

Betriebssysteme 1

SS 2016

Prof. Dr.-Ing. Hans-Georg Eßer
Fachhochschule Südwestfalen

Foliensatz C:

- Geräte
- Interrupts und Faults
- System Calls

v1.1, 2016/05/17

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-1

Motivation: Tastatur am PC

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-2

Motivation: Tastatur am PC (1)

- Szenario: Standard-PC mit Bildschirm, Maus und Tastatur (PS/2-Anschluss)
- Frage: Wie kommen die Tastendrücke in laufenden Programmen an?



Quelle: https://commons.wikimedia.org/wiki/File:%3AIBM_Model_M.png

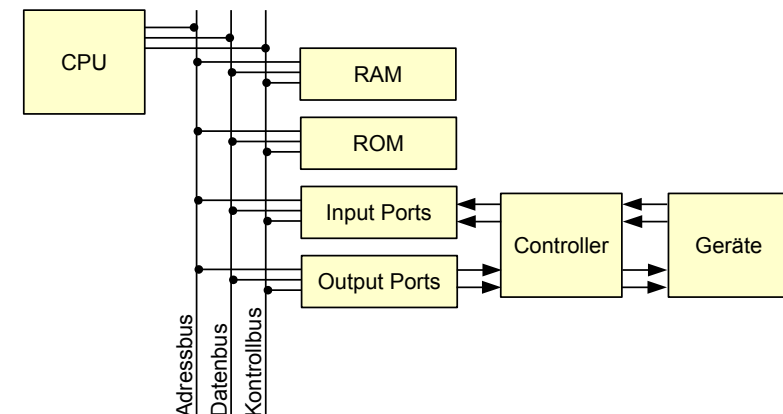
17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-3

Motivation: Tastatur am PC (2)

- Klassisch: Kommunikation über **Ports**



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-4

Motivation: Tastatur am PC (3)

- PS/2-Keyboard-Controller hat zwei solche Ports

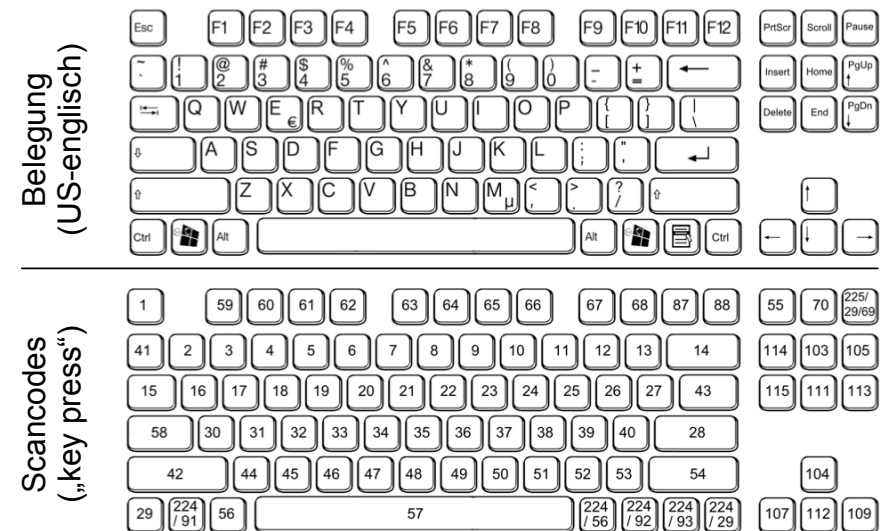
```
#define KBD_DATA_PORT    0x60
#define KBD_STATUS_PORT 0x64
```

- Kommunikation über Assembler-Befehle
`inb` (liest Wert aus Port, speichert in Reg.),
`outb` (schreibt Reg.-Inhalt auf Port)

Motivation: Tastatur am PC (4)

- Jeder „Event“ (Ereignis: Tastendruck oder Loslassen) generiert einen Scancode
- Beispiel, Taste „A“:
 - Drücken = Scancode 30
 - Loslassen = Scancode 30 + 128 = 158
- Scancodes über Datenport (0x60) des Keyboard-Controllers auslesen:
`char scancode = inb (KBD_DATA_PORT);`
- Statusport verrät, ob Event vorliegt

Motivation: Tastatur am PC (5)



Motivation: Tastatur am PC (6)

- Simpler Treiber (Pseudocode, vereinfacht)

```
do forever {
    // warte auf Event
    while (inb (KBD_STATUS_PORT) != NEW_EVENT) ;

    // lies Scancode
    scancode = inb (KBD_DATA_PORT);

    if (scancode < 128) {
        // nur keypress event verarbeiten
        ascii = lookup_table (scancode);
        printf ("Zeichen %d eingegeben\n", ascii);
    }
}
```

Port-Mapped vs. Memory-Mapped I/O

- auf gleiche Weise (**port-mapped I/O**, PMIO) Kommunikation mit anderen Geräten möglich, z. B. Platten-Controller
- Alternative: **memory-mapped I/O** (MMIO)
 - Controller hat eigenen Speicher
 - Einblendung in Adressraum
 - CPU liest/schreibt einfach „Hauptspeicher“



Interrupts

Einführung (1)

- Festplattenzugriff um mehr als Faktor 1.000.000 langsamer als Ausführen einer CPU-Anweisung
 - Taktfrequenz 1 GHz: 1.000.000.000 Instruktionen / s
 - Festplatte: 7.200 Umdrehungen / min = 120 U. / s
Im Schnitt: halbe Umdrehung (240 / s) nötig, um richtige Position zu erreichen (zzgl. Transferzeit)
 - Plattenzugriff braucht im Schnitt
 $1.000.000.000 / 240 \approx 4.166.666$ mal so lang wie eine CPU-Instruktion

Einführung (2)

