Betriebssysteme

SS 2015

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz V:
- Ulix: Interrupts und Faults
- Ulix: System Calls

v1.0, 2015/05/28
(klassische Dokumentation)

Übersicht: BS Praxis und BS Theorie

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

Ulix: Interrupts und Faults

Interrupts: Bedarf bei Ulix

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

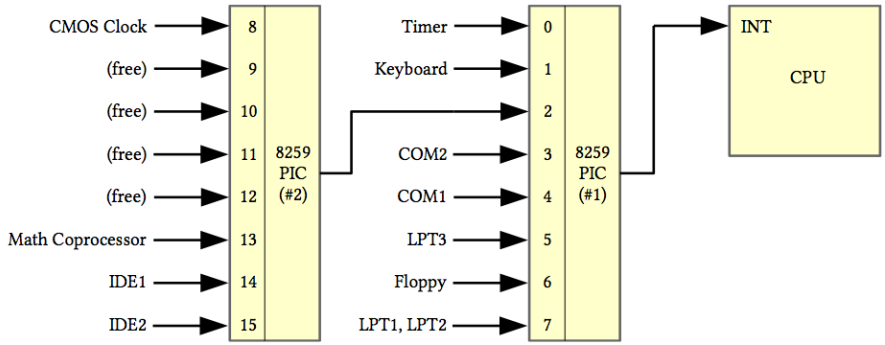
Faults

- 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



Konstanten für Interrupt-Nummern

irq.h (19–29)

```
19 // IRQ numbers 0 to 7 are handled by the "master PIC", numbers 8 to 15 are
20 // handled by the "slave PIC".
21 // The slave PIC generates an IRQ number 2 on the master PIC.
22
23 #define IRQ_TIMER      0
24 #define IRQ_KBD        1
25 #define IRQ_SLAVE      2     // Here the slave PIC connects to master
26 #define IRQ_COM2       3
27 #define IRQ_COM1       4
28 #define IRQ_FDC        6
29 #define IRQ_IDE       14     // primary IDE controller; secondary has IRQ 15
```

später: Interrupt-Handler für diese Interrupt-Nummern einrichten

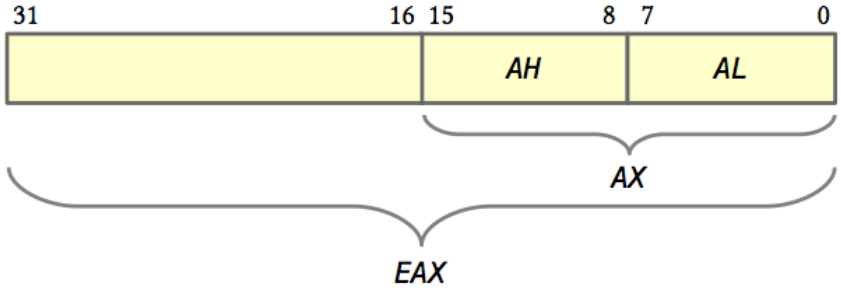
PICs programmieren

- 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)



Ein-/Ausgabe

`irq.c (31–39, 48–51)`

```
31 // The inport*() and outport*() functions retrieve a byte or word
32 // from a port or send it to the port.
33
34 byte
35 inportb (word port) {
36   byte retval;
37   asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
38   return retval;
39 }
```

```
48 void
49 outportb (word port, byte data) {
50   asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
51 }
```

analog: `inportw`, `outportw` (mit `inw`, `outw`)

