

Betriebssysteme 1

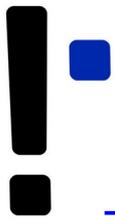
SS 2020

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

Foliensatz E:

v2.0, 2020/06/04

- Geräte
- Interrupts und Faults
- System Calls



Motivation: Tastatur am PC

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?

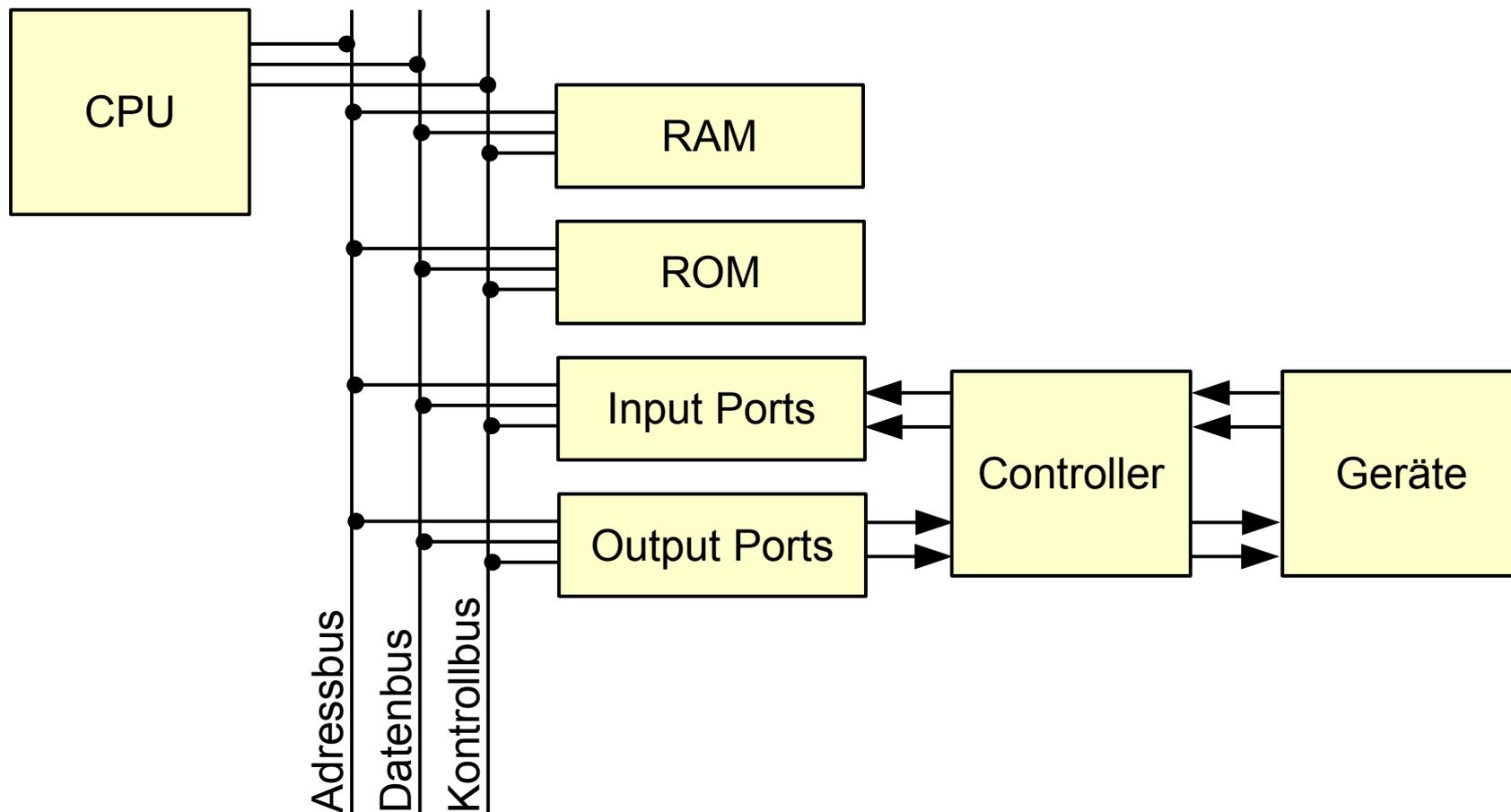


IBM Model M

Quelle: https://commons.wikimedia.org/wiki/File:IBM_Model_M.png

Motivation: Tastatur am PC (2)

- Klassisch: Kommunikation über **Ports**



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)

- „Events“ (Tastendruck oder Loslassen) generieren Scancodes
- Beispiel, Taste „A“:
 - Drücken = Scancode 30
 - Loslassen = Scancode 30 + 128 = 158
- Scancodes über Datenport (0x60) des Keyboard-Controllers auslesen:

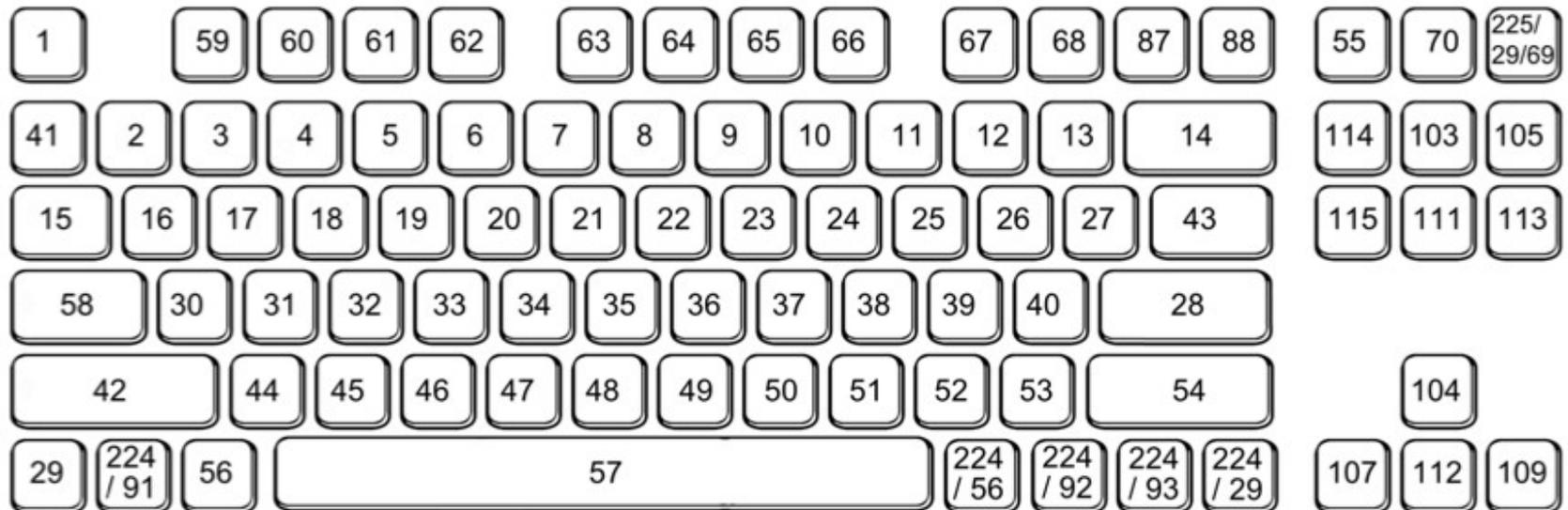
```
char scancode = inb (KBD_DATA_PORT);
```
- Statusport verrät, ob Event vorliegt

