# Interrupts



## Gliederung





## Interrupts

Übersicht:

- Motivation
- Interrupt-Klassen
- Interrupt-Bearbeitung
- Interrupt-Prioritäten
- Mehrfach-Interrupts
- I/O vs. CPU-lastige Prozesse
- Interrupt-Handler
- Software-Interrupts und System Calls



## Motivation (1)

Wozu Interrupts ?

- Festplattenzugriff ca. um Faktor 1.000.000 langsamer als CPU-Anweisung
- Naiver Ansatz:

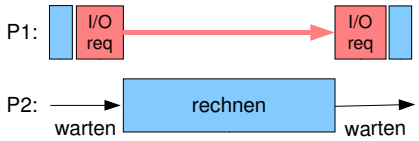
```
naiv() {
   rechne( 500 ZE );
   sende_anfrage_an( disk );
   while( ! antwort ) {
      /* Schleife rechnet 1.000.000 ZE lang */
      antwort =  test_ob_fertig( disk );
   }
   ...
}
```

- **Pollen**: in Dauerschleife wiederholte Geräteabfragen

## Motivation (2)

Pollen:

- Pollen verbraucht sehr viel Rechenzeit
- auch bei parallelen Prozessen:
  Pollen noch ungünstig
- **besser:** in der Wartezeit etwas anderes tun
- **Idee:**
  - Prozess, der I/O-Anfrage gestartet hat, unterbrechen und schlafen legen, bis Anfrage bearbeitet
  - die Zwischenzeit anderweitig nutzen
- **Frage:**
  - woher weiß System ohne Polling,
    - wann Anfrage fertig ist,
    - wann Prozess weiterarbeiten kann




## Motivation (3)

- **Lösung:**
  - Interrupts: bestimmte Ereignisse können normalen Ablauf unterbrechen
  - nach jeder CPU-Anweisung prüfen (in Hardware), ob Interrupt-Anfrage anhängig
- **Vorteile:**
  - Effizienz:
    - I/O-Zugriffe langsam -> Nutzung der Zeit für andere Prozesse
  - Programmlogik:
    - kein Pollen, sondern Geräte signalisieren Hardware-basiert den Status
- Nachteile:
  - Mehraufwand:
    - Kommunikation mit Hardware komplexer
    - Instruction-Cycle enthält zusätzlichen Schritt



## Interrupt-Klassen

Welche Interrupt-Klassen werden unterschieden ?

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



## Interrupt-Bearbeitung (1)

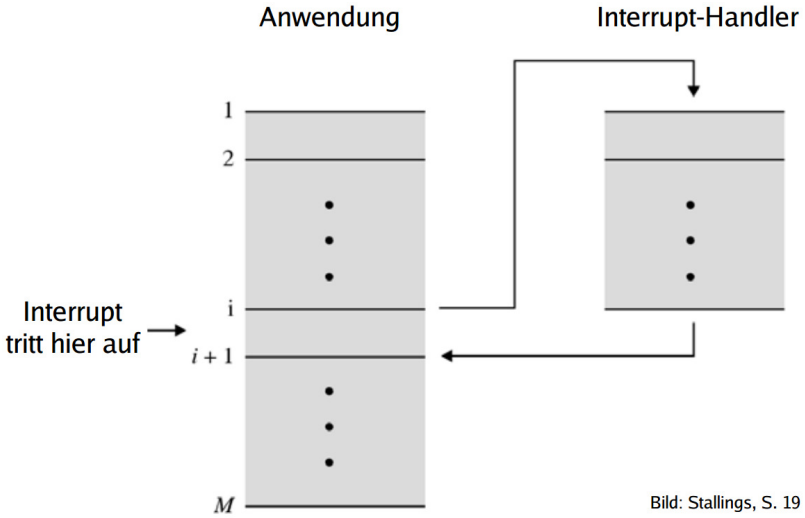



Bild: Stallings, S. 19

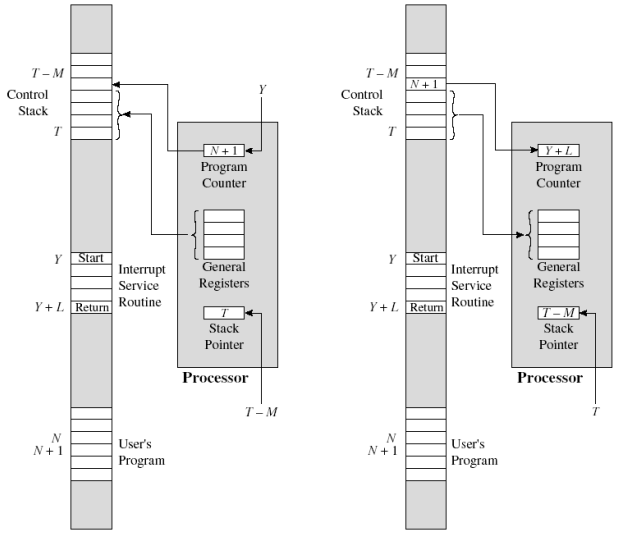
## Interrupt-Bearbeitung (2)

