

# Betriebssysteme Theorie

SS 2011

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

Foliensatz B (14.04.2011)  
Interrupts



14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-1

## Motivation (2)

- 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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-3

## Motivation (1)

- Festplattenzugriff ca. um Faktor 1.000.000 langsamer als Ausführen einer CPU-Anweisung
- 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;
}
```

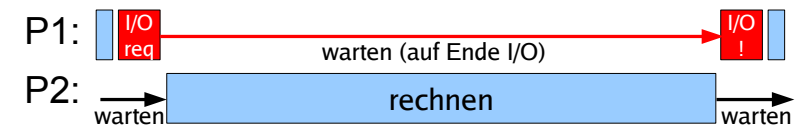
14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-2

## Motivation (3)

- 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?

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-4

## Motivation (4)

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

## 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 abwarten, bis passender Interrupt kommt

### Nachteile

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

## Interrupt-Klassen

- **I/O (Eingabe/Ausgabe, asynchr. 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, ...

## Interrupt-Bearbeitung (1)

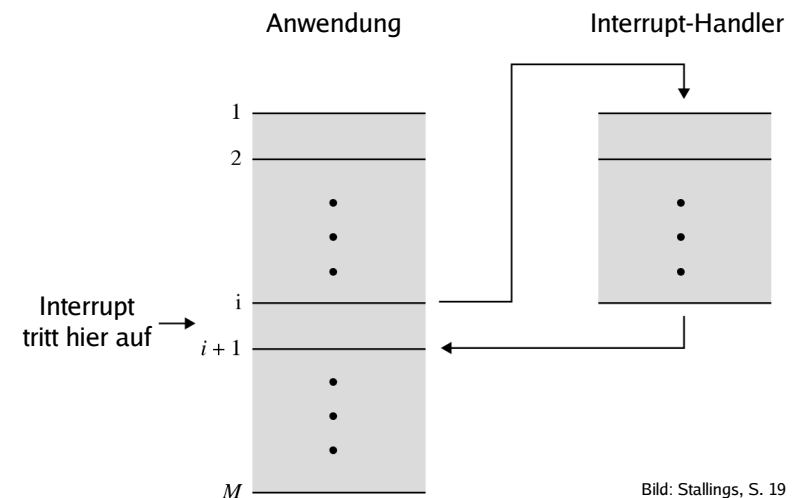


Bild: Stallings, S. 19

## 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: Scheduler wählt Prozess aus, der weiterarbeiten darf (z. B. den unterbrochenen)

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-9

## Mehrfach-Interrupts (1)

Bild: Stallings, S. 26

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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-11

## 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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-10

## Mehrfach-Interrupts (2)

Bild: Stallings, S. 26

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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-12

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

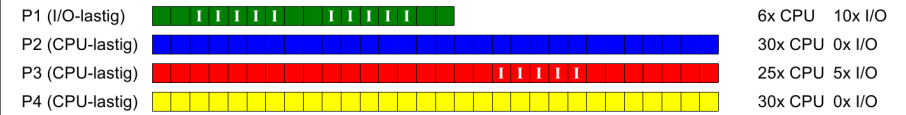
### • CPU-lastiger Prozess

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

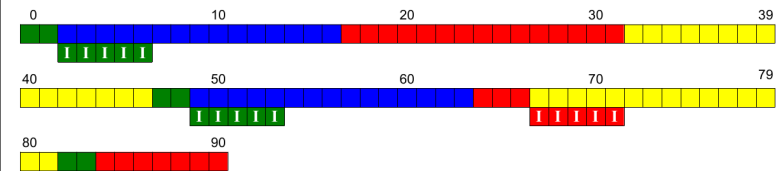
### • I/O-lastiger Prozess

- 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 (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

## 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 weiter
- Prozess stößt I/O-Operation an und legt sich schlafen (wartet auf Signal)
- optimale Performance: gute Mischung I/O- und CPU-lastiger Prozesse

# 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
```

## Für jedes Gerät:

- Interrupt Request (IRQ) Line
- Interrupt Handler (Interrupt Service Routine, ISR) Ì 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 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.
- Alles andere Ì bottom half

### bottom half

- startet später, macht die eigentliche Arbeit

## Interrupt Handler (3)

### Wichtig: In welchem Context läuft was?

- **User Context:** unterbrechbar (HW oder SW interrupts), kann system calls aufrufen,
- **Process Context:** nach Software Interrupt aus User Context, 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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-21