Motivation: Tastatur am PC (5)

Belegung
(US-englisch)



Scancodes
(„key press“)



Motivation: Tastatur am PC (6)

- Simpler Treiber (Pseudocode, vereinfacht)

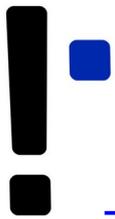
```
do forever {
    // warte auf Event
    while (inb (KBD_STATUS_PORT) != NEW_EVENT) ;

    // 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)

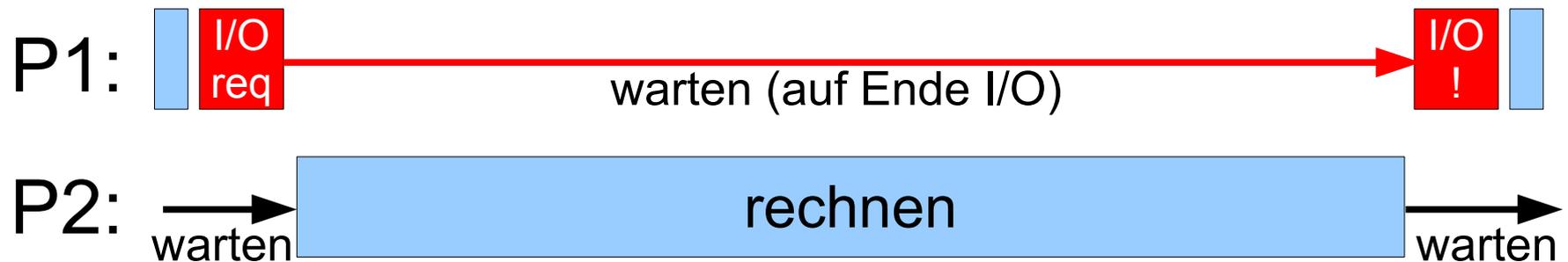
- Naiver Ansatz heißt „Pollen“: in Dauerschleife ständig wiederholte Geräteabfrage
- Pollen verbraucht sehr viel Rechenzeit:



- 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