- Naiver Ansatz für Plattenzugriff:

```
naiv () {
    rechne (500 ZE);
    sende_anfrage_an (disk);
    antwort = false;
    while ( ! antwort ) {
        /* diese Schleife rechnet 1.000.000 ZE lang */
        antwort = test_ob_fertig (disk);
    }
    rechne (500 ZE);
    return 0;
}
```

Einführung (3)

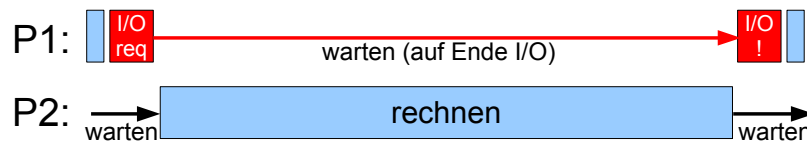
- Naiver Ansatz heißt „Pollen“: in Dauerschleife ständig wiederholte Geräteabfrage
- Pollen verbraucht sehr viel Rechenzeit:



- Besser wäre es, in der Wartezeit etwas anderes zu tun
- Auch bei Parallelbearbeitung mehrerer Prozesse: Polling immer noch ungünstig

Einführung (4)

- Idee: Prozess, der I/O-Anfrage gestartet hat, solange schlafen legen, bis die Anfrage bearbeitet ist – in der Zwischenzeit was anderes tun



- Woher weiß das System,
 - wann die Anfrage bearbeitet ist, also
 - wann der Prozess weiterarbeiten kann?

Einführung (5)

- Lösung: Interrupts – bestimmte Ereignisse können den „normalen“ Ablauf unterbrechen
- In der CPU: Nach jeder ausgeführten Anweisung prüfen, ob es einen Interrupt gibt (gab)

Interrupt-Klassen

- I/O (Eingabe/Ausgabe, **asynchrone** Interrupts)
 - Meldung vom I/O-Controller: „Aktion ist abgeschlossen“
 - Timer
- Hardware-Fehler, z. B. RAM-Parität
- Software-Interrupts (Exceptions, Traps, **synchrone** Interrupts)
 - Falscher Speicherzugriff, Division durch 0, unbekannte CPU-Instruktion, ...

Interrupts: Vor- und Nachteile

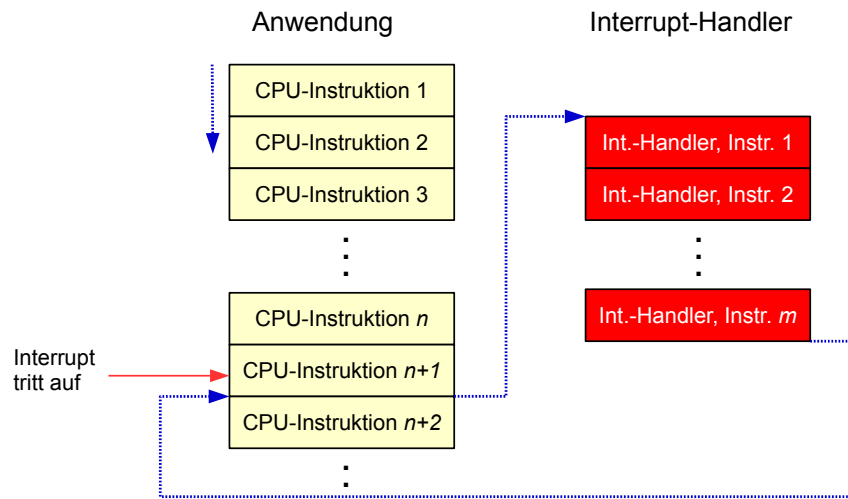
Vorteile

- Effizienz
I/O-Zugriff sehr langsam → sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist
- Programmierlogik
Nicht immer wieder Gerätestatus abfragen (Polling), sondern blockieren, bis passender Interrupt kommt

Nachteile

- Mehraufwand
Kommunikation mit Hardware wird komplexer, Instruction Cycle erhält zusätzlichen Schritt

Interrupt-Bearbeitung (1)



Interrupt-Bearbeitung (2)

Grundsätzlich

- Interrupt tritt auf
- Laufender Prozess wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
- BS speichert Daten des Prozesses (wie bei Prozesswechsel → Scheduler)
- BS ruft Interrupt-Handler auf
- Danach (evtl.): Prozess-Fortsetzung (evtl. ein anderer Prozess)

Interrupt-Bearbeitung (3)

Was tun bei Mehrfach-Interrupts?

Drei Möglichkeiten

- Während Abarbeitung eines Interrupts alle weiteren ausschließen (DI, disable interrupts) → Interrupt-Warteschlange
- Während Abarbeitung andere Interrupts zulassen
- Interrupt-Prioritäten: Nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer

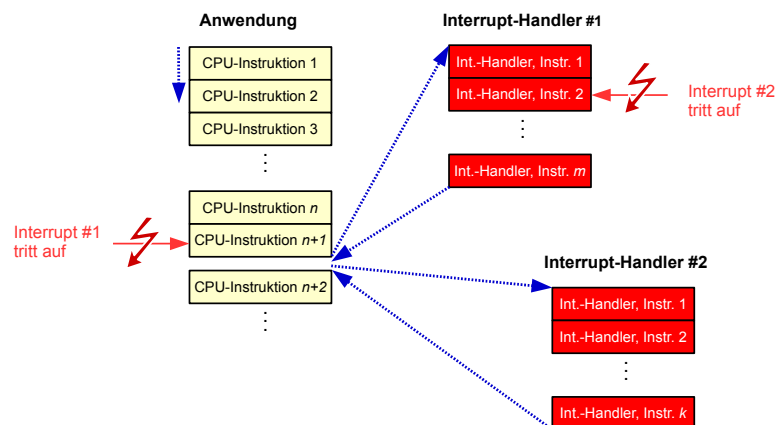
! ■ Mehrfach-Interrupts (1)

