

Betriebssysteme

SS 2015

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

Foliensatz D:

v1.2, 2015/05/03

- Interrupts und Faults
- System Calls

Übersicht: BS Praxis und BS Theorie

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

Folien D

Interrupts

- 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

- Naiver Ansatz für Plattenzugriff:

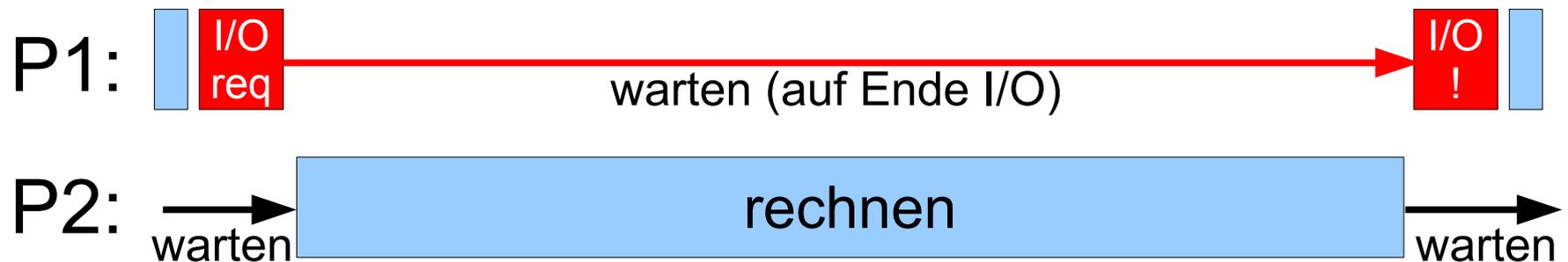
```
naiv () {  
    rechne (500 ZE);  
    sende_anfrage_an (disk);  
    antwort = false;  
    while ( ! antwort ) {  
        /* diese Schleife rechnet 1.000.000 ZE lang */  
        antwort = test_ob_fertig (disk);  
    }  
    rechne (500 ZE);  
    return 0;  
}
```

- 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

- 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?

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

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

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)