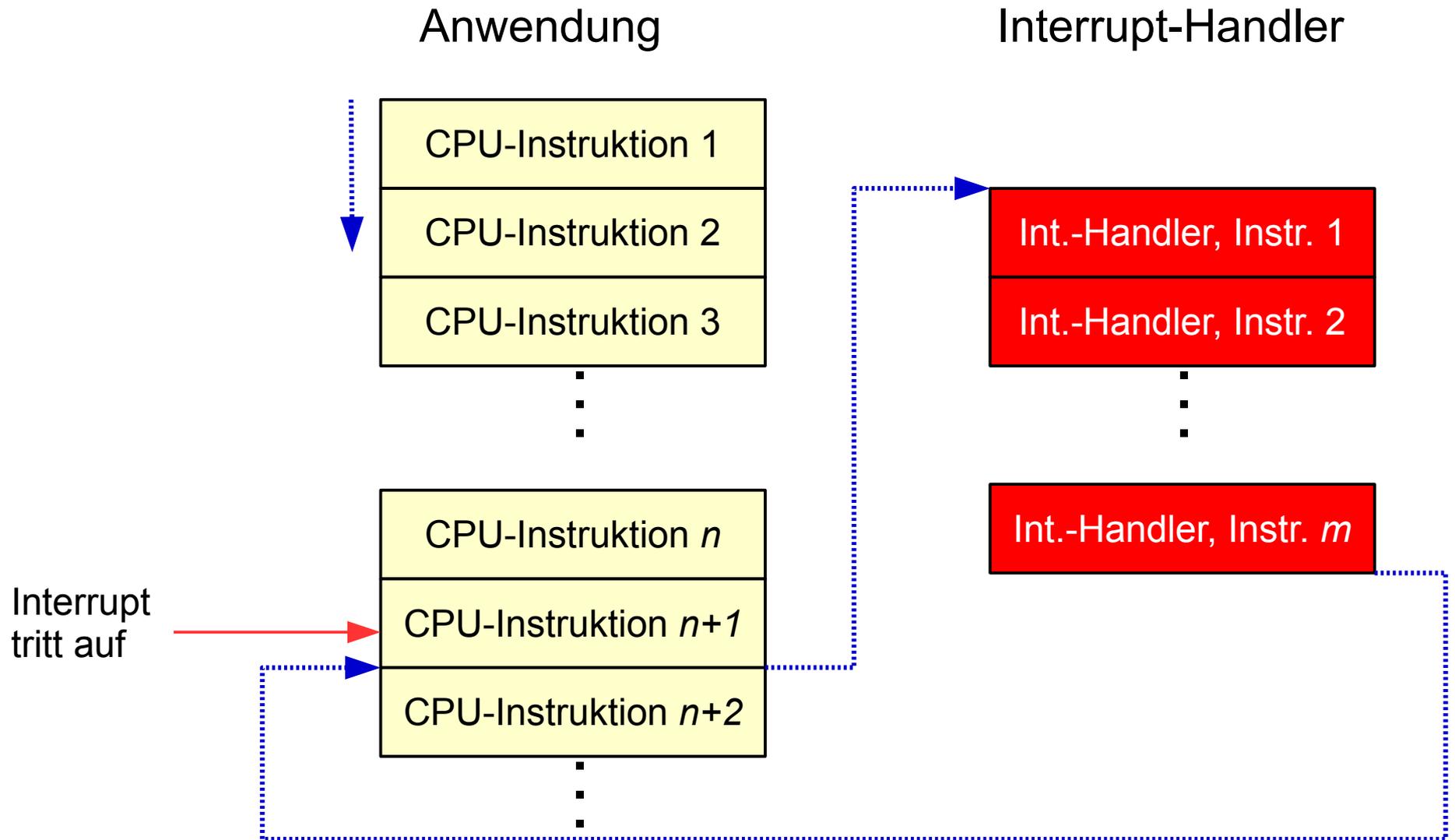
Vorteile

- 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

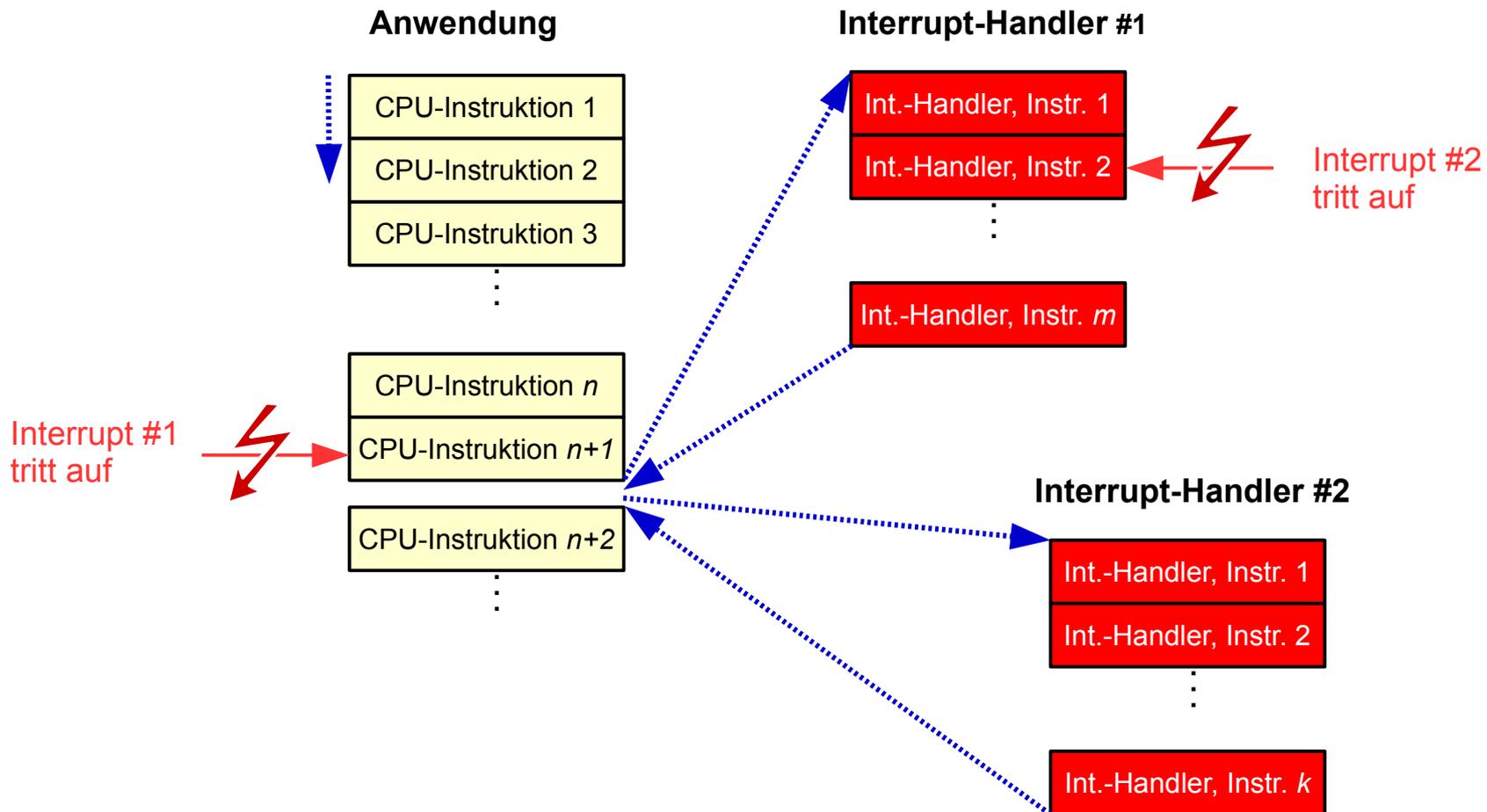
Mehrfach-Interrupts (1)

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



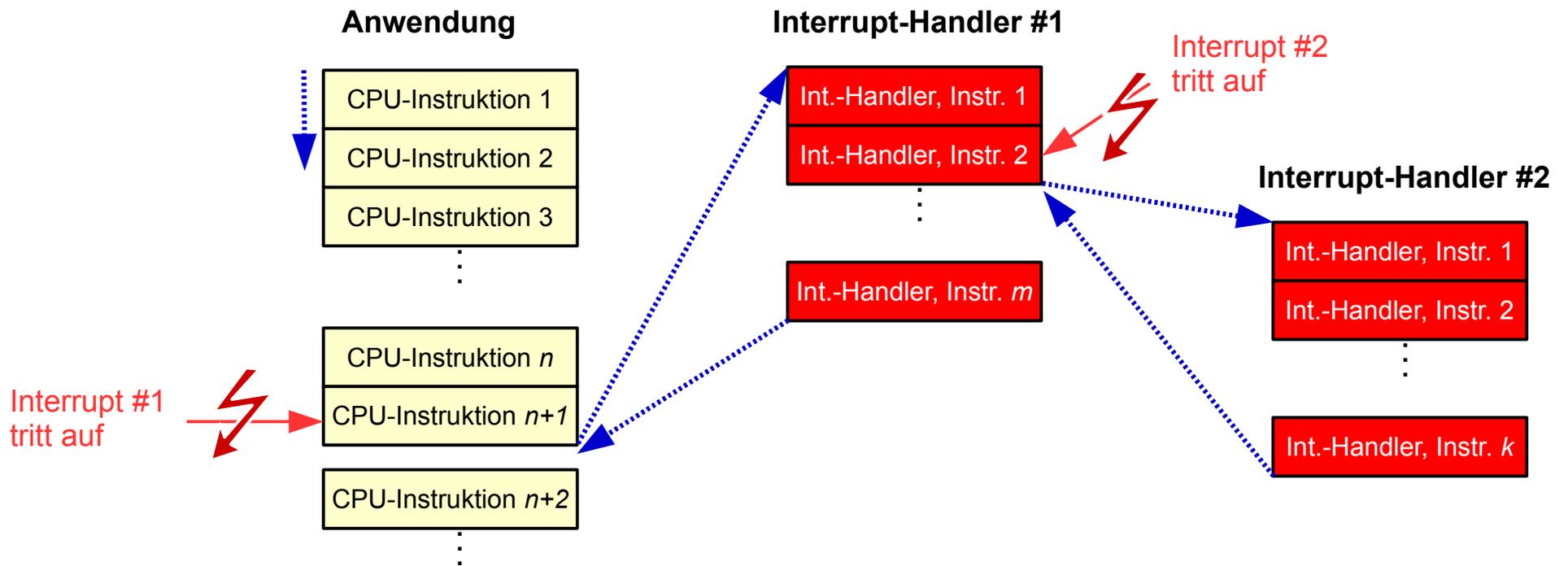
Mehrfach-Interrupts (3)

Variante 2

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

Mehrfach-Interrupts (4)

Variante 2



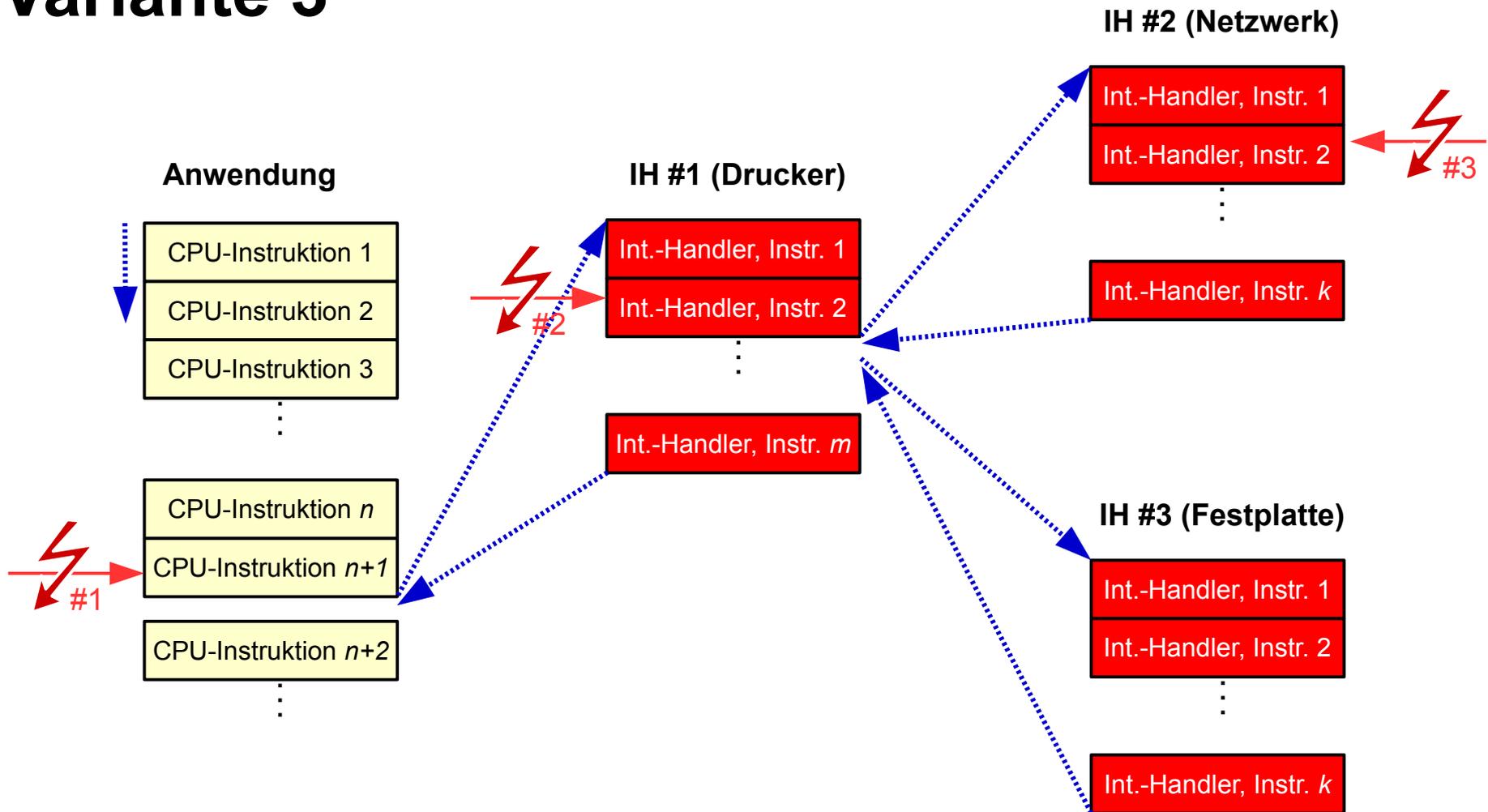
Mehrfach-Interrupts (5)

Variante 3

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

Mehrfach-Interrupts (6)

Variante 3



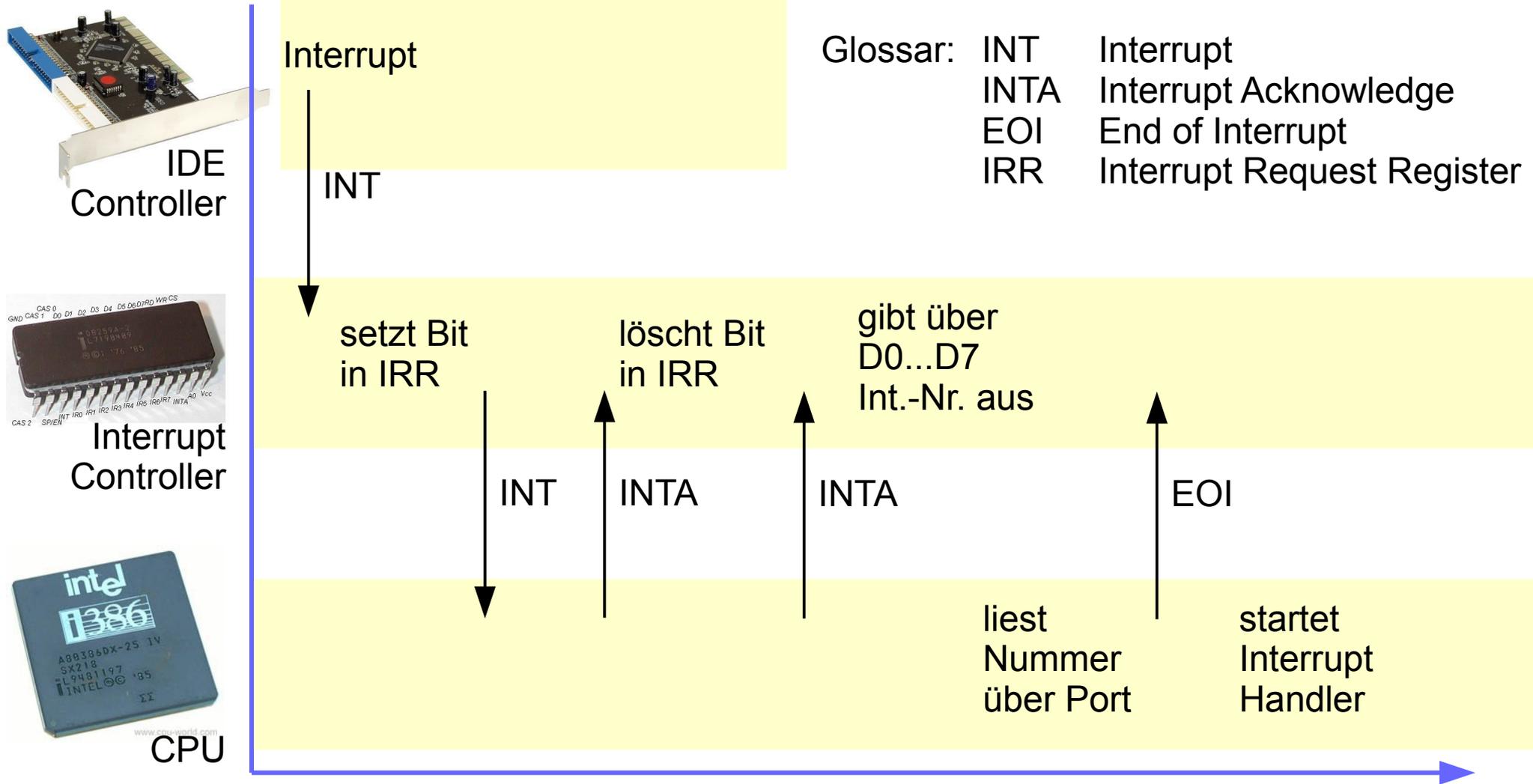