Variante 1

- Alle Interrupts „gleichwertig“, keine Prioritäten
- zu Beginn einer Int.-Routine alle Interrupts abschalten
- kein Interrupt unterbricht einen anderen

! ■ Mehrfach-Interrupts (2)

Variante 1



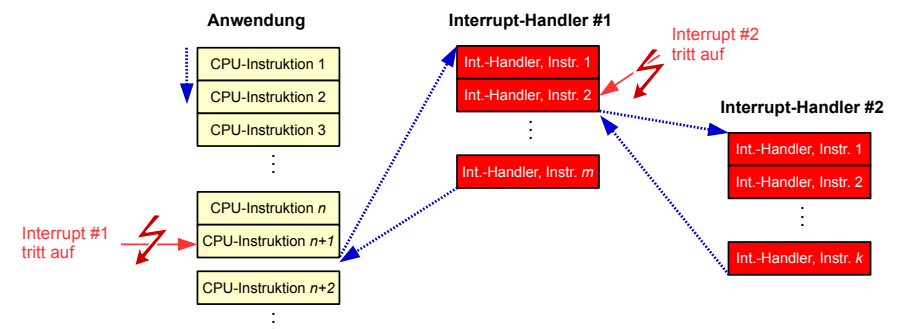
! ■ Mehrfach-Interrupts (3)

Variante 2

- Interrupt-Handler können unterbrochen werden
- Rücksprung in vorherigen Interrupt-Handler
- Zustand sichern!

! ■ Mehrfach-Interrupts (4)

Variante 2



Mehrfach-Interrupts (5)

Variante 3

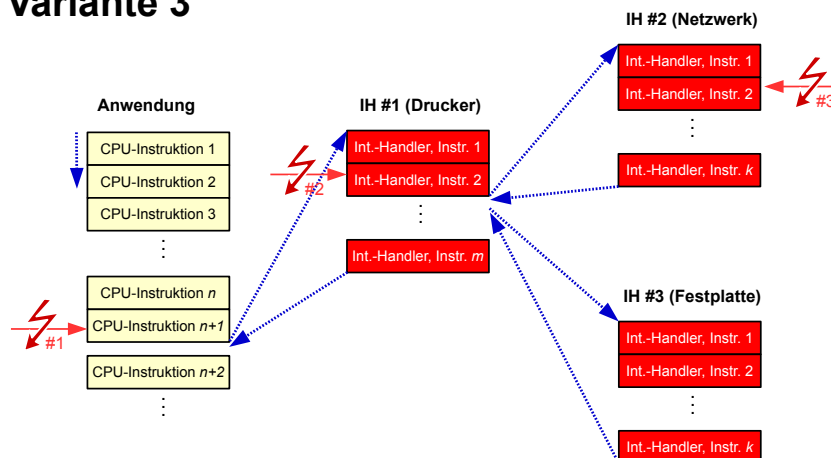
- Interrupts haben Prioritäten, z. B. Netzwerkkarte > Drucker
- Interrupt mit hoher Priorität unterbricht Interrupt mit niedrigerer Priorität

Mehrfach-Interrupts (7)

- Problem bei gesperrten Interrupts: Behandlung muss schnell erfolgen
- Lösung: Aufteilung des Int.-Handlers in zwei Komponenten
 - erste Komponente bestätigt Interrupt, sichert wichtige Informationen und gibt Interrupts wieder frei
 - zweite Komponente läuft später (bei aktivierten Interrupts) und erledigt restliche Aufgaben
- Beispiel: Linux „top half/bottom half“ → später

Mehrfach-Interrupts (6)

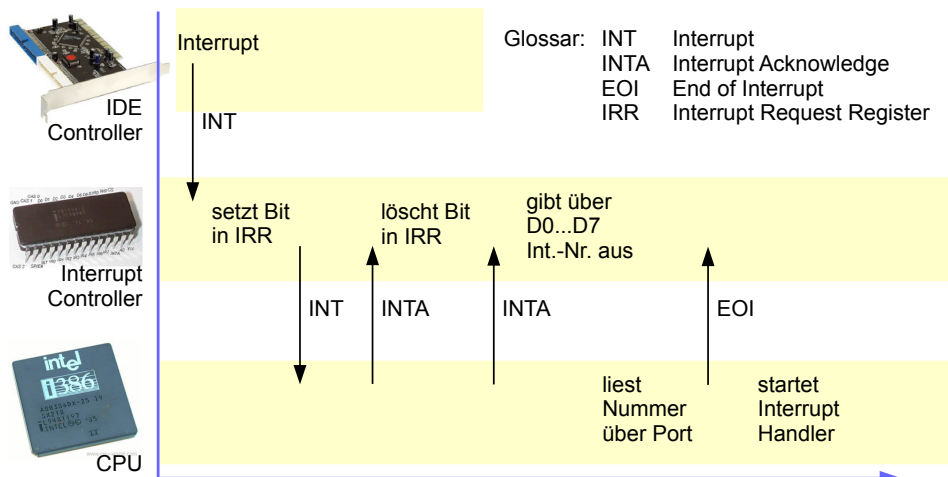
Variante 3



Interrupts unterscheiden (1)

- CPU hat nur einen Interrupt-Eingang – wie kann sie zwischen verschiedenen Geräten unterscheiden, die einen Interrupt erzeugen?
- Interrupt Controller
 - mehrere Eingänge (z. B. 8 beim Intel 8259)
 - ein Ausgang (zur CPU)
- Kommunikation der Interrupt-Nummer (an CPU) über zusätzliche Datenleitungen

Interrupts unterscheiden (2)



Bildquellen: IDE Controller: <http://www.ebay.de/usr/sm-pc>, 8259A: <http://www.brokenhorn.com/Resources/OSDevPic.html>, i386: <http://www.cpu-world.com/CPUs/80386/Intel-A80386DX-25%20IV.html>

