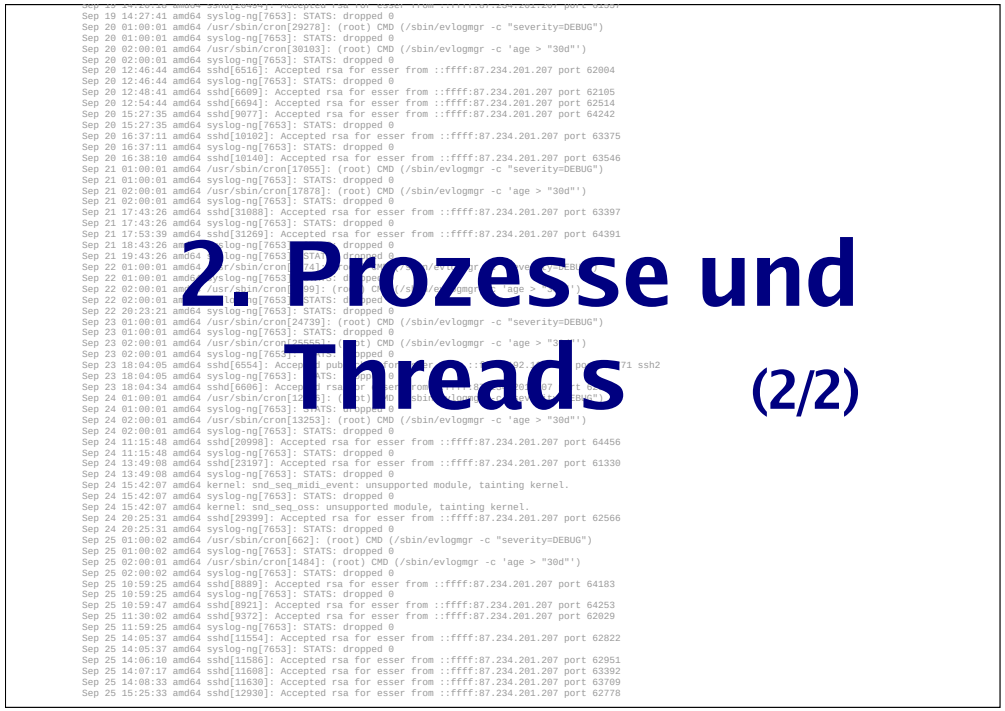


2. Prozesse und Threads (2/2)

Nachträge (2/3)

Abbruch aller Kind-Prozesse

Zwei Szenarien:

1. Shell wird mit `exit` verlassen
   ➞ Kind-Prozesse laufen weiter.
2. Shell wird gewaltsam geschlossen (`kill`, Fenster schließen etc.)
   ➞ Kind-Prozesse werden auch beendet.

Nachträge (1/3)

Abfrage, ob Programmstart über `fork()`, `exec()` erfolgreich war:

```
#include <errno.h>
main() {
  int pid = fork();
  int errno2;
  if (pid==0) {
    execl("/bin/xls",0);
    errno2=errno;
    perror ();
    printf("Fehlercode errno = %d\n",
           errno2);
  } else { wait(); }
}
```

```
> gcc -o fork-exec-fail fork-exec-fail.c
> ./fork-exec-fail
/bin/xls: No such file or directory
Fehlercode errno = 2
```

- `perror()`: Fehlermeldung in lesbarem Format
- `errno`: Globale Fehlervariable
- Nicht mit jeder `gcc`-Version...

Nachträge (3/3)

```
                          [ In xterm-Fenster ]   > nedit &

> pstree | grep nedit
     |     |-xterm---bash---nedit
> ps auxw | grep nedit
esser    24676  1.0  0.8   8248  4336 pts/4    S    15:13   0:00 nedit
> cat /proc/24676/status | grep PPid
PPid:   24659
> ps auxw|grep 24659
esser    24659  0.0  0.3   4424  1936 pts/4    Ss+  15:12   0:00 bash

                                [ In xterm-Fenster ]   > exit

> cat /proc/24676/status | grep PPid
PPid:   1
```

Praxis III: Linux-Kernel

Prozessliste (1/10)

Kernel unterscheidet nicht zwischen Prozessen und Threads.