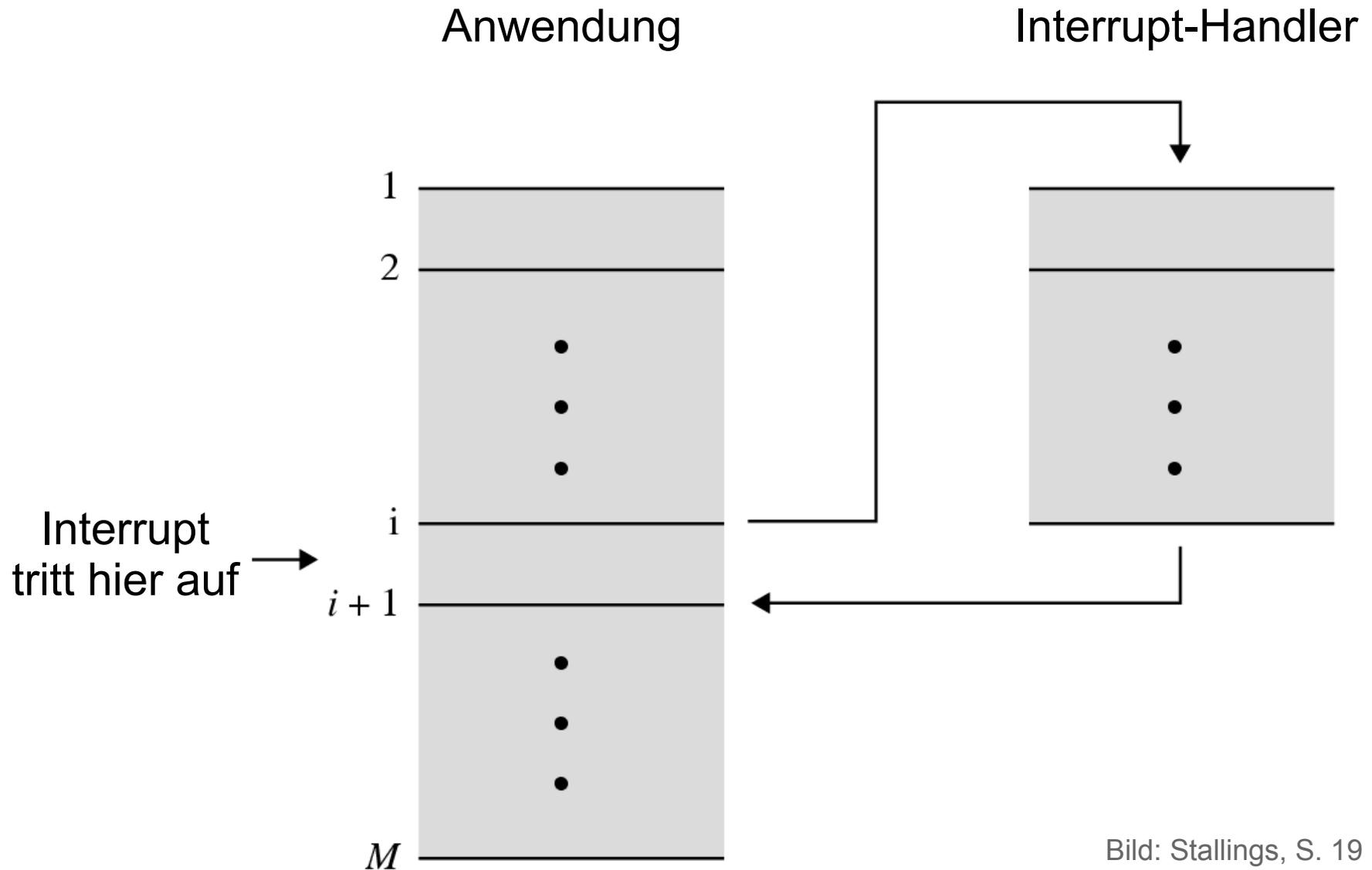


Bild: Stallings, S. 19

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)

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)

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

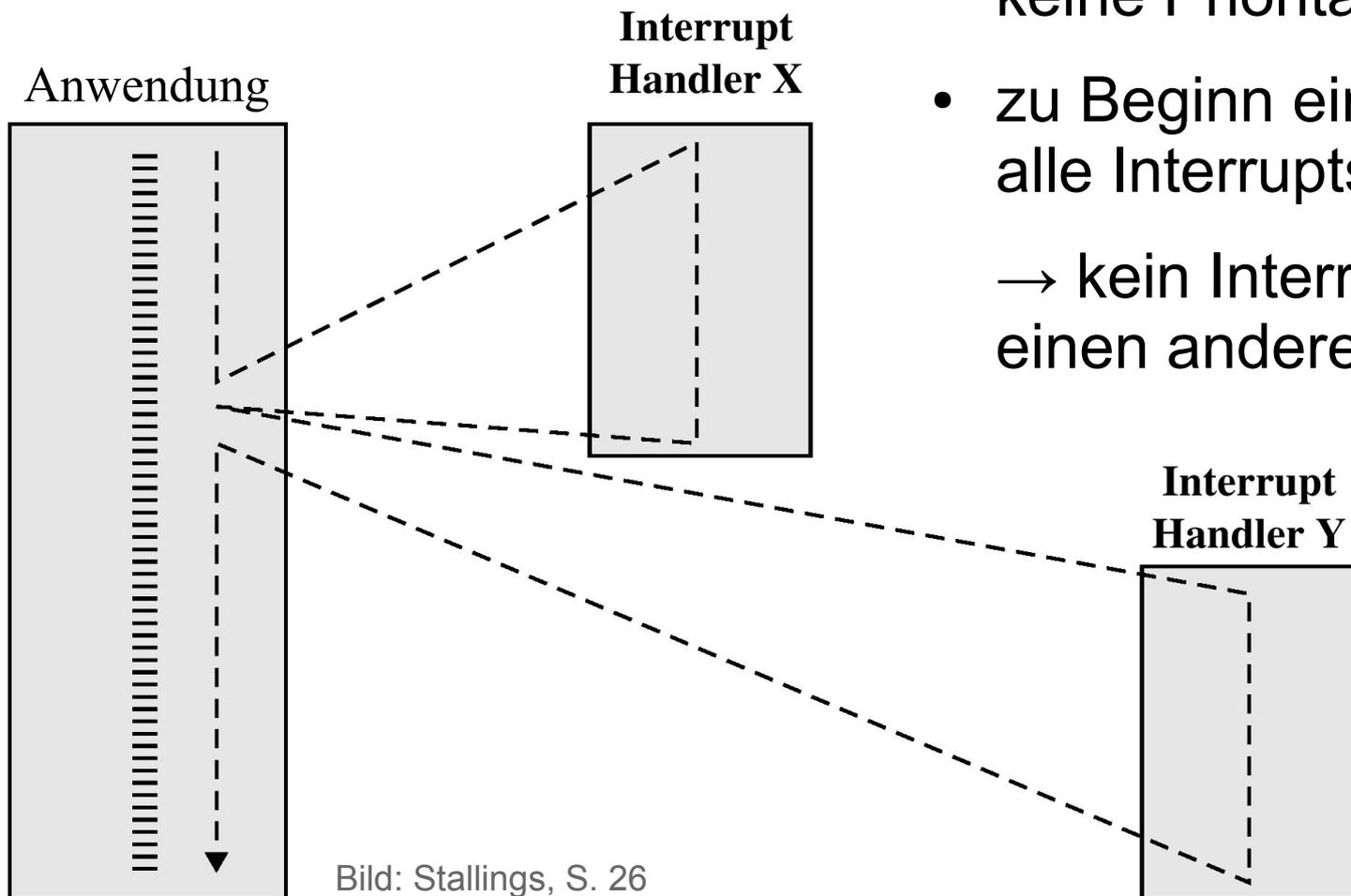


Bild: Stallings, S. 26

Mehrfach-Interrupts (2)