Mehrfach-Interrupts (7)

- Problem bei gesperrten Interrupts: Behandlung muss schnell erfolgen
- Lösung: Aufteilung des Interrupt-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

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

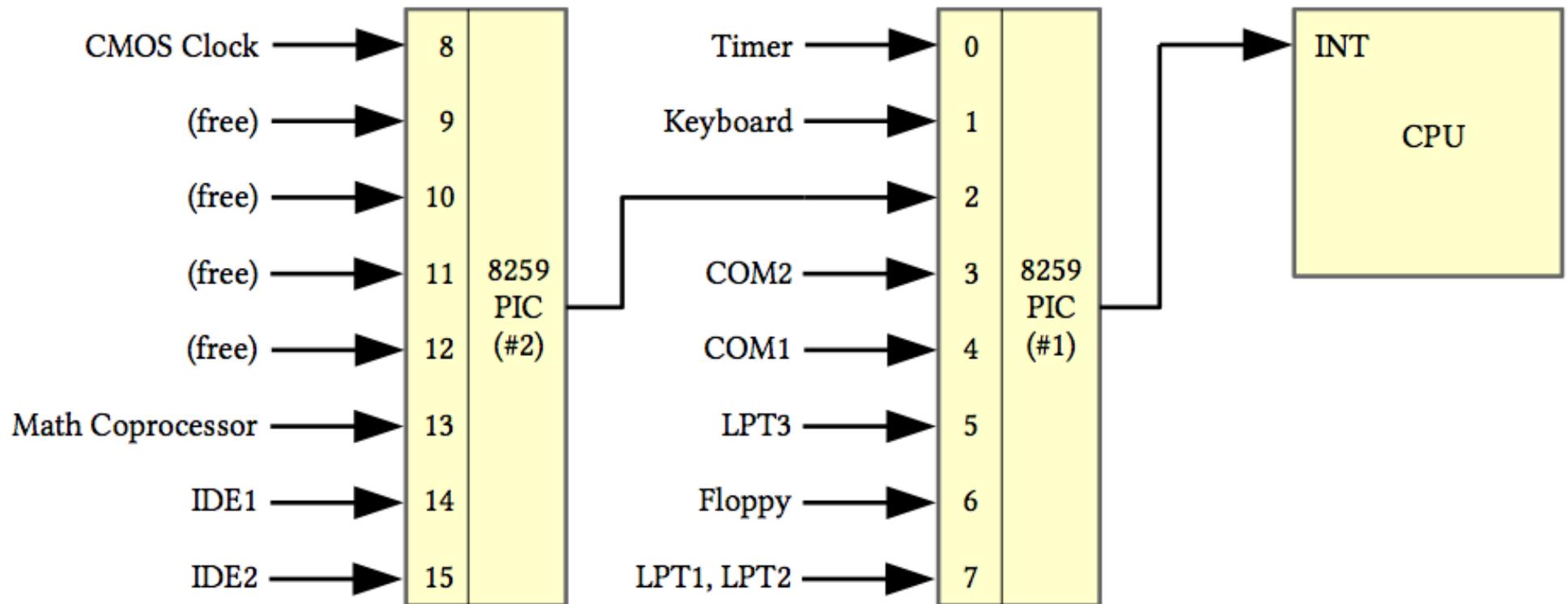
Interrupts unterscheiden (2)



Bildquellen: IDE Controller: http://www.ebay.de/usr/sm-pc_8259A: <http://www.brokenhorn.com/Resources/OSDevPic.html>, i386: <http://www.cpu-world.com/CPU/80386/Intel-A80386DX-25%20IV.html>

Interrupts unterscheiden (3)