Grundsätzlicher Ablauf:

- Interrupt tritt auf
  - Laufender Befehl wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
  - BS speichert Daten des Prozesses (Kontext-Wechsel) (wie bei Prozess-Wechels -> Scheduler)
  - BS ruft hinterlegten **Interrupt-Handler** auf
    - Interrupt-Vektor (Adresse des Int.Handlers) zusammengesetzt aus
      - I-Register: oberer Teil der Adresse der Interrupt-Handler-Tabelle
      - Adreßteil vom Peripheriegerät (unterer Teil)
  - danach: Scheduler wählt Prozess aus, der weiterarbeiten darf (z.B. den unterbrochenen)



## Interrupt-Bearbeitung (3)






## Interrupt-Prioritäten

Was tun bei Mehrfach-Interrupts ?

**Drei Möglichkeiten:**

1. Während Interrupt andere sperren (andere „maskieren“, DI = disable interrupts) --> Warteschlange
2. Während Interrupts andere zulassen
3. **Interrupt-Prioritäten**:
   a. nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer
   b. CPU vermerkt Priorität in Interrupt-Statusregister

Interrupt-Prioritäten:

**Beispiele:**

| | |
|---|---|
| Power Failure | ↑ höher |
| Systemuhr | |
| Netzwerk-I/O | |
| Festplatten-I/O | |
| Terminals | |
| Software-Interrupts | ↓ geringer |



## Mehrfach-Interrupts

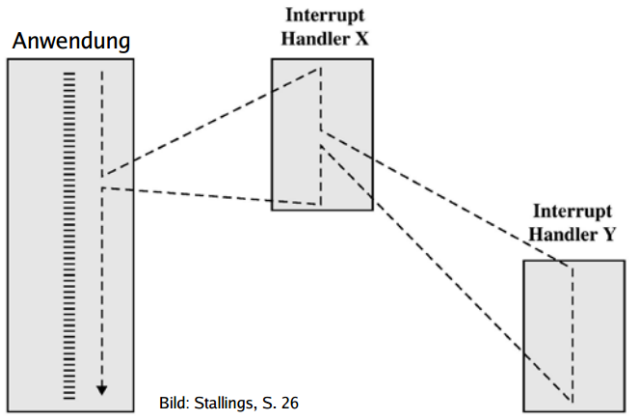



Bild: Stallings, S. 26

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

Welchen Einfluß haben Interrupts auf unterschiedliche Prozesstypen ?

- **CPU-lastiger Prozess**
  - benötigt überwiegend CPU-Rechenzeit und vergleichsweise wenig I/O-Operationen
  - nach längeren Rechenphasen nur gelegentlich durch I/O-Wartezeiten unterbrochen
- **I/O-lastiger Prozess:**
  - führt viele I/O-Operationen durch und benötigt vergleichsweise wenig Rechenzeit
  - kurze Rechenphasen wechseln sich mit häufigen I/O-Wartezeiten ab



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

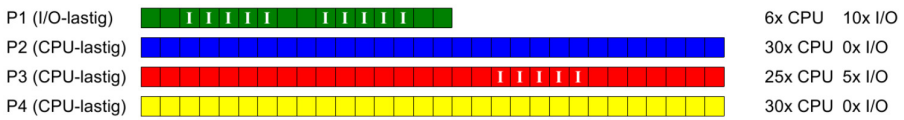
Welchen Einfluß haben Interrupts auf Prozesse mit unterschiedlichen Anforderungen ?

**Multitasking und Interrupts**

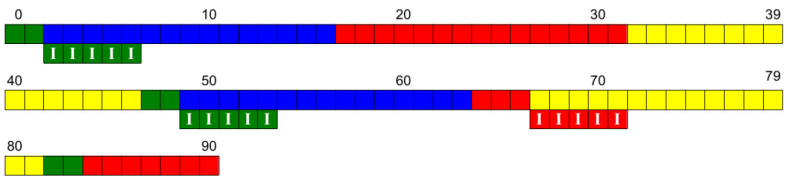
- Multitasking verbessert CPU-Nutzung:
  - I/O-lastiger Prozess wartet auf I/O-Events
  - CPU-lastiger Prozess rechnet weiter
- Prozess stößt I/O-Operation an und legt sich schlafen (wartet auf Signal)
- optimale Performance:
  gute Mischung I/O- und CPU-lastiger Prozesse
- aber: I/O-Prozesse können benachteiligt ein --> Kap.Scheduling