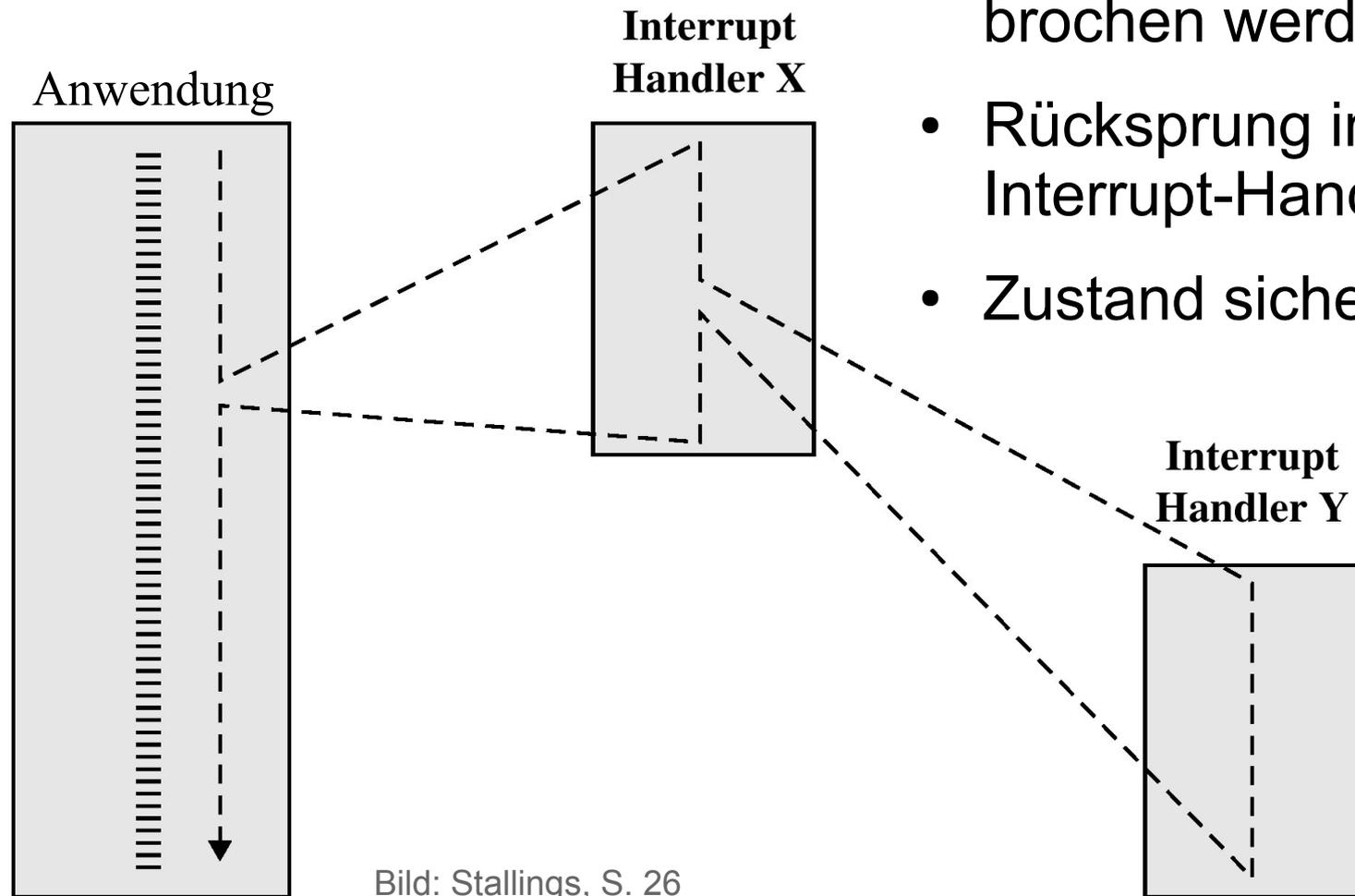


Bild: Stallings, S. 26

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

Mehrfach-Interrupts (3)

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

Mehrfach-Interrupts (4)