- PCs: Klassischer PIC (**P**rogrammable **I**nterrupt **C**ontroller) Intel 8259 hat nur acht Eingänge
→ Kaskadierung von zwei PICs



Interrupts unterscheiden (4)

Intel 8259 ist programmierbar. Ziel:

Bildquellen:
siehe Folie 29

CPU



PIC 1



PIC 2



erzeugt Faults (Nr. 0–31)

0 = Division by Zero

1 = Debug

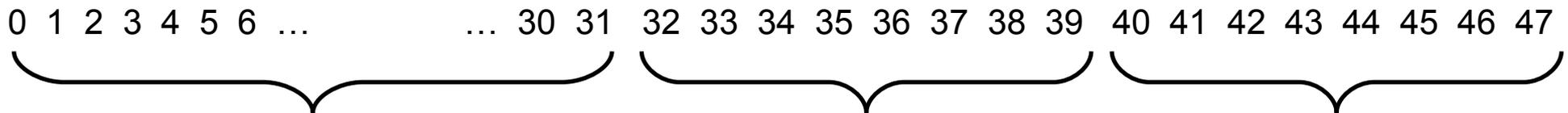
2 = Non-maskable Interrupt

3 = Breakpoint

...

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

programmiertes Mapping gibt höhere Nummer weiter



Interrupt-Nummern (aus Sicht des Prozessors und des Betriebssystems)

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.“