17.05.2016

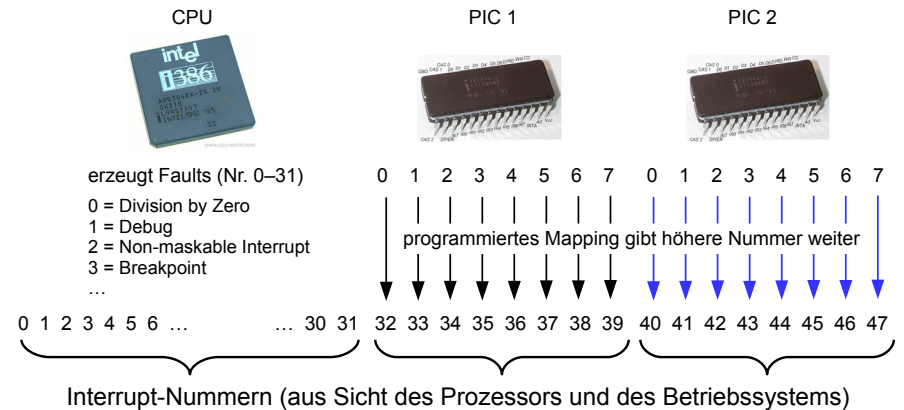
Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-29

Interrupts unterscheiden (4)

Intel 8259 ist programmierbar. Ziel:

Bildquellen:
siehe Folie 29



17.05.2016

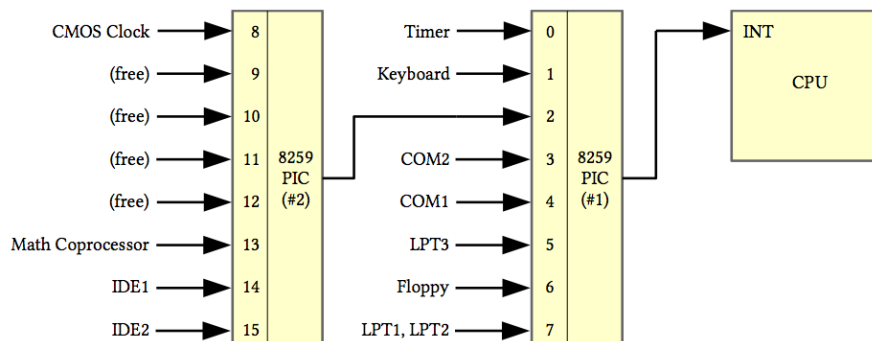
Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-31

Interrupts unterscheiden (3)

- PCs: Klassischer PIC (**Programmable Interrupt Controller**) Intel 8259 hat nur acht Eingänge

→ Kaskadierung von zwei PICs



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-30

Interrupts unterscheiden (5)

- Teilweise Mehrfachbelegung von Interrupts
- Betriebssystem muss alle Geräte (die sich diesen Interrupt teilen) befragen
- Beispiel Linux-Treiber:
 - für jedes Gerät einen Handler registrieren
 - Interrupt-Handler für Int.-Nummer X ruft nacheinander alle Handler (von Geräten mit Int.-Nr. X) auf – bis ein Handler sagt: „Das war mein Interrupt.“

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-32

I/O-lastig vs. CPU-lastig (1)

• CPU-lastiger Prozess

- Prozess benötigt überwiegend CPU-Rechenzeit und vergleichsweise wenig I/O-Operationen
- Längere Rechenphasen werden nur gelegentlich durch I/O-Wartezeiten unterbrochen

• I/O-lastiger Prozess

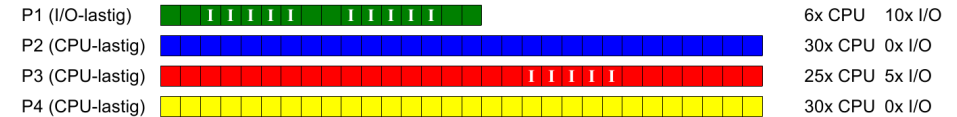
- Prozess führt viele I/O-Operationen durch und benötigt vergleichsweise wenig Rechenzeit
- Sehr kurze Rechenphasen wechseln sich mit häufigen Wartezeiten auf I/O ab

I/O-lastig vs. CPU-lastig (2)

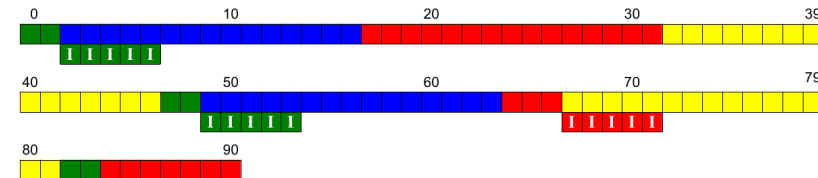
Multitasking und Interrupts

- Multitasking verbessert CPU-Nutzung:
 - I/O-lastiger Prozess wartet auf I/O-Events,
 - CPU-lastiger Prozess rechnet währenddessen weiter
- Prozess stößt I/O-Operation an und blockiert (wartet darauf, dass das BS ihn wieder auf „bereit“ setzt und irgendwann fortsetzt)
- optimale Performance: gute Mischung I/O- und CPU-lastiger Prozesse

I/O-lastig vs. CPU-lastig (3)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