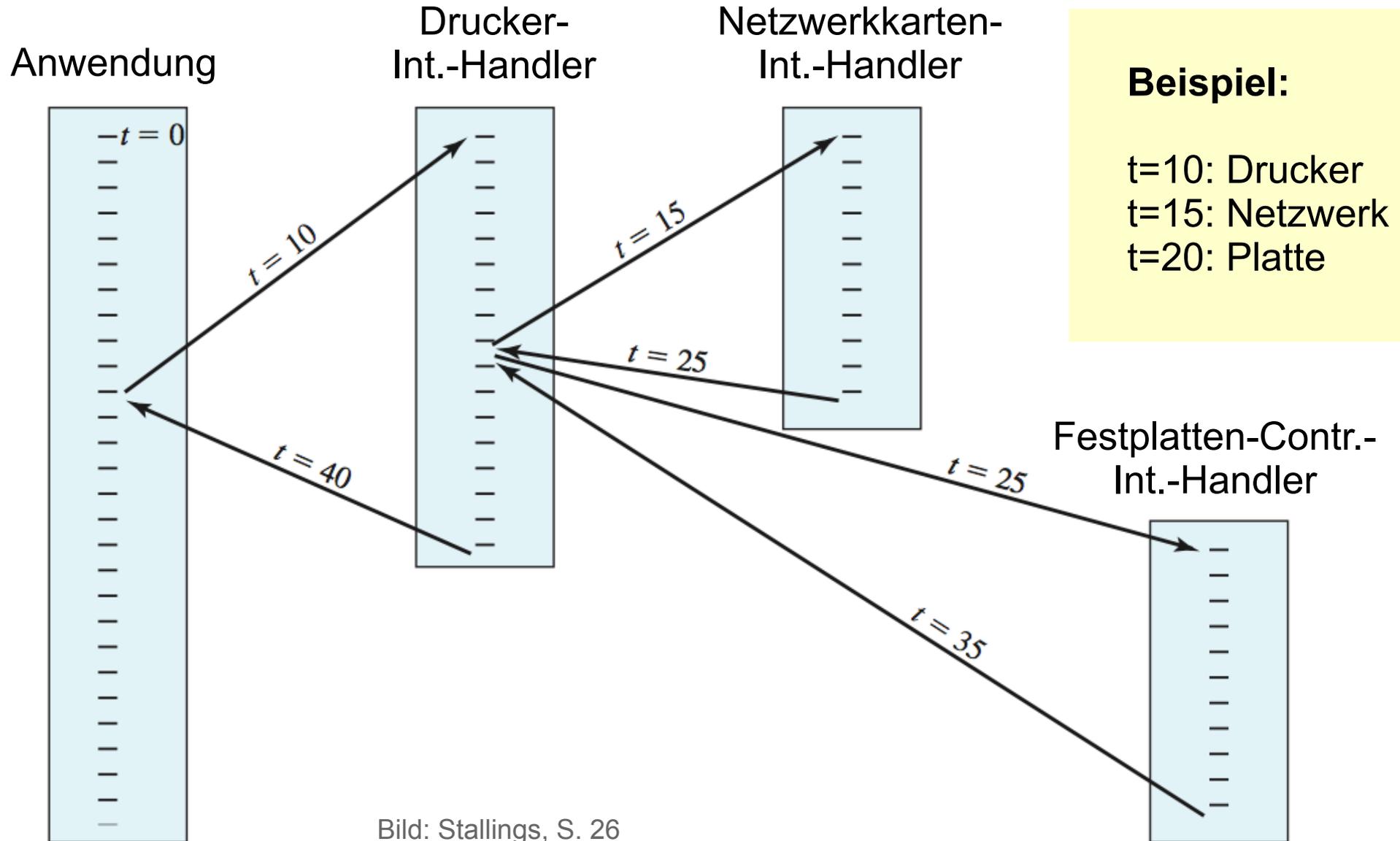


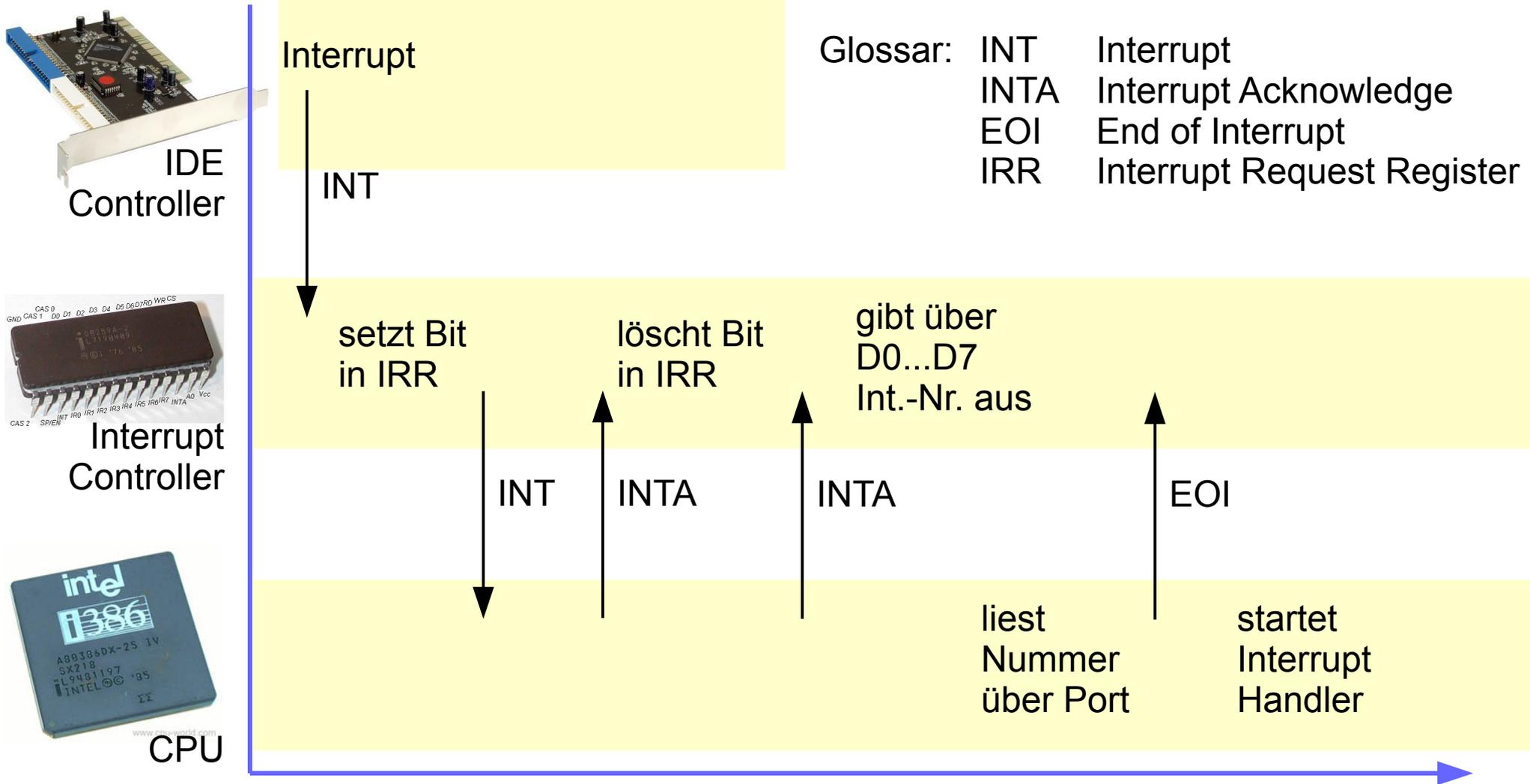
Bild: Stallings, S. 26

- 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