PIC-Register

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

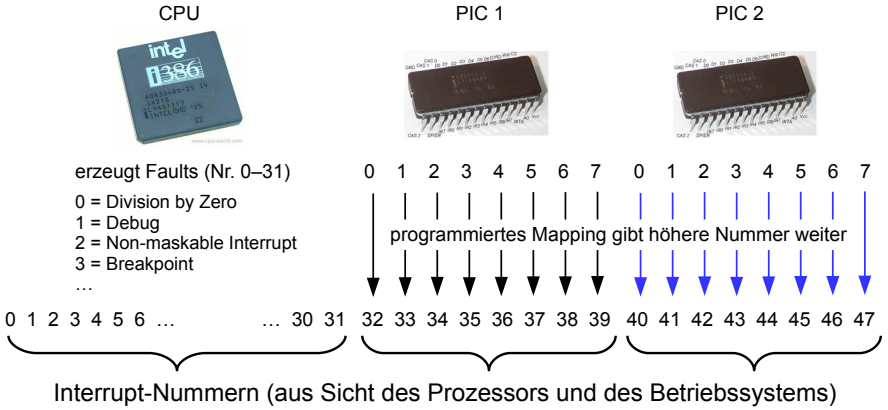
`irq.h (31–41)`

```
31 // The two PICs (programmable interrupt controllers) must be configured:
32 // they have to be aware of their master/slave states and how the slave
33 // connects to the master.
34
35 // I/O Addresses of the PICs:
36
37 #define IO_PIC_MASTER_CMD   0x20  // Master (IRQs 0-7), command register
38 #define IO_PIC_MASTER_DATA  0x21  // Master, control register
39
40 #define IO_PIC_SLAVE_CMD    0xA0  // Slave (IRQs 8-15), command register
41 #define IO_PIC_SLAVE_DATA   0xA1  // Slave, control register
```

PIC programmieren

Interrupt-Nummern remappen

Bildquellen: siehe Foliensatz D



PIC programmieren

- 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
  → `irq.c`, 165-166

PIC programmieren

- ICW2: Remapping festlegen durch Angabe des Offset; einmal 32 (0x20), einmal 40 (0x28)
  → `irq.c`, 167-170
- ICW3: Slave-Konfiguration
  → `irq.c`, 171-173
- ICW4: 8086 mode
  → `irq.c`, 174-175

PIC programmieren

`irq.c (165–175), setup_irqs_and_faults()`

```
165     outportb (IO_PIC_MASTER_CMD,  0x11);  // ICW1: initialize; begin programming
166     outportb (IO_PIC_SLAVE_CMD,   0x11);  // ICW1: dito, for PIC2
167     outportb (IO_PIC_MASTER_DATA, 0x20);  // ICW2 for PIC1: offset 0x20
168                                           //   (remaps 0x00..0x07 -> 0x20..0x27)
169     outportb (IO_PIC_SLAVE_DATA,  0x28);  // ICW2 for PIC2: offset 0x28
170                                           //   (remaps 0x08..0x0f -> 0x28..0x2f)
171     outportb (IO_PIC_MASTER_DATA, 0x04);  // ICW3 for PIC1: there's a slave on
172                                           //   IRQ 2 (0b00000100 = 0x04)
173     outportb (IO_PIC_SLAVE_DATA,  0x02);  // ICW3 for PIC2: your slave ID is 2
174     outportb (IO_PIC_MASTER_DATA, 0x01);  // ICW4 for PIC1 and PIC2: 8086 mode
175     outportb (IO_PIC_SLAVE_DATA,  0x01);
```

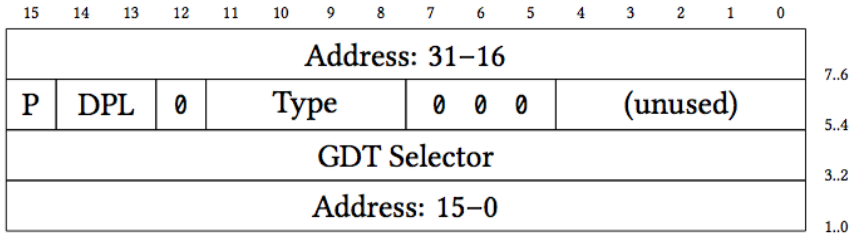
Interrupt-Handler

- 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**

Interrupt Descriptor (Typ)

irq.h (58–65)

```
56 // idt_entry is an entry in the Interrupt Descriptor Table (IDT)
57
58 struct idt_entry {
59     unsigned int addr_low  : 16;   // lower 16 bits of address
60     unsigned int gdtsel    : 16;   // use which GDT entry?
61     unsigned int zeroes    :  8;   // must be set to 0
62     unsigned int type      :  4;   // type of descriptor
63     unsigned int flags     :  4;
64     unsigned int addr_high : 16;   // higher 16 bits of address
65 } __attribute__((packed));
```