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**

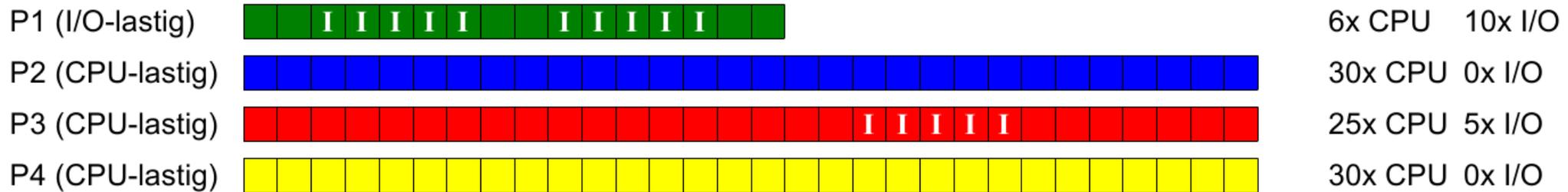
- Prozess führt viele I/O-Operationen durch und benötigt vergleichsweise wenig Rechenzeit
- Sehr kurze Rechenphasen wechseln sich mit häufigen Wartezeiten auf I/O ab

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

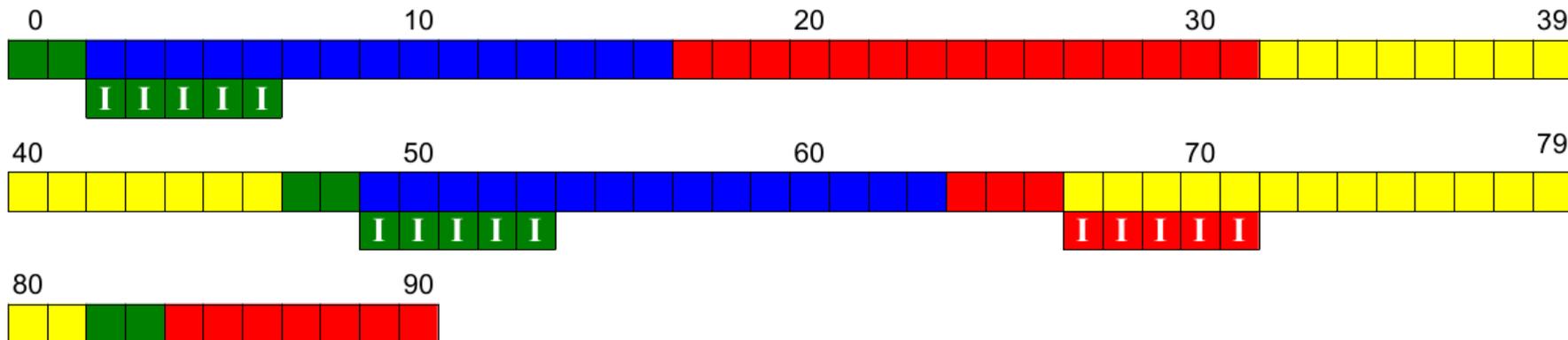
Multitasking und Interrupts

- 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

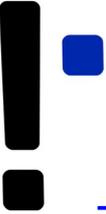
I/O-lastig vs. CPU-lastig (3)



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
```