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

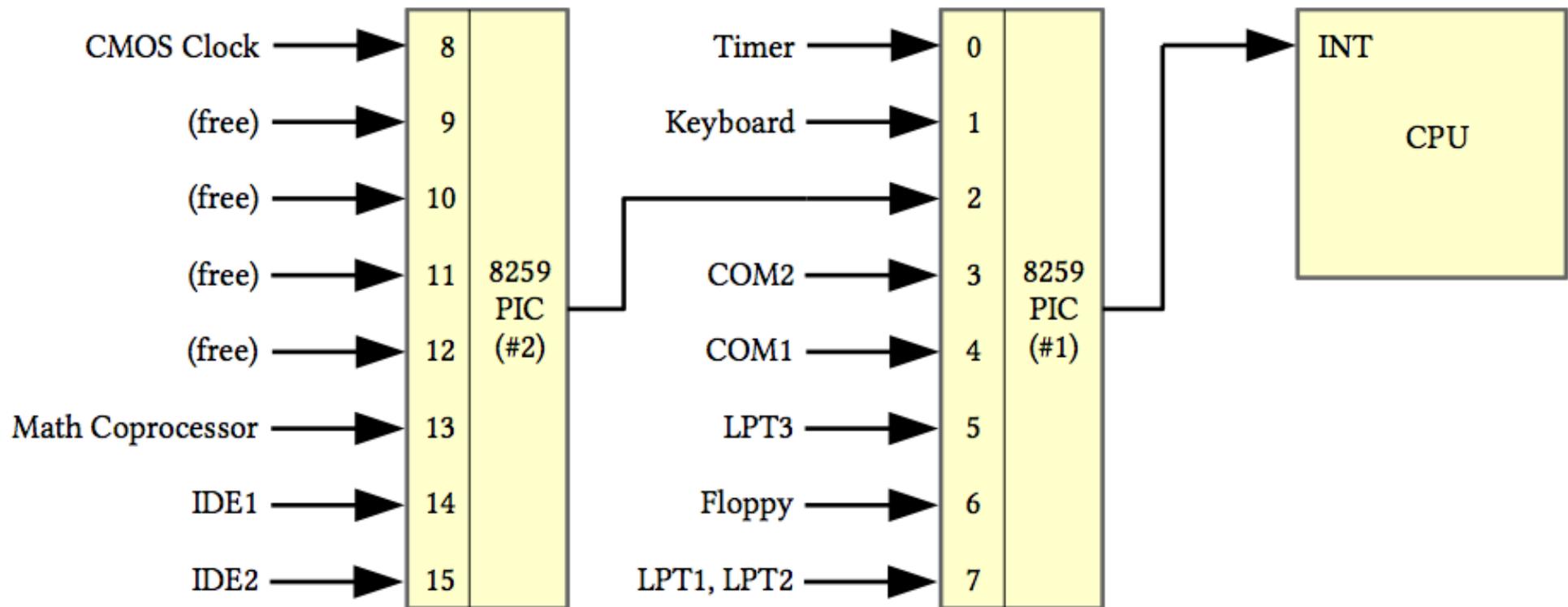
Interrupts unterscheiden (2)