IDT-Pointer (Typ)

irq.h (71–74)

```
68 // idt_ptr describes address and size of the IDT. The address of
69 // this structure must be loaded in the IDTR (IDT register).
70
71 struct idt_ptr {
72     unsigned int limit   : 16;
73     unsigned int base    : 32;
74 } __attribute__((packed));
```


IDT und IDT-Pointer (Variablen)

- Globale Variablen:
  - Die Tabelle selbst (`idt`)
  - der **IDT Pointer** auf die Tabelle (`idt_ptr`)
  - Adresse von `idt_ptr` später in IDTR schreiben

globals.h (82–83)

```
81 // interrupts
82 struct idt_entry idt[256] = { { 0 } };
83 struct idt_ptr idtp;
```

Interrupt Descriptor füllen (1)

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

Interrupt Descriptor füllen (2)

irq.c (72–83)

```
59 // fill_idt_entry() fills an entry in the interrupt descriptor table (IDT).
60 //
61 // arguments:
62 // - byte num:  entry number (0..255)
63 // - unsigned long address:  address of the interrupt handler function
64 // - word gdtsel:  index into the global descriptor table (GDT), this
65 //     will always be set to 0x08 since we have prepared segment 0x08 as
66 //     code/kernel mode (see fill_gdt_entry in memory.c)
67 // - byte flags:  flags for this selector; these will always be set to
68 //     0b1110 (1 = present, 11 = DPL 3, 0)
69 // - byte type:  selector type; this will always be set to 0b1110
70 //     (32 bit interrupt gate)
71
72 void
73 fill_idt_entry (byte num, unsigned long address,
74     word gdtsel, byte flags, byte type) {
75   if (num >= 0 && num < 256) {
76     idt[num].addr_low  = address & 0xFFFF; // address is the handler address
77     idt[num].addr_high = (address >> 16) & 0xFFFF;
78     idt[num].gdtsel    = gdtsel;           // GDT sel.: user or kernel mode?
79     idt[num].zeroes    = 0;
80     idt[num].flags     = flags;
81     idt[num].type      = type;
82   }
83 }
```


Interrupt-Handler

- `irq0` bis `irq15` sind die Handler-Funktionen
  → Implementation in der Assembler-Datei
- einfacheres „Ansprechen“ über ein Array:

globals.h (84–87)

```
84 void (*irqs[16])() = {
85   irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7,      // store them in
86   irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15      // an array
87 };
```


Eintragen der Interrupt-Handler in IDT

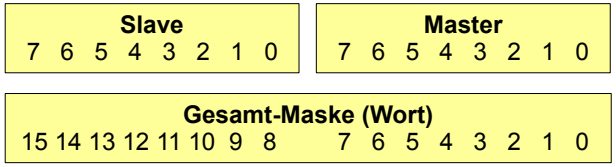
irq.c (181–188), setup_irqs_and_faults()

```
181   for (int i = 0;  i < 16;  i++) {
182     fill_idt_entry (32 + i,
183                     (unsigned int)irqs[i],
184                     0x08,
185                     0b1110,     // flags: 1 (present), 11 (DPL 3), 0
186                     0b1110);    // type:  1110 (32 bit interrupt gate)
187   }
188 }
```

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

Interrupt-Maske setzen und lesen (1)

- `set_irq_mask()` – Setzen der Interrupt-Maske
  → `irq.c`, 92-96
- `get_irq_mask()` – Auslesen der Maske
  → `irq.c`, 104-108
- Maske ist ein 16-Bit-Wort (1 Bit pro Interrupt)