Praxis: Interrupts unter Linux

Interrupt-Übersicht (Single-Core-CPU)

```
[esser@server ~]$ cat /proc/interrupts
CPU0
0: 3353946487      XT-PIC  timer
2:      0          XT-PIC  cascade
3:      4663       XT-PIC  NVidia CK804
5: 159275991       XT-PIC  ohci1394, nvidia
7:   971775       XT-PIC  hsfpcibasic2
8:      2         XT-PIC  rtc
9:      0         XT-PIC  acpi
10:   31052       XT-PIC  libata, ohci_hcd
11: 197906977      XT-PIC  libata, ehci_hcd
12: 16904921      XT-PIC  eth0
14: 60349322      XT-PIC  ide0
NMI:      0
LOC:      0
ERR:      0
MIS:      0
```

Interrupt Handler (1)

Für jedes Gerät:

- Interrupt Request (IRQ) Line
- Interrupt Handler (Interrupt Service Routine, ISR) → Teil des Gerätetreibers
- C-Funktion
- läuft in speziellem Context (Interrupt Context)
- „top half“ und „bottom half“

Moderne Maschine mit vier Cores

```
[esser@quad:~]$ cat /proc/interrupts
CPU0      CPU1      CPU2      CPU3
0: 5224      3          1          1  IO-APIC-edge  timer
1: 298114    774        793        793  IO-APIC-edge  i8042
3: 9          8          6          9  IO-APIC-edge
4: 8          9          8          6  IO-APIC-edge
8: 0          0          0          1  IO-APIC-edge  rtc0
9: 0          0          0          0  IO-APIC-fasteoi acpi
12: 3070145  16539      16542      16485  IO-APIC-edge  i8042
16: 2760924  881        904        886  IO-APIC-fasteoi uhci_hcd:usb1, nvidia
18: 24122388  6538       6698       6647  IO-APIC-fasteoi ehci_hcd:usb6, uhci_hcd:usb7
19: 281       28         27         10  IO-APIC-fasteoi uhci_hcd:usb3, uhci_hcd:usb5
21: 22790     0          0          0  IO-APIC-fasteoi uhci_hcd:usb2
22: 7786588  10464141  8251870    8439964  IO-APIC-fasteoi HDA Intel
23: 899       0          1          1  IO-APIC-fasteoi uhci_hcd:usb4, ehci_hcd:usb8
221: 9519152  10751650  9745810    10326363  PCI-MSI-edge  eth0
222: 14462926  38205     38095     38178  PCI-MSI-edge  ahci
NMI: 0          0          0          0  Non-maskable interrupts
LOC: 724999305 786034088 748693018 748218173  Local timer interrupts
RES: 5334382  3576152  3464671  3357556  Rescheduling interrupts
CAL: 2111668  4233550  4067655  3871450  function call interrupts
TLB: 101757  113319  88752  107777  TLB shootdowns
TRM: 0          0          0          0  Thermal event interrupts
SPU: 0          0          0          0  Spurious interrupts
ERR: 0
MIS: 0
```

Interrupt Handler (2)

„top half“ und „bottom half“

top half

- Interrupt handler
- startet sofort, erledigt zeitkritische Dinge
- bestätigt (der Hardware) den Erhalt des Interrupts, setzt Gerät zurück etc.
- erzeugt bottom half
- Alles andere → bottom half

Interrupt Handler (3)

Tasklet (bottom half)

- Tasklets führen längere Berechnungen durch, die zur Interrupt-Verarbeitung gehören – dabei sind Interrupts zugelassen
- Tasklet ist kein Prozess (struct tasklet_struct), läuft direkt im Kernel; im Interrupt-Context
- Zwei Prioritäten:
 - *tasklet_hi_schedule*: startet direkt nach ISR
 - *tasklet_schedule*: startet erst, wenn kein anderer Soft IRQ mehr anliegt

Interrupt Handler (4)

Mehr Informationen:

- [1] Linux Kernel 2.4 Internals, Kapitel 2,
http://www.faqs.org/docs/kernel_2_4/lki-2.html
- [2] J. Quade, E.-K. Kunst: „Linux-Treiber entwickeln“, dpunkt-Verlag,
<http://ezs.kr.hsnr.de/TreiberBuch/html/>

System Calls

User Mode vs. Kernel Mode (1)

- Anwendungen laufen im nicht-privilegierten User Mode
 - Beispiel: Intel
 - Ring 0 = Betriebssystem (Kernel Mode)
 - Vollzugriff auf die Hardware
 - Ring 3 = Anwendung (User Mode)

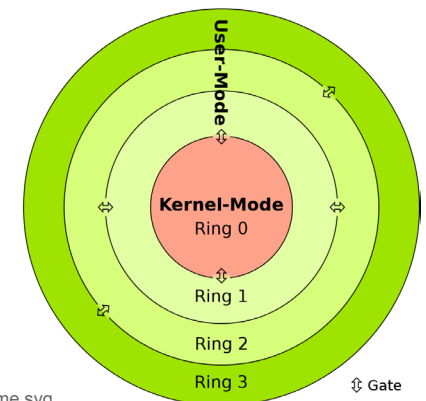
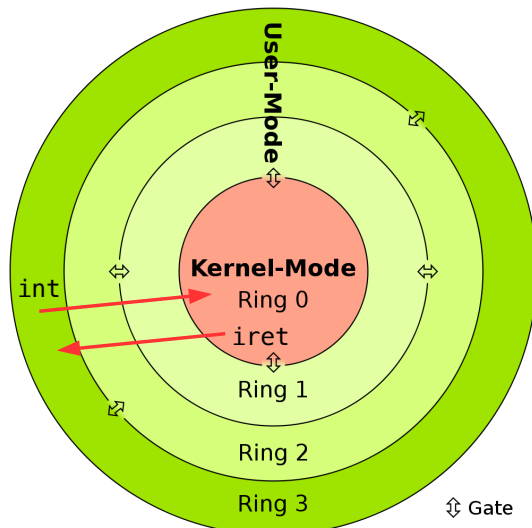


Bild: http://commons.wikimedia.org/wiki/File:CPU_ring_scheme.svg
(Autor: <http://commons.wikimedia.org/wiki/User:Sven>; GNU Free Documentation License)