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 20

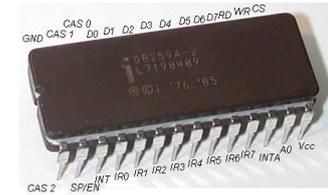
CPU



PIC 1

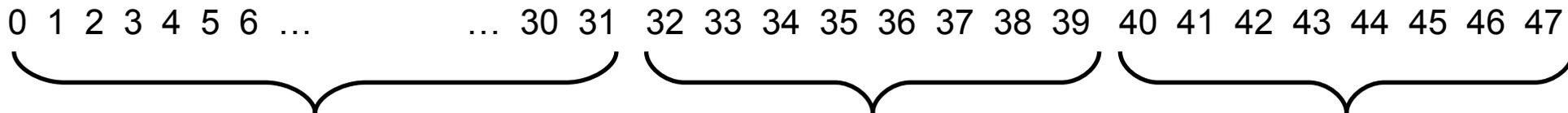
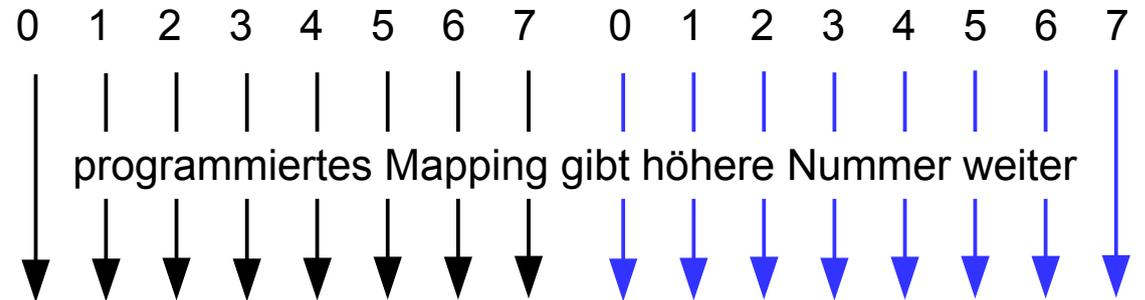


PIC 2



erzeugt Faults (Nr. 0–31)

- 0 = Division by Zero
- 1 = Debug
- 2 = Non-maskable Interrupt
- 3 = Breakpoint
- ...



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

- 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.“

- **CPU-lastiger Prozess**

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

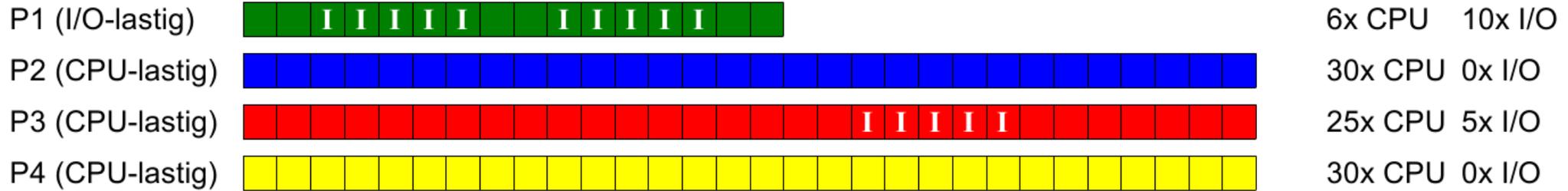
- **I/O-lastiger Prozess**

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

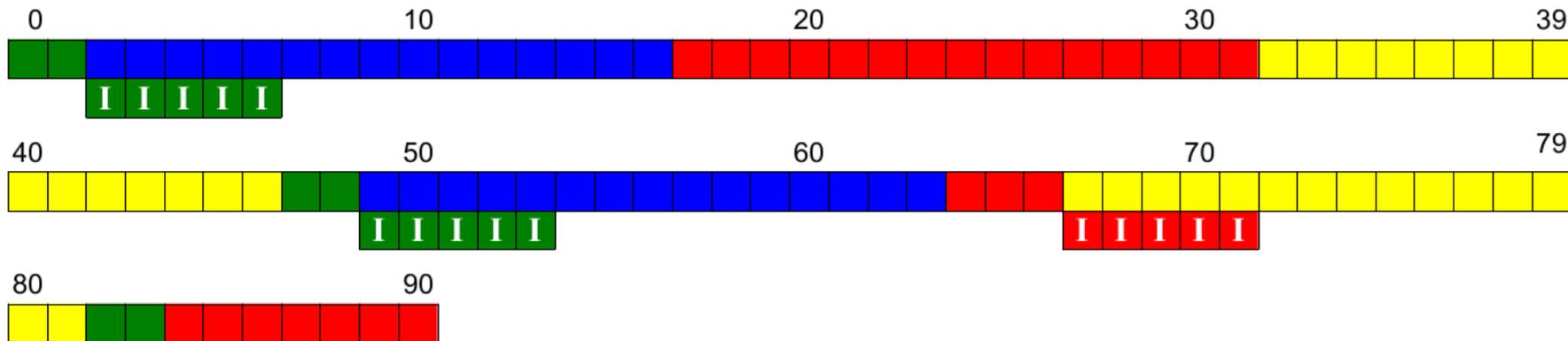
Multitasking und Interrupts

- 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

```
[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
```

```
[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
```