Interrupt-Maske setzen und lesen (2)

`irq.c (92–96, 104–108)`

```
86  // set_irqmask() sets the 16 bit IRQ mask.
87  //
88  // Since we use two PICs (with each one handling eight interrupts),
89  // we send the lower 8 bits to the primary PIC and the upper 8 bits
90  // to the secondary PIC.
91
92  void
93  set_irqmask (word mask) {
94    outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
95    outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8)  );
96  }
97
98
99  // get_irqmask() reads the IRQ mask.
100 //
101 // Similar to set_irqmask(), we need to query both PICs. They each
102 // return eight bits which we combine to form a 16 bit mask value.
103
104 word
105 get_irqmask () {
106   return inportb (IO_PIC_MASTER_DATA)
107       + (inportb (IO_PIC_SLAVE_DATA) << 8);
108 }
```


Interrupt aktivieren

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

`irq.c (116–122)`

```
111 // enable_interrupt() enables a specific interrupt.
112 //
113 // It reads the IRQ mask, clears the bit indicated by number and
114 // writes the modified mask back to the PICs.
115
116 void
117 enable_interrupt (int number) {
118   set_irqmask (
119     get_irqmask ()        // the current value
120     & ~(1 << number)      // 16 one-bits, but bit "number" cleared
121   );
122 }
```

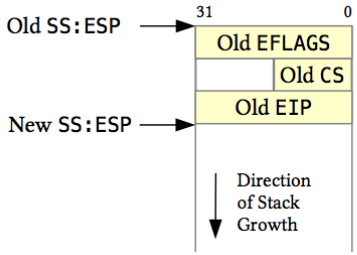

Stack-Einsatz beim Interrupt (1)

- 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)

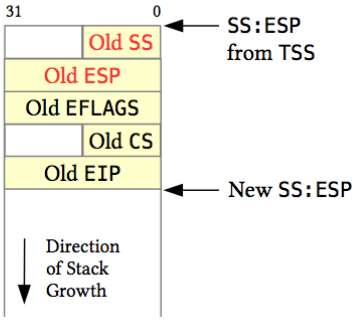
CPU im Kernel Mode (Ring 0)

→ 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



Handler-Funktionen

- Alle Interrupt-Handler haben die Signatur

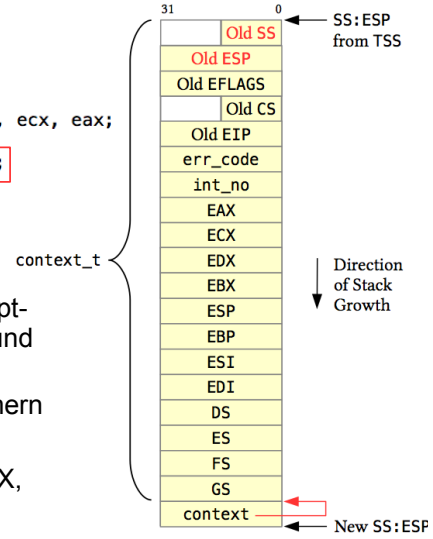
```
void handler_function (context_t *r);
```

- 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

Prozess-Kontext

ulix.h (624–629)

```
624 typedef struct {
625   unsigned int gs, fs, es, ds;
626   unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
627   unsigned int int_no, err_code;
628   unsigned int eip, cs, eflags, useresp, ss;
629 } 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

- Interrupt-Nummer und Fehlercode
- Standardregister (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, DS, ES, FS, GS)

Bsp. für kompletten Handler (IRQ 15)

```
1  push byte 0          ; error code
2  push byte 15         ; interrupt number
3  pusha
4  push ds
5  push es
6  push fs
7  push gs
8  push esp             ; pointer to the context_t
9  call irq_handler     ; call C function
10 pop  esp
11 pop  gs
12 pop  fs
13 pop  es
14 pop  ds
15 popa
16 add  esp, 8          ; for errcode, irq no.
17 iret
```

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

Diesen Code bräuchten wir jetzt 16 mal …

Makro für alle 16 Interrupt-Handler (1)

start.asm (108–112, 131–147)