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-fastedge   acpi
12:     3070145           16539          16542          16485   IO-APIC-edge       i8042
16:     2760924             881            904            886   IO-APIC-fastedge   uhci_hcd:usb1, nvidia
18:    24122388           6538           6698           6647   IO-APIC-fastedge   ehci_hcd:usb6, uhci_hcd:usb7
19:         281             28             27            10   IO-APIC-fastedge   uhci_hcd:usb3, uhci_hcd:usb5
21:     22790              0              0              0   IO-APIC-fastedge   uhci_hcd:usb2
22:     7786588          10464141         8251870         8439964   IO-APIC-fastedge   HDA Intel
23:         899              0              1              1   IO-APIC-fastedge   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 shutdowns
TRM:            0              0              0              0   Thermal event interrupts
SPU:            0              0              0              0   Spurious interrupts
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“

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)

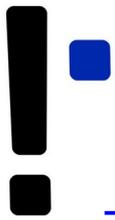
Tasklet (bottom half)

- 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)

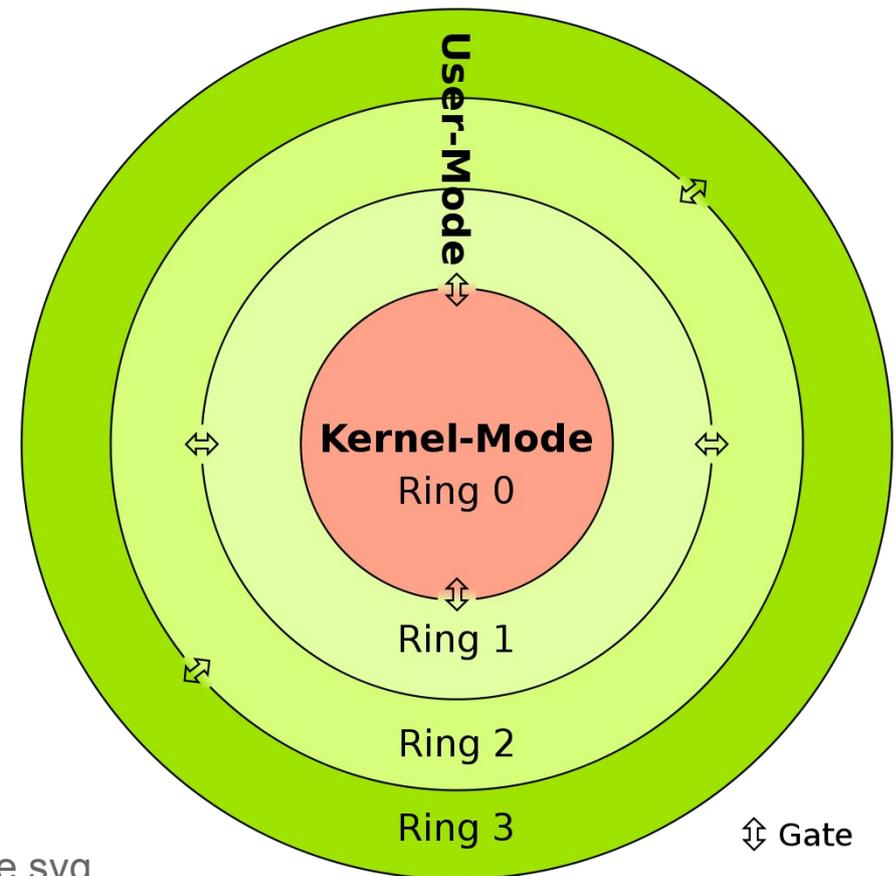
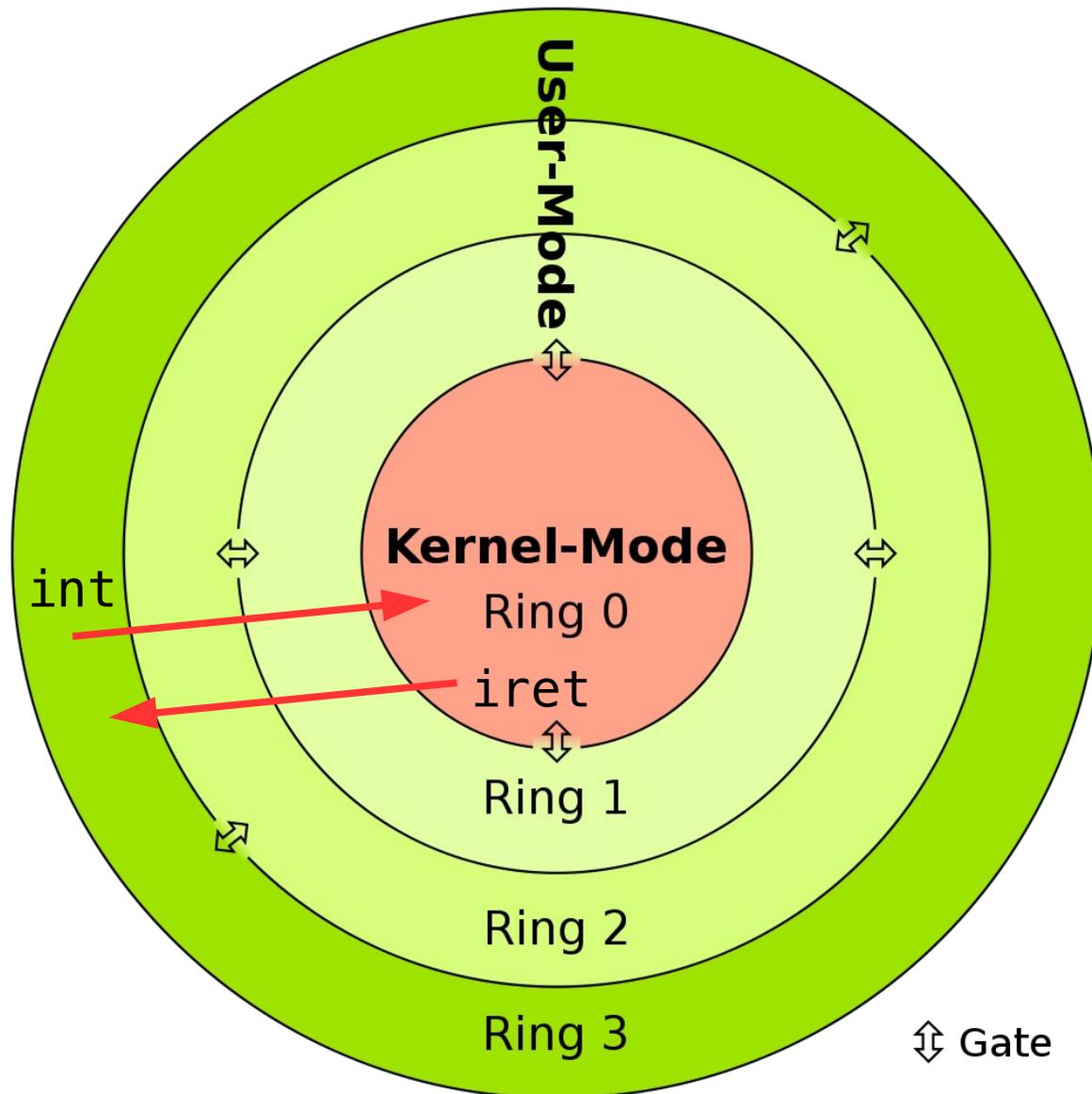