User Mode vs. Kernel Mode (2)

- Problem:
 - Daten und Code des Betriebssystems sollen vor Zugriff durch Anwendungen geschützt sein,
aber:
 - Anwendungen müssen Betriebssystem-Funktionen nutzen, um z. B. auf Datenträger zuzugreifen.
- Lösung: System Calls – ein kontrollierter Übergang vom User Mode in den Kernel Mode
- Mechanismus: Software Interrupt (CPU-Instruktion)
 - Intel: z. B. `int 0x80`

User Mode vs. Kernel Mode (3)



User Mode vs. Kernel Mode (4)

Vorgehensweise:

- Anwendung trägt **System-Call-Nummer** in ein Register ein (oder schreibt sie auf den Stack)
- Parameter für den System Call: in weitere Register (oder Stack)
- Software-Interrupt auslösen → durch spezielle CPU-Instruktion, z. B.
 - `sysenter` (Intel, ab Pentium II),
 - `syscall` (AMD64)
 - `int` (alle Intel und kompatibel)
- CPU reagiert auf `int`-Instruktion wie auf einen HW-Interrupt und ruft Interrupt-Handler auf

User Mode vs. Kernel Mode (5)

Vorgehensweise (Fortsetzung):

- generischer Interrupt-Handler für System-Call-Behandlung liest System-Call-Nummer (aus Register oder vom Stack)
- über **System-Call-Tabelle** den richtigen **System-Call-Handler** identifizieren und aufrufen
- im spezifischen System-Call-Handler:
 - Argumente auswerten
 - prüfen, ob Anwendung zum Aufruf (diese Funktion, mit diesen Parametern) berechtigt ist
 - Aufruf geeigneter Kernel-Funktionen

User Mode vs. Kernel Mode (6)

Vorgehensweise (Fortsetzung):

- nach Bearbeitung:
 - Rücksprung aus spezifischem System-Call-Handler (ggf. mit Rückgabe eines Ergebnis-Werts)
 - Rücksprung aus Interrupt-Handler: `sysleave` (Pentium II), `sysret` (AMD), `iret` (Intel); ggf. mit Rückgabe des Ergebnis aus dem Syscall-Handler
→ Übergang von Ring 0 zurück in Ring 3
 - Fortsetzung der Prozess-Ausführung (ggf. Auswertung des Syscall-Ergebnis-Werts)
- zur Vereinfachung für Anwendungs-Entwickler: **User-Mode-Bibliothek** mit Wrappern für die System Calls

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-49

Beispiel für Anwendung (Linux)

- Ausgabe auf stdout

```
_start:                ; tell linker entry point
    mov edx, len        ; message length
    mov ecx, msg         ; message to write
    mov ebx, 1           ; file descriptor (stdout)
    mov eax, 4           ; system call number (sys_write)
    int 0x80            ; software interrupt 0x80
    mov eax, 1           ; system call number (sys_exit)
    int 0x80            ; software interrupt 0x80

section .data
msg db 'Hello, world!', 0xa ; the string to be printed
len equ $ - msg           ; length of the string
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

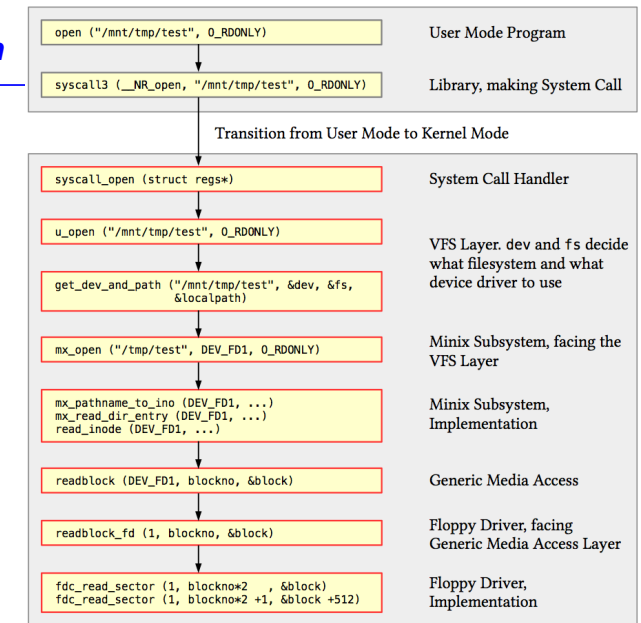
Folie C-50

Beispiel Implementation

Beschreibung
folgt später

`syscall3()`

- kopiert drei Argumente in EAX, EBX, ECX
- führt int 0x80 aus
- liest Rückgabewert aus EAX



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-51

System-Call-Nummern

`/usr/include/asm/unistd_32.h`: Über 300 System Calls

```
/*
 * This file contains the system call
 * numbers.
 */

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16

#define __NR_break 17
#define __NR_oldstat 18
#define __NR_lseek 19
#define __NR_getpid 20
#define __NR_mount 21
#define __NR_umount 22
#define __NR_setuid 23
#define __NR_getuid 24
#define __NR_stime 25
#define __NR_ptrace 26
#define __NR_alarm 27
#define __NR_oldfstat 28
#define __NR_pause 29
#define __NR_utime 30
#define __NR_stty 31
#define __NR_gtty 32
#define __NR_access 33
#define __NR_nice 34
#define __NR_ftime 35
#define __NR_sync 36
#define __NR_kill 37
...
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-52

System Calls für Programmierer:

Standardfunktionen in C

System Calls / User-Mode-Bibliothek

- Standardbibliotheken stellen Wrapper für System Calls bereit
- hier nur kleine Auswahl:
 - Dateizugriff: open, read, write, close
 - Prozesse: fork, exit, execl(p)

open () Daten zum Lesen/Schreiben öffnen

```
int open(const char *pathname, int flags);  
int open(const char *pathname, int flags, mode_t mode);  
int creat(const char *pathname, mode_t mode);
```