```
108 %macro irq_macro 1
109   push  byte 0           ; error code (none)
110   push  byte %1          ; interrupt number
111   jmp irq_common_stub  ; rest is identical for all handlers
112 %endmacro

131 extern irq_handler          ; defined in the C source file
132 irq_common_stub:            ; this is the identical part
133   pusha
134   push  ds
135   push  es
136   push  fs
137   push  gs
138   push  esp          ; pointer to the context_t
139   call  irq_handler  ; call C function
140   pop   esp
141   pop   gs
142   pop   fs
143   pop   es
144   pop   ds
145   popa
146   add esp, 8
147   iret
```


Makro für alle 16 Interrupt-Handler (2)

start.asm (114–129)

```
114 irq0: irq_macro 32
115 irq1: irq_macro 33
116 irq2: irq_macro 34
117 irq3: irq_macro 35
118 irq4: irq_macro 36
119 irq5: irq_macro 37
120 irq6: irq_macro 38
121 irq7: irq_macro 39
122 irq8: irq_macro 40
123 irq9: irq_macro 41
124 irq10:  irq_macro 42
125 irq11:  irq_macro 43
126 irq12:  irq_macro 44
127 irq13:  irq_macro 45
128 irq14:  irq_macro 46
129 irq15:  irq_macro 47
```


Allgemeiner Interrupt-Handler (1)

- Wird von Assembler-Funktionen `irq0` … `irq15` aufgerufen
- schickt „End of Interrupt“ an den ersten oder an beide Controller

irq.h (47)

```
47 #define END_OF_INTERRUPT    0x20
```

- ruft spezifischen Handler auf, dessen Adresse in Handler-Array gespeichert ist

globals.h (88)

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


Allgemeiner Interrupt-Handler (2)

irq.c (206–218)

```
191 // irq_handler() is the generic interrupt handler.
192 //
193 // It is called from the assembler functions irq0(), ..., irq15()
194 // which have already prepared the stack so that the interrupt number
195 // and process context are located on top of it (allowing irq_handler
196 // to access that data structure via its content_t *r argument).
197 //
198 // The primary PIC must be acknowledged via END_OF_INTERRUPT (in all
199 // cases). If the interrupt was raised by the secondary PIC, it must
200 // also be acknowledged. (Otherwise the PIC would stop sending
201 // further interrupt notifications.)
202 //
203 // If a handler function was entered in the interrupt_handlers[]
204 // table, it is called.
205
206 void
207 irq_handler (context_t *r) {
208   int number = r->int_no - 32;                    // interrupt number
209   void (*handler)(context_t *r);                  // type of handler functions
210
211   if (number >= 8) {
212     outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT);  // notify slave  PIC
213   }
214   outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);   // notify master PIC (always)
215
216   handler = interrupt_handlers[number];
217   if (handler != NULL)  handler (r);
218 }
```

Bestimmten Interrupt-Handler einrichten

irq.c (230–234)

```
221 // install_interrupt_handler() installs a new interrupt handler
222 //
223 // arguments:
224 // - int irq:  entry number for the interrupt_handlers[] array (0..15)
225 // - void (*handler)(context_t *r):  address of the handler function
226 //
227 // If irq is a valid number, the given address is entered into the
228 // right entry of the array that stores the handler addresses.
229
230 void
231 install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
232   if (irq >= 0 && irq < 16)
233     interrupt_handlers[irq] = handler;
234 }
```

Während Kernel-Initialisierung:

```
install_interrupt_handler (IRQ_IDE,   ide_handler);               // in block.c
install_interrupt_handler (IRQ_FDC,   floppy_handler);            // in block.c
install_interrupt_handler (IRQ_COM2,  serial_hard_disk_handler);  // in fs.c
install_interrupt_handler (IRQ_KBD,   keyboard_handler);          // in keyboard.c
install_interrupt_handler (IRQ_TIMER, timer_handler);             // in timer.c
```


CPU über IDT informieren

irq.c (153–157)