Bild: http://commons.wikimedia.org/wiki/File:CPU_ring_scheme.svg
(Autor: <http://commons.wikimedia.org/wiki/User:Sven>; GNU Free Documentation License)

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 kompatible)
- 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: `sys leave` (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

Beispiel für *Anwendung* (Linux)

Ausgabe auf stdout

```
_start:                ; fuer Linker (wo gehts los)
    mov     edx, len    ; Nachrichtenlaenge
    mov     ecx, msg    ; Adresse der Nachricht
    mov     ebx, 1      ; file descriptor (1=stdout)
    mov     eax, 4      ; Syscall-Nr. (sys_write)
    int     0x80        ; Syscall ausfuehren

    mov     eax, 1      ; Syscall-Nr. (sys_exit)
    int     0x80        ; Syscall ausfuehren

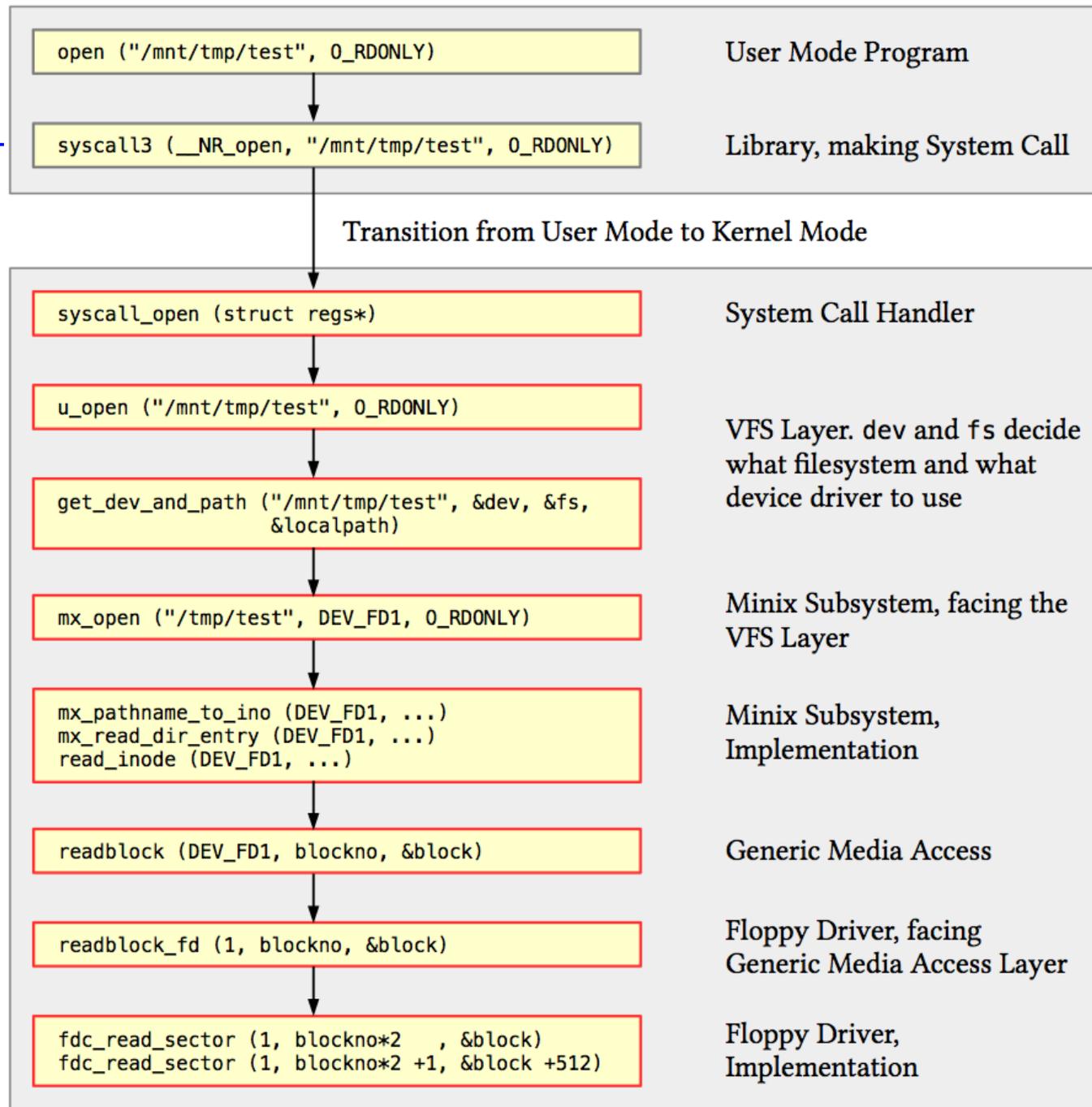
msg db    'Hallo Welt!',
        0xa            ; Text
len equ   $ - msg      ; Laenge
```

Beispiel *Implementation*

Dateizugriff im Lehr-BS Ulix (siehe Ulix-Buch, S. 407)

`syscall3()`

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



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_fstime             35
#define __NR_sync                36
#define __NR_kill                37
...
```