- Doppelt verkettete, ringförmige Liste
- Jeder Eintrag vom Typ `struct task_struct`
- Typ definiert in `include/linux/sched.h`
- Enthält alle Informationen, die Kernel benötigt
- `task_struct`-Definition 132 Zeilen lang!
- Maximale PID: 32767 (short int)

Prozessliste (2/10)

```
struct task_struct {
   volatile long state;/* -1 unrunnable, 0 runnable, >0 stopped */
```

`#define` **`TASK_RUNNING`**	`0`	ausführbar – läuft gerade oder wartet („ready“)
`#define` **`TASK_INTERRUPTIBLE`**	`1`	schläft – wird geweckt, wenn er Signal erhält oder bestimmter Zustand eintritt (Kernel)
`#define` **`TASK_UNINTERRUPTIBLE`**	`2`	schläft – wie oben, aber ohne Signale
`#define` **`TASK_STOPPED`**	`4`	läuft nicht und kann auch nicht (nach Signalen SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
`#define TASK_TRACED`	`8`	
`#define` **`TASK_ZOMBIE`**	`16`	terminiert, aber Vaterprozess hat noch nicht `wait()` aufgerufen.
`#define TASK_DEAD`	`32`	

Prozessliste (3/10)

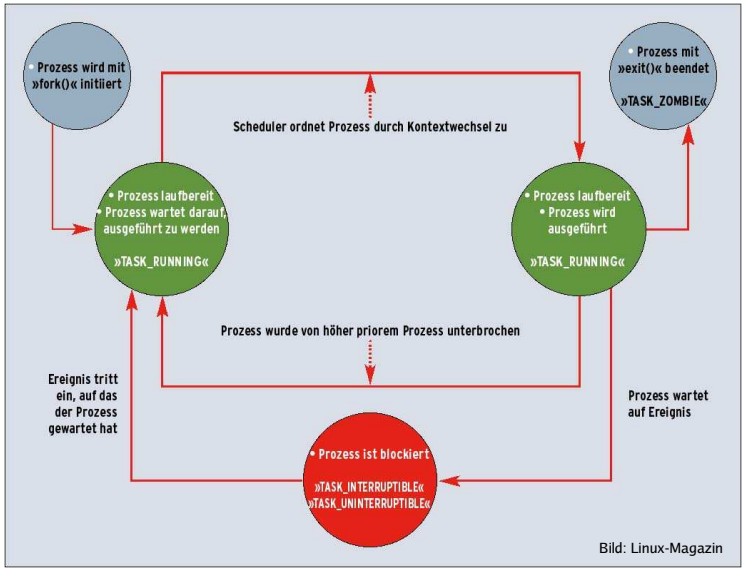
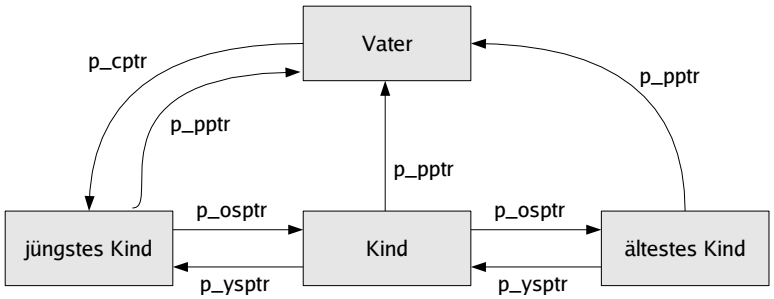


Bild: Linux-Magazin

Prozessliste (4/10)

Verwandtschaftsverhältnisse (alte Linux-Version)

```
struct task_struct {
  [...]
  struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
```



Prozessliste (5/10)

Verwandtschaftsverhältnisse (neue Linux-Version)

```
struct task_struct {
  [...]
  struct task_struct *parent;  /* parent process */
  struct list_head children;   /* list of my children */
  struct list_head sibling;    /* linkage in my parent's children list */
```

Zugriff auf alle Kinder:

```
list_for_each(list, &current->children) {
  task = list_entry(list, struct task_struct, sibling);
  /* task zeigt jetzt auf eines der Kinder */
}
```

Vom aktuellen Pfad durch den Prozessbaum bis zu `init`:

```
for (task = current; task != &init_task; task = task->parent) {
  ...
}
```


Prozessliste (6/10)