```
153 void
154 setup_irqs_and_faults () {
155   idtp.limit = (sizeof (struct idt_entry) * 256) - 1;   // must do -1
156   idtp.base  = (int) &idt;
157   idt_load ();
```

start.asm (149–153)

```
149 extern idtp                 ; defined in the C file
150 global idt_load
151 idt_load:
152         lidt [idtp]
153         ret
```


Faults

Fault-Handler

Analog zu `irq0` … `irq15`:

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

ulix.h (721–726)

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

(Diese Funktionen sind wieder in Assembler geschrieben.)

Fault-Handler in IDT eintragen

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

irq.c (129–130, 158–164)

```
125 // FILL_IDT() calls fill_idt_entry() with default values. It is
126 // used for entering the fault0(), fault1(), ..., fault(31) and
127 // fault128() functions (which are defined in the assembler code).
128
129 #define FILL_IDT(i) \
130   fill_idt_entry (i, (unsigned int)fault##i, 0x08, 0b1110, 0b1110)
```

```
153 void
154 setup_irqs_and_faults () {
155   idtp.limit = (sizeof (struct idt_entry) * 256) - 1;   // must do -1
156   idtp.base  = (int) &idt;
157   idt_load ();
158   FILL_IDT( 0); FILL_IDT( 1); FILL_IDT( 2); FILL_IDT( 3); FILL_IDT( 4);
159   FILL_IDT( 5); FILL_IDT( 6); FILL_IDT( 7); FILL_IDT( 8); FILL_IDT( 9);
160   FILL_IDT(10); FILL_IDT(11); FILL_IDT(12); FILL_IDT(13); FILL_IDT(14);
161   FILL_IDT(15); FILL_IDT(16); FILL_IDT(17); FILL_IDT(18); FILL_IDT(19);
162   FILL_IDT(20); FILL_IDT(21); FILL_IDT(22); FILL_IDT(23); FILL_IDT(24);
163   FILL_IDT(25); FILL_IDT(26); FILL_IDT(27); FILL_IDT(28); FILL_IDT(29);
164   FILL_IDT(30); FILL_IDT(31); FILL_IDT(128);
```


Faults – ohne und mit Fehlercode

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
```


Makros für Fault Handler (Assembler)

start.asm (160–170)

```
160 %macro fault_macro_0 1
161   push byte 0  ; error code
162   push byte %1
163   jmp fault_common_stub
164 %endmacro
165
166 %macro fault_macro_no0 1
167   ; don't push error code
168   push byte %1
169   jmp fault_common_stub
170 %endmacro
```

start.asm (205–221)

```
205 extern fault_handler
206 fault_common_stub:
207   pusha
208   push  ds
209   push  es
210   push  fs
211   push  gs
212   push  esp                ; pointer to the context
213   call  fault_handler      ; call C function
214   pop esp
215   pop gs
216   pop fs
217   pop es
218   pop ds
219   popa
220   add esp, 8             ; for errcode, irq no.
221   iret
```

`fault_handler` ist wieder eine C-Funktion ...

Die 32 Fault-Handler (Assembler)

start.asm (172–203)

```
172 fault0:  fault_macro_0     0   ;  Divide by Zero
173 fault1:  fault_macro_0     1   ;  Debug
174 fault2:  fault_macro_0     2   ;  Non Maskable Interrupt
175 fault3:  fault_macro_0     3   ;  INT 3
176 fault4:  fault_macro_0     4   ;  INTO
177 fault5:  fault_macro_0     5   ;  Out of Bounds
178 fault6:  fault_macro_0     6   ;  Invalid Opcode
179 fault7:  fault_macro_0     7   ;  Coprocessor not available
180 fault8:  fault_macro_no0   8   ;  Double Fault
181 fault9:  fault_macro_0     9   ;  Coprocessor Segment Overrun
182 fault10: fault_macro_no0  10   ;  Bad TSS
183 fault11: fault_macro_no0  11   ;  Segment Not Present
184 fault12: fault_macro_no0  12   ;  Stack Fault
185 fault13: fault_macro_no0  13   ;  General Protection Fault
186 fault14: fault_macro_no0  14   ;  Page Fault
187 fault15: fault_macro_0    15   ;  (reserved)
188 fault16: fault_macro_0    16   ;  Floating Point
189 fault17: fault_macro_0    17   ;  Alignment Check
190 fault18: fault_macro_0    18   ;  Machine Check
191 fault19: fault_macro_0    19   ;  (reserved)
```