Rückgabewert: File Descriptor

man 2 open

Beispiel:

```
fd = open("/tmp/datei.txt", O_RDONLY);
```

Linux System Calls (3)

read () Daten aus Datei (File Descriptor) lesen

```
ssize_t read(int fd, void *buf, size_t count);
```

Rückgabewert: Anzahl gelesene Bytes

man 2 read

Beispiel:

```
int bufsiz=128; char line[bufsiz+1];  
int fd = open( "/etc/fstab", O_RDONLY );  
int len = read ( fd, line, bufsiz );
```

Linux System Calls (4)

Beispiel: read() und open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main (void) {
    int len; int bufsiz=128; char line[bufsiz+1];
    line[bufsiz] = '\0';
    int fd = open( "/etc/fstab", O_RDONLY );
    while ( (len = read ( fd, line, bufsiz )) > 0 ) {
        if ( len < bufsiz ) { line[len]='\0'; }
        printf ("%s", line );
    }
    close(fd);
    return 0;
}
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-57

Linux System Calls (5)

write() Daten in Datei (File Descriptor) schreiben

```
ssize_t write(int fd, void *buf, size_t count);
```

Rückgabewert: Anzahl geschriebene Bytes

man 2 write

Beispiel:

```
main() {
    char message[] = "Hello world\n";
    int fd = open( "/tmp/datei.txt",
        O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR );
    write ( fd, message, sizeof(message) );
    close(fd);
    exit(0);
}
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-58

Linux System Calls (6)

close() Datei (File Descriptor) schließen

```
int close(int fd);
```

Rückgabewert: 0 bei Erfolg, sonst -1 (errno enthält dann Grund)

man 2 close

Beispiel:

```
close(fd);
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-59

Linux System Calls (7)

exit() Programm beenden

```
void exit(int status);
```

Kein Rückgabewert, aber *status* wird an aufrufenden Prozess weitergegeben.

man 3 exit

Beispiel:

```
exit(0);
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-60

Linux System Calls (8)

fork() neuen Prozess starten

```
pid_t fork(void);
```

Rückgabewert: Child-PID (im Vaterprozess); 0 (im Sohnprozess); -1 (im Fehlerfall)

man fork

Beispiel:

```
pid=fork()
```

Header-Dateien einbinden

Am Anfang jedes C-Programms:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
```

sys/stat.h enthält z. B. S_IRUSR, S_IWUSR
fcntl.h enthält z. B. O_CREAT, O_WRONLY

Linux System Calls (9)

exec() Anderes Programm im Prozess laden

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execev(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Rückgabewert: keiner (Funktion kehrt nicht zurück)

Parameter arg0 (Name), arg1, ...; letztes Argument: NULL-Zeiger

man 3 exec

Beispiele:

```
execl ("/usr/bin/vi", "", "/etc/fstab", (char *) NULL);
execlp ("vi", "", "/etc/fstab", (char *) NULL);
```

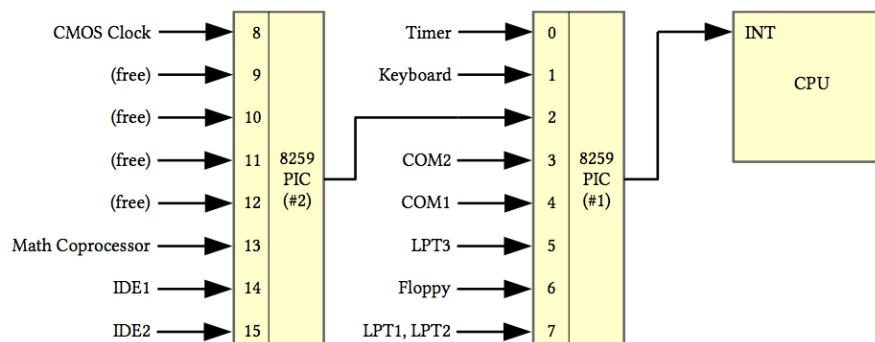
Ulix: Interrupts

Interrupts: Bedarf bei Ulix

- Timer → u. a. Aufruf des Schedulers (Prozess- bzw. Thread-Wechsel), Hochzählen der Systemzeit
- Tastatur
- Festplatten- und Floppy-Controller
- serielle Schnittstelle
- Quellcode nachlesen im PDF-Dokument *ulix-interrupt-listings.pdf* (Webseite)
- Ziel: Komplexität einer (einfachen) Beispiel-Implementierung sehen (*nicht*: auswendig lernen...)

Kaskadierte PICs [= Folie 30]

- 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
```

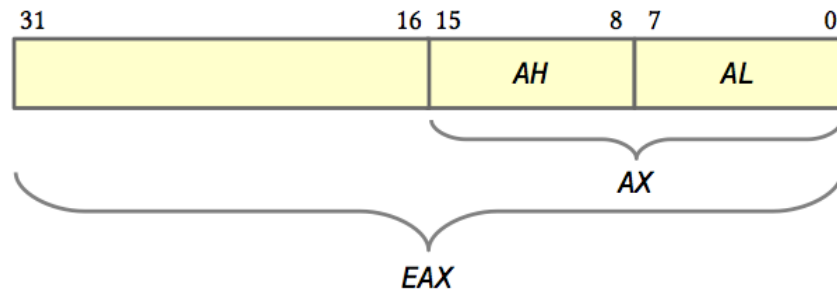
später: Interrupt-Handler für diese Interrupt-Nummern einrichten

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)



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-69

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 }
40
41
42 void
43 outportb (word port, byte data) {
44     asm volatile ("outb %al, %dx" : : "d" (port), "a" (data));
45 }
46
47
```

analog: inportw, outportw (mit inw, outw)