## I/O 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

```
schnoerr@clswork: ~ 3% cat /proc/interrupts
           CPU0       CPU1 [...]
  0:      11102          0  IR-IO-APIC-edge      timer
  1:          9          0  IR-IO-APIC-edge      i8042
  8:          1          0  IR-IO-APIC-edge      rtc0
  9:          0          0  IR-IO-APIC-fasteoi   acpi
 12:        164          0  IR-IO-APIC-edge      i8042
 16:          0          0  IR-IO-APIC-fasteoi   uhci_hcd:usb3, pata_jmicron
 17:         10          0  IR-IO-APIC-fasteoi   firewire_ohci
 18:          0          0  IR-IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb8
 19:      10182          0  IR-IO-APIC-fasteoi   ata_piix, ata_piix, [...]
 21:          0          0  IR-IO-APIC-fasteoi   uhci_hcd:usb4
 23:          0          0  IR-IO-APIC-fasteoi   ehci_hcd:usb2, uhci_hcd:usb6
 24:        183        340  IR-IO-APIC-fasteoi   nvidia
 54:       2076          0  IR-IO-APIC-fasteoi   nvidia
 61:        336          0  IR-IO-APIC-fasteoi   snd_hda_intel
 88:          0          0  DMAR_MSI-edge      dmar0
 89:          0          0  DMAR_MSI-edge      dmar1
 91:          1          0  IR-PCI-MSI-edge      eth0
 92:        290          0  IR-PCI-MSI-edge      eth0-TxRx-0
 93:         24          0  IR-PCI-MSI-edge      eth0-TxRx-1
[...]
```

## Interrupt-Handler (1)

**Für jedes Gerät:**

- Interrupt Request (IRQ) Line
- Interrupt-Handler (Interrupt Service Routine (ISR)) ist Teil des Gerätetreibers
- C-Funktion
- läuft in speziellem Kontext (Interrupt Context, I-Register der CPU gesetzt)
- „top half“ und „bottom half“

**„top half“**

- Interrupt-Handler
- startet sofort, erledigt zeitkritische Dinge, z.B.:
  - Kontext-Wechsel,
  - bestätigt der Hardware den Erhalt des Interrupts (Interrupt acknowledge)
  - setzt Gerät zurück
- alles andere „bottom half“

**„bottom half“**

- heißt Tasklet
- startet später, macht eigentliche Arbeit



## Interrupt-Handler (2)

In welchem Kontext läuft was ?

- **User Context:**
  - unterbrechbar (HW oder SW interrupts)
  - kann System-Calls aufrufen
- **Process Context:**
  - nach Software-Interrupt aus User-Kontext
  - läuft im Kernel
  - Daten zwischen Kernel- und Prozessspeicher übertragen
  - nur durch HW-Interrupt unterbrechbar
- **Kernel Context:**
  - Funktionen des Kernels
  - kein Datenaustausch zwischen Kernel- und User-Space
  - nur durch HW-Interrupt unterbrechbar
- **Interrupt Context:**
  - Software- und Hardware-Interrupts



## Interrupt-Handler (3)

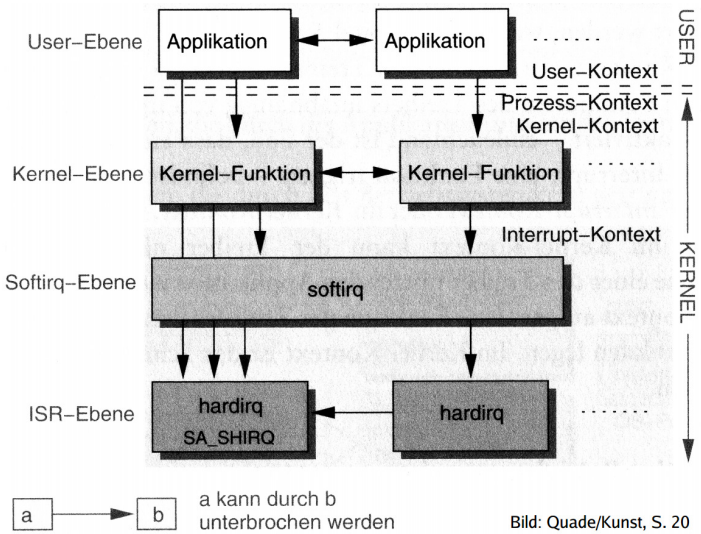



Bild: Quade/Kunst, S. 20



## Interrupt-Handler (5)

Top und bottom half / Tasklet

- **top half (ISR):**
  - erledigt zeitkritische Dinge
  - erzeugt Tasklet und beendet sich
  - dabei sind Interrupts gesperrt
- **bottom half (Tasklet)**
  - für längere Berechnungen für Interrupt-Behandlung zuständig
  - dabei Interrupts zugelassen
- **Tasklets**
  - ist kein Prozess (struct tasklet_struct)
  - läuft direkt im Kernel
  - im Interrupt-Kontext
  - zwei Priotitäten:
    - *tasklet_hi_schedule:* startet direkt nach ISR
    - *tasklet_schedule:* startet erst, wenn kein anderer Soft-IRQ mehr anliegt

--> Lit. [4],[5]

## System Calls / Software Interrupts (1)

Was sind System Calls / Software Interrupts ?