(… und weiter bis `fault31`; alle „reserved“)

Allgemeiner Fault-Handler (C-Teil)

irq.c (254–296)

```
254 void
255 fault_handler (context_t *r) {
256   if (r->int_no == 14) {
257     // fault 14 is a page fault
258     page_fault_handler (r); return;
259   }
260
261   memaddress fault_address = (memaddress)(r->eip);
262
263   // interrupt number should be in 0..31
264   if (r->int_no < 32) {
265     // output debugging information

280     if ( fault_address < 0xc0000000 ) {
281       // user mode: terminate current process

288     }
289
290     // error inside the kernel, jump to kernel shell
291     scheduler_is_active = false;  _set_statusline ("SCH:OFF", 16);
292     asm ("sti");
293     printf ("\n");
294     asm ("jmp kernel_shell");
295   }
296 }
```


System Calls

Vom User Mode in den Kernel Mode

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

System-Call-Tabelle

globals.h (136)

```
135 // system calls
136 void *syscall_table[MAX_SYSCALLS];
```

syscall.c (27–31, 69–132)

```
24  // install_syscall_handler() is used for installing a single
25  // system call handler function in the syscall table.
26
27  void
28  install_syscall_handler (int syscallno, void *syscall_handler) {
29    if (syscallno >= 0 && syscallno < MAX_SYSCALLS)
30      syscall_table[syscallno] = syscall_handler;
31  }

67  // setup syscall handlers
68
69  void
70  install_all_syscall_handlers () {
71    install_syscall_handler (__NR_brk, syscall_sbrk);
72    install_syscall_handler (__NR_get_errno, syscall_get_errno);
73    install_syscall_handler (__NR_set_errno, syscall_set_errno);
74    install_syscall_handler (__NR_fork, syscall_fork);
75    install_syscall_handler (__NR_exit, syscall_exit);
```

Allgemeiner Syscall-Handler

syscall.c (37–47)

```
34 // syscall_handler() is the generic system call handler which
35 // is called when a process executes the "int 0x80" instruction.
36
37 void
38 syscall_handler (context_t *r) {
39   void (*handler) (context_t*);   // handler is a function pointer
40   int number = r->eax;
41   if (number != __NR_get_errno) set_errno (0); // default: no error
42   handler = syscall_table[number];
43   if (handler != 0)  handler (r);
44   else
45     printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
46             "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
47 }
```


Syscall-Handler (Assembler)

Assembler-Funktion wie bei den anderen Interrupt-Handlern:

start.asm (254–272)

```
252 extern syscall_handler
253 global fault128
254 fault128:
255         push    byte 0          ; put 128 on the stack so it looks the same
256         ; push  byte 128    ; as it does after a hardware interrupt
257         push    byte -128       ; (getting rid of nasm error for signed byte)
258         pusha
259         push    ds
260         push    es
261         push    fs
262         push    gs
263         push    esp  ; pointer to the context_t
264         call    syscall_handler
265         pop     esp
266         pop     gs
267         pop     fs
268         pop     es
269         pop     ds
270         popa
271         add     esp, 8          ; undo the two "push byte" commands from the start_
272         iret
```


Bibliotheksfunktionen

- 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

ulixlib.c

```
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) ); };
```

- Bibiliotheksfunktion `read` (User Mode)

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