleicher Ansatz wie bei Interrupt-Handlern
- 31 Faults (div/0, page fault, general protection fault etc.)
- Funktionen isr0 bis isr31 in start.asm

⟨function prototypes⟩+≡
```
extern void isr0(),  isr1(),  isr2(),  isr3(),  ↩
…isr4(),  isr5(),
     isr6(),  isr7(),  isr8(),  isr9(),  isr10(), ↩
…isr11(), isr12(),
     isr13(), isr14(), isr15(), isr16(), isr17(), ↩
…isr18(), isr19(),
     isr20(), isr21(), isr22(), isr23(), isr24(), ↩
…isr25(), isr26(),
     isr27(), isr28(), isr29(), isr30(), isr31();
```

Chunk: ⟨macros⟩ (1)

⟨macros⟩≡
```
#define FILL_IDT(i) \
  fill_idt_entry (i, (unsigned int)isr##i, 0x08, 0b1110, 0b1110)
```

Chunk: ⟨install the fault handlers⟩ (1)

⟨install the fault handlers⟩≡

```
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);
```

Chunk: ⟨start.asm⟩ (3)

⟨start.asm⟩+≡

```
global isr0,  isr1,  isr2,  isr3,  isr4,  isr5,  isr6,  isr7,  isr8
global isr9,  isr10, isr11, isr12, isr13, isr14, isr15, isr16, isr17
global isr18, isr19, isr20, isr21, isr22, isr23, isr24, isr25, isr26
global isr27, isr28, isr29, isr30, isr31
```

Chunk: ⟨global variables⟩ (3)

⟨global variables⟩+≡

```
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
};
```

Chunk: ⟨function prototypes⟩ (8)

- `fault_handler` ist vergleichbar mit `irq_handler`
- aber: hier verwalten wir keine Liste mit speziellen Handlern
- `fault_handler` muss Prozesse abbrechen, die Fault verursacht haben

⟨function prototypes⟩+≡

```
void fault_handler (context_t *r);
```

Chunk: ⟨function implementations⟩ (6)

⟨function implementations⟩+≡

```
void fault_handler (context_t *r) {
  if (r->int_no >= 0 && r->int_no < 32) {
    ⟨fault handler: display status information⟩

    if ( (unsigned int)(r->eip) < 0xc0000000 ) {     // user mode
      ⟨fault handler: terminate process⟩
    }

    printf ("System Stops\n");
    asm ("cli; \n hlt;");
  }
}
```

Chunk: ⟨fault handler: display status information⟩ (1)

⟨fault handler: display status information⟩≡

```
printf ("'%s' (%d) Exception at 0x%08x.\n",
  exception_messages[r->int_no], r->int_no, r->eip);
printf ("eflags: 0x%08x  errcode: 0x%08x\n", r->eflags, r->err_code);
printf ("eax: %08x  ebx: %08x  ecx: %08x  edx: %08x \n",
  r->eax, r->ebx, r->ecx, r->edx);
printf ("eip: %08x  esp: %08x  int: %8d  err: %8d \n",
  r->eip, r->esp, r->int_no, r->err_code);
printf ("ebp: %08x  cs: %d  ds: %d  es: %d  fs: %d  ss: %x \n",
  r->ebp, r->cs, r->ds, r->es, r->fs, r->ss);
```

Ausblick

- Es fehlt noch ein Page Fault Handler
  (kommt nach der Einführung in Prozesse)
- Neben Interrupt- und Fault-Handlern gibt es noch
  System Call Handler