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-70

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
```

17.05.2016

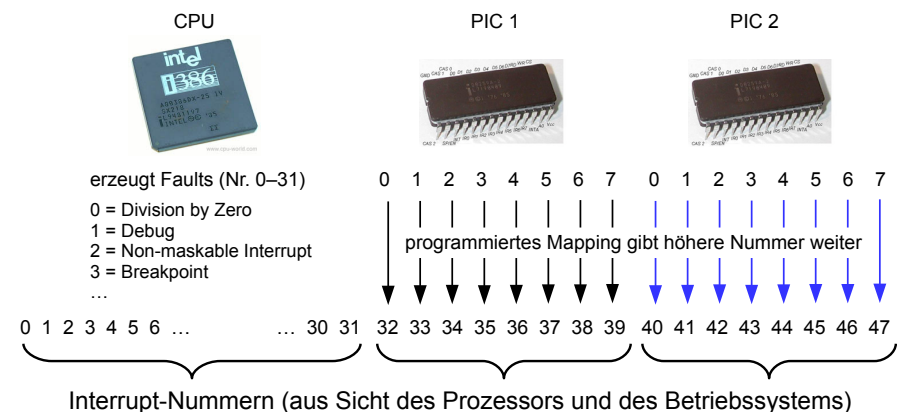
Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-71

PIC programmieren [= Folie 31]

Interrupt-Nummern remappen

Bildquellen:
siehe Folie 29



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-72

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);
```

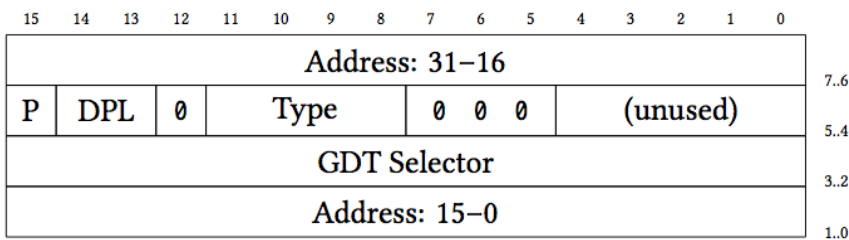
Interrupt-Handler

- 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 gdt_sel : 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));
```



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-77

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;
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-79

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));
```

Interrupt Descriptor füllen (1)

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

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-78

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-80

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 gdt sel: 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 gdt sel, 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].gdt sel = gdt sel; // GDT sel.: user or kernel mode?
79         idt[num].zeroes = 0;
80         idt[num].flags = flags;
81         idt[num].type = type;
82     }
83 }
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-81

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 };
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-82

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 gdt sel=0x08: Kernel Mode

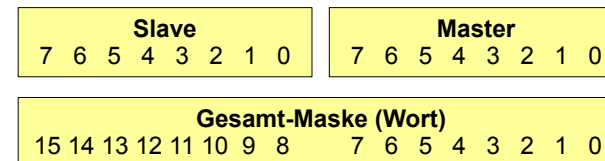
17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-83

Interrupt-Maske setzen und lesen

- set_irq_mask() – Setzen der Interrupt-Maske
→ irq.c, 92–96 (über outportb auf Data-Ports)
- get_irq_mask() – Auslesen der Maske
→ irq.c, 104–108 (inportb aus Data-Ports)
- Maske ist ein 16-Bit-Wort (1 Bit pro Interrupt)



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-84

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)

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-85

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

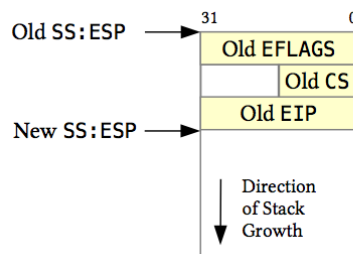
17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

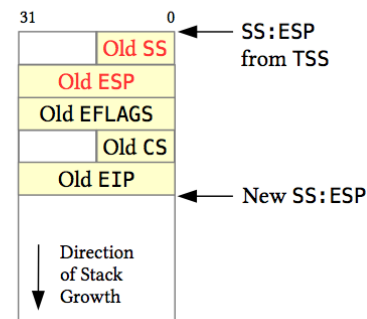
Folie C-87

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



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-86

Prozess-Kontext

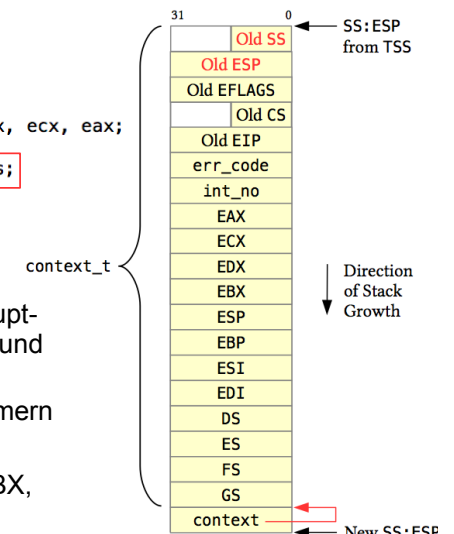
ulix.h (624-629)

```
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)



17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-88

Bsp. für kompletten Handler (IRQ 15)

```
1 push byte 0           ; error code
2 push byte 15          ; interrupt number
3 pusha                 (pusha / popa: EAX,
4 push ds               ECX, EDX, EBX, ESP,
5 push es               EBP, ESI, EDI)
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
```

Diesen Code bräuchten wir jetzt 16 mal ...

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
```

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
```

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)

```
191 // irq_handler() is the generic interrupt handler.          irq.c (206-218)
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 context_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 }
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-93

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
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-95

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
```

17.05.2016

Betriebssysteme 1, SS 2016, Hans-Georg Eßer

Folie C-94