## Interrupt Handler (5)

### Top und bottom half / Tasklet

### Bottom half heißt im Linux-Kernel (seit Version 2.6) Tasklet

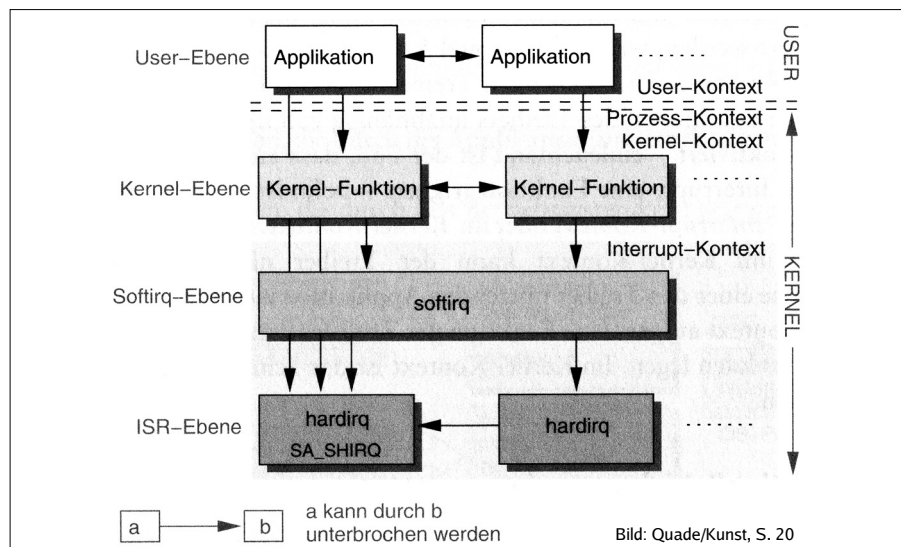
- Interrupt Service Routine (top half) erledigt das Wichtigste (zeitkritische Dinge), erzeugt Tasklet und beendet sich – dabei sind Interrupts gesperrt
- Tasklets führen längere Berechnungen durch, die zur Interrupt-Verarbeitung gehören – dabei sind Interrupts zugelassen

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-23

## Interrupt Handler (4)



14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-22

## Interrupt Handler (6)

### Tasklets

- 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

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-24

## Interrupt Handler (7)

### Mehr Informationen:

- [1] Linux Kernel 2.4 Internals, Kapitel 2,  
[http://www.faqs.org/docs/kernel\\_2\\_4/iki-2.html](http://www.faqs.org/docs/kernel_2_4/iki-2.html)
- [2] J. Quade, E.-K. Kunst: „Linux-Treiber entwickeln“,  
dpunkt-Verlag,  
<http://ezs.kr.hsnr.de/TreiberBuch/html/>

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-25

## System Calls (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
#define __NR_nice              34
#define __NR_ftime             35
#define __NR_sync              36
#define __NR_kill              37
...

```

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-27

## Sys-Calls: Software Interrupts

- **System Call:** Mechanismus, über den ein Anwendungsprogramm Dienstleistungen des BS nutzt.
- Führt eine Anwendung einen System Call aus, schaltet das BS in den **Kernel-Modus** („privilegierten Modus“) um.
- Für viele Aufgaben (etwa Zugriff auf Geräte oder Kommunikation mit anderen Prozessen) sind Rechte nötig, die normale Anwendungen nicht besitzen (User mode vs. Kernel mode). Das geht dann nur mit System Calls.
- Oft implementiert über **Software Interrupt (Trap)**. Nach Interrupt Wechsel in den Kernel-Modus.
- System-Call-Nummer in ein Register eintragen und den Software-Interrupt auslösen

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-26

## System Calls (3)

### Beispiel für einen System Call:

### Library-Funktion *read()*

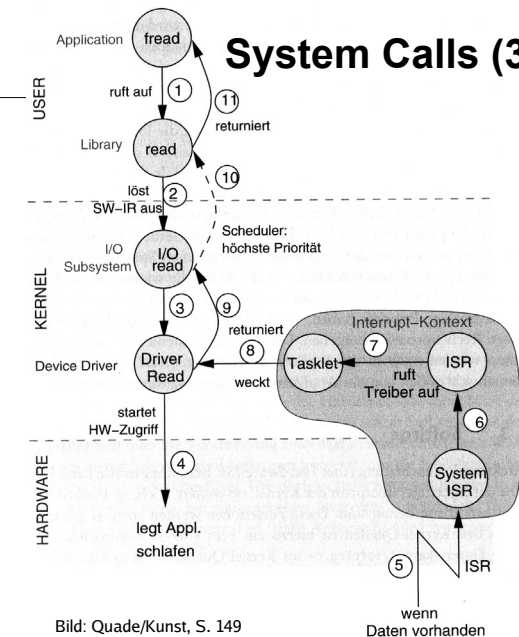


Bild: Quade/Kunst, S. 149

wenn Daten vorhanden

14.04.2011

Betriebssysteme-Theorie, Hans-Georg Eßer

Folie B-28

# System Calls für Programmierer: Standardfunktionen in C

### 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 );
```

### 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);
```

### 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;  
}
```



## Linux System Calls (5)

### `write( )` Daten in Datei (File Descriptor) schreiben

```
ssize_t read(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);  
}
```

## 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);
```

## 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);
```

## 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();
```

### **exec():** Anderes Programm in Prozess laden

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl(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