- **System-Call:**
  - Mechanismus, über den ein Anwendungsprogramm Dienste des BS nutzt
  - bei Aufruf eines System-Calls wechselt BS in den Kernel-Mode („privilegierter Modus“)
  - für viele Aufgaben sind Rechte notwendig, die normale Anwendungen nicht besitzen (User Mode vs. Kernel-Mode), --> nur über System-Calls möglich, z.B.
    - Zugriff auf Geräte (I/O)
    - Kommunikation mit anderen Prozessen (IPC)
- **Software-Interrupt (Trap):**
  - oft genutzt zur Implementierung von System-Calls
  - Nummer des System-Calls in Register eintragen und Software-Interrupt auslösen
  - Wechsel in den Kernel-Mode und Bearbeitung des Interrupts/System-Calls



## System Calls / Software Interrupts (2)

*/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
[...]
```



## System Calls / Software Interrupts (3)

Beispiel für System-Call:

Library-Funktion fread()

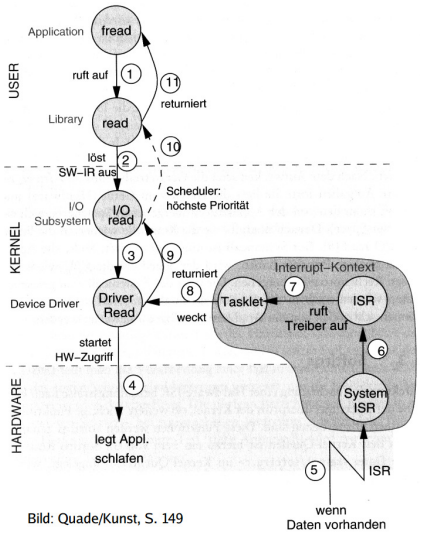



Bild: Quade/Kunst, S. 149



## Praxisbeispiele

## Praxis: System Calls für Entwickler (1)

**open()**

**Daten zum Lesen/Schreiben öffnen**

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

Rückgabe: File Descriptor

```
man 2 open
```

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



## Praxis: System Calls für Entwickler (2)

**read()**

**Daten aus Socket/Datei lesen**

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

Rückgabe: Anzahl gelesene Bytes

```
man 2 read
```

Beispiel:
```
int bufsize = 128; char line[bufsize+1];
int fd = open( "/tmp/datei.txt", O_RDONLY );
int len = read( fd, line, bufsize );
```



## Praxis: System Calls für Entwickler (3)

**write()**

**Daten in Socket / Datei (File Descriptor) schreiben**

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

Rückgabe: Anzahl geschriebene Bytes

```
man 2 write
```

Beispiel:
```
int len = write( fd, line, bufsize );
```



## Praxis: System Calls für Entwickler (4)

**close()**

**Socket / Datei (File Descriptor) schließen**

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

Rückgabe: 0 bei Erfolg, sonst -1 (`errno` enthält den Grund)

```
man 2 close
```

Beispiel:
```
int rc;
if ( (rc = close( fd )) < 0 )
    printf("Fehler bei Schließen der Datei\n");
```

## Praxis: System Calls für Entwickler (5)

**exit()**

**Programm mit Statusangabe beenden**

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

Rückgabe: keine, aber Status wird an aufrufenden Prozess weitergegeben (<-> `wait()`)

```
man 3 exit
```

Beispiel:
```
exit( 727 );
```

Anm.: auf der Shell kann der Status mit **`$?`** abgefragt werden.



## Praxis: System Calls für Entwickler (6)

**fork()**

**Neuen Prozess starten**

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

Rückgabe: Child-PID (im Vaterprozess)
-1 im Fehlerfall (im Vaterprozess)
0 (im Kindprozess)

```
man fork
```

Beispiel:
```
pid = fork();
```



## Praxis: System Calls für Entwickler (7)

**exec()**

**Anderes Programm in Prozess laden**

```
extern char **environ;

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 execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int execvpe( const char *file, char *const argv[],
             char *const envp[]);oid exit( int status );
```

Rückgabe: keine, Funktion kehrt nicht zurück

```
man 3 exec
```

Beispiel:
```
execl( "/usr/bin/emacs", "", "/etc/fstab", (char *)NULL );
```



## Praxis: Header-Dateien

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

Dateien sind in /usr/include zu finden, oder in Compiler-spezifischen Verzeichnissen.

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