Prozessgruppen und Sessions

```
struct task_struct {
  [...]
  struct task_struct *group_leader;
       /* threadgroup leader */
  [...]
  /* signal handlers */
  struct signal_struct *signal;
```

```
struct signal_struct {
  /* job control IDs */
  pid_t pgrp;        Process Group ID
  pid_t tty_old_pgrp;
  pid_t session;     Session ID
  /* boolean value for session
     group leader */
  int leader;
```

- Jeder Prozess Mitglied einer Prozessgruppe
- Process Group ID (PGID) – `ps j`
- `current->signal->pgrp`

Prozessliste (7/10)

Prozessgruppen

- Signale an alle Mitglieder einer Prozessgruppe:
  `killpg(pgrp, sig);`
- Warten auf Kinder aus der eigenen Prozessgruppe:
  `waitpid(0, &status, ...);`
- oder einer speziellen Prozessgruppe:
  `waitpid(-pgrp, &status, ...);`

Prozessliste (8/10)

Sessions

- Meist beim Starten einer Login-Shell neu erzeugt
- Alle Prozesse, die aus dieser Shell gestartet werden, gehören zur Session
- Gemeinsames „kontrollierendes TTY“

Prozessliste (9/10)

```
> ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
19287  7628  7628 19287 pts/8    19287 S      500   0:00 /bin/sh /usr/bin/mozilla -mail
 7628  7637  7628 19287 pts/8    19287 Sl     500  20:50 /opt/moz/lib/mozilla-bin -mail
 9634 10095 10095 10095 tty1     10114 Ss     500   0:00 -bash
10095 10114 10114 10095 tty1     10114 S+     500   0:00 /bin/sh /usr/X11R6/bin/startx
10095 10115 10114 10095 tty1     10114 S+     500   0:00 tee /home/esser/.X.err
10114 10135 10114 10095 tty1     10114 S+     500   0:00 xinit /home/esser/.xinitrc
10135 10151 10151 10095 tty1     10114 S      500   0:00 /bin/sh /usr/X11R6/bin/kde
10151 10238 10151 10095 tty1     10114 S      500   0:00 kwrapper ksmserver
10258 10270 10270 10270 pts/2    10270 Ss+    500   0:00 bash
10276 10278 10278 10278 pts/4    10278 Ss+    500   0:00 bash
10260 10284 10284 10284 pts/5    10284 Ss+    500   0:00 bash
10275 10292 10292 10292 pts/6    10989 Ss     500   0:00 bash
10259 10263 10263 10263 pts/1    10263 Ss+    500   0:00 bash
10263 28869 28869 10263 pts/1    10263 S      500   0:16 konqueror /media/usbdisk/dcim
10263 28872 28872 10263 pts/1    10263 S      500   0:13 konqueror /home/esser
29201 29203 29203 29203 pts/7    29203 Ss+    500   0:00 bash
 4822  4823  4823  4823 pts/14    4823 Ss+    500   0:00 -bash
 4823 31118 31118  4823 pts/14    4823 S      500   0:00 nedit kernel/sched.c
 4823 31297 31297  4823 pts/14    4823 S      500   0:00 nedit kernel/fork.c
23115 32703 32703 23115 pts/13   32703 R+     500   0:00 ps j
```

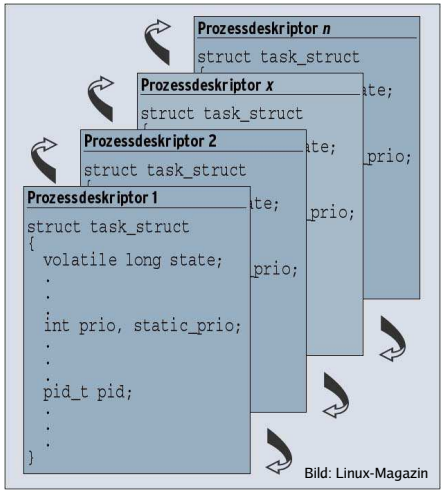

Prozessliste (10/10)

Durch die Liste laufen

```
struct list_head tasks;


#define next_task(p)
  list_entry((p)->tasks.next,
  struct task_struct, tasks)
#define prev_task(p)
  list_entry((p)->tasks.prev,
  struct task_struct, tasks)

#define for_each_process(p)
  for (p = &init_task ;
       (p = next_task(p))
        != &init_task ; )
```



Prozesserzeugung (1/2)

Wichtigste Datei in den Kernel-Quellen: `kernel/fork.c` (enthält u. a. `copy_process`)