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“

„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

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

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/>

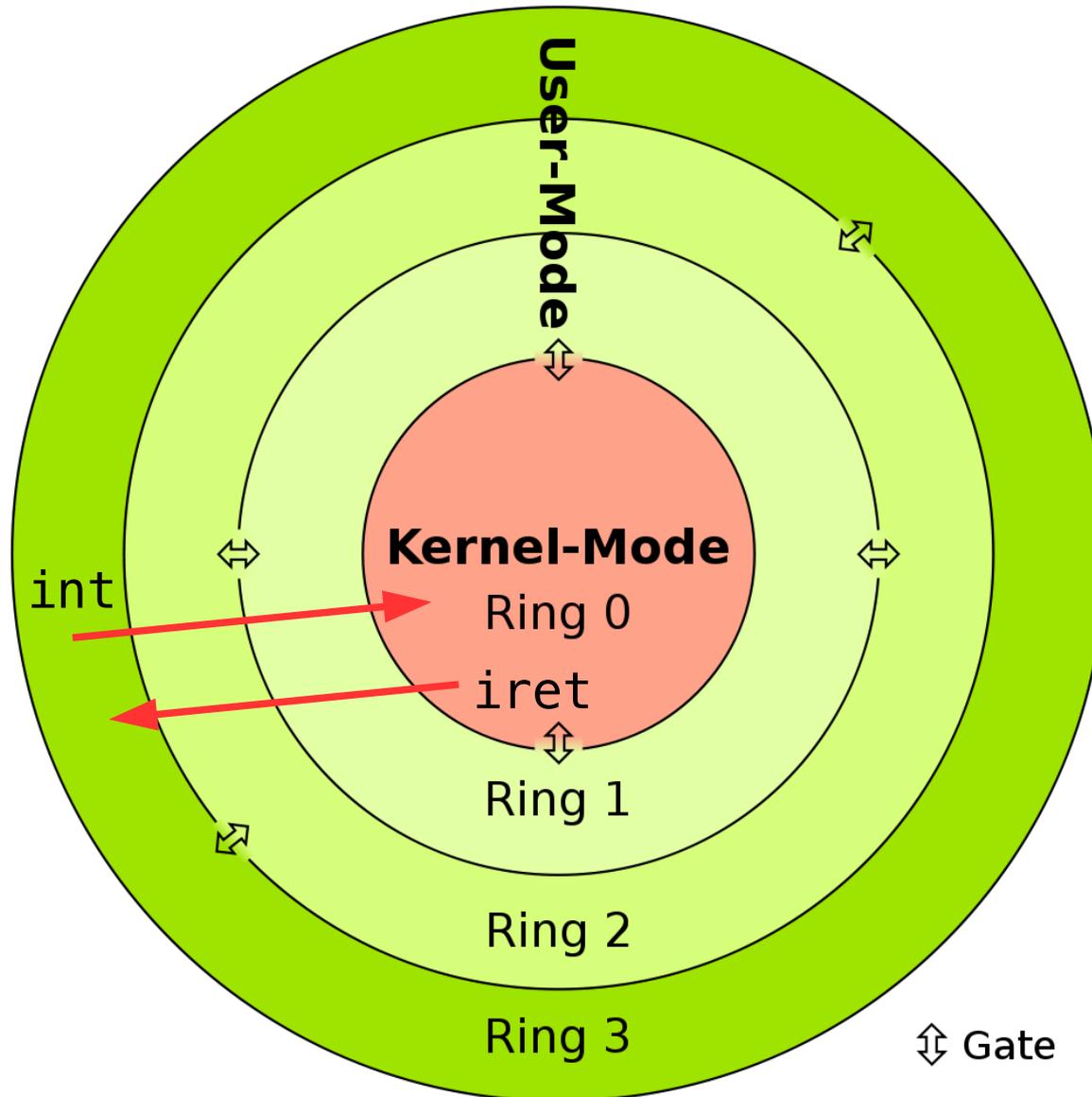
System Calls

- 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)



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 compatible)
- CPU reagiert auf `int`-Instruktion wie auf einen HW-Interrupt und ruft Interrupt-Handler auf

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

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 Verfeinerung für Anwendungs-Entwickler: **User-Mode-Bibliothek** mit Wrappern für die System Calls

- Ausgabe auf stdout

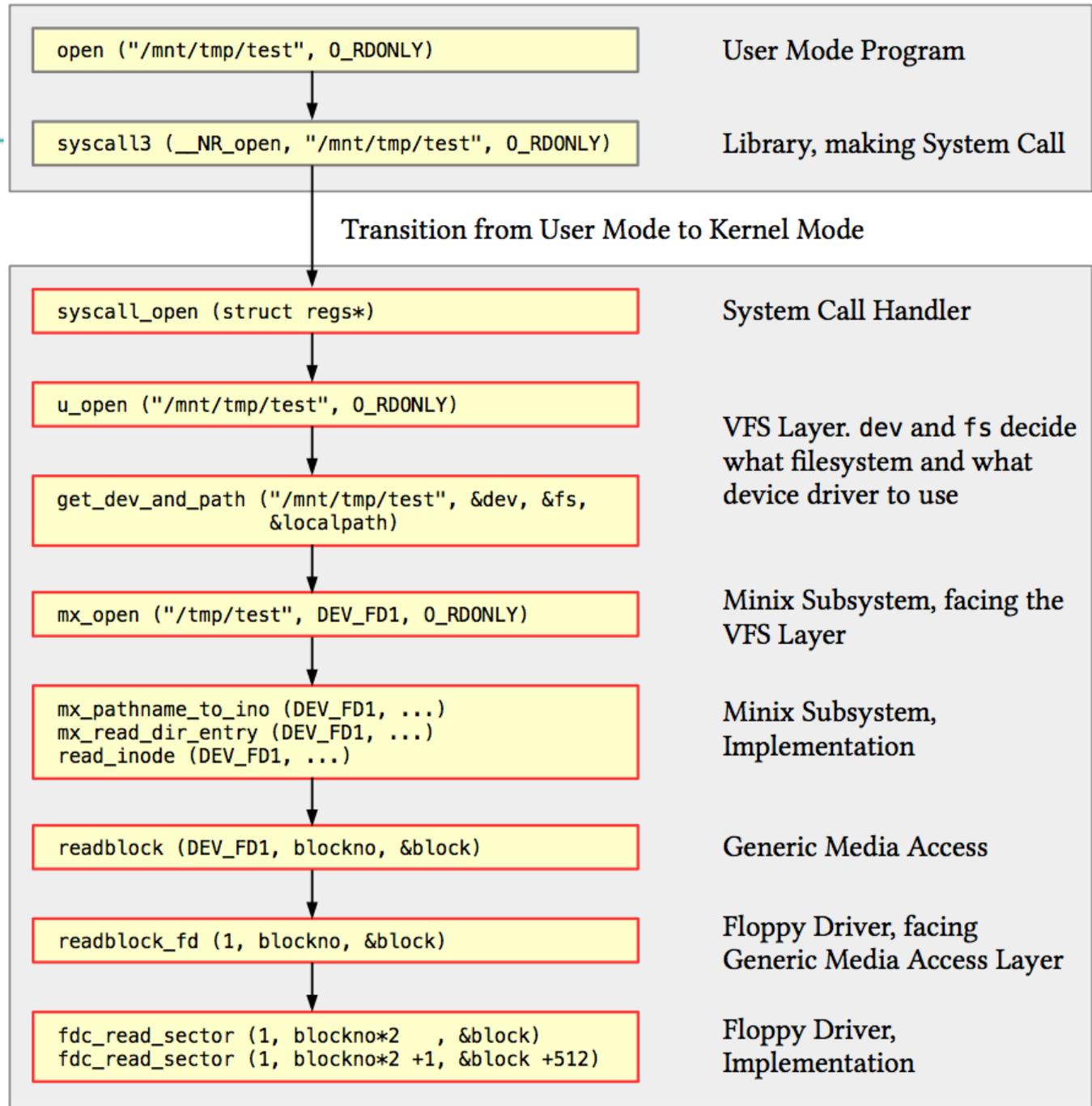
```
_start:                ; tell linker entry point
    mov edx,len        ; message length
    mov ecx,msg        ; message to write
    mov ebx,1          ; file descriptor (stdout)
    mov eax,4          ; system call number (sys_write)
    int 0x80           ; software interrupt 0x80
    mov eax,1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

Beschreibung folgt im UNIX-Teil (Foliensatz V)

`syscall3()`

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



`/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
...

```

System Calls für Programmierer:

Standardfunktionen in C

- Standardbibliotheken stellen Wrapper für System Calls bereit
- hier nur kleine Auswahl:
 - Dateizugriff: `open`, `read`, `write`, `close`
 - Prozesse: `fork`, `exit`, `exec1(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);
```

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 );
```

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;
}
```

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);  
}
```

`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);
```

`exit()` Programm beenden

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

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

```
man 3 exit
```

Beispiel:

```
exit(0);
```

`fork()` neuen Prozess starten

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

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

```
man fork
```

Beispiel:

```
pid=fork( )
```

exec ()

Anderes Programm im Prozess laden

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg, ..., char * const envp[]);
int execv(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);
```

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

- Foliensatz V:
 - Implementierung in UNIX
 - Interrupt Handler
 - Fault Handler
 - System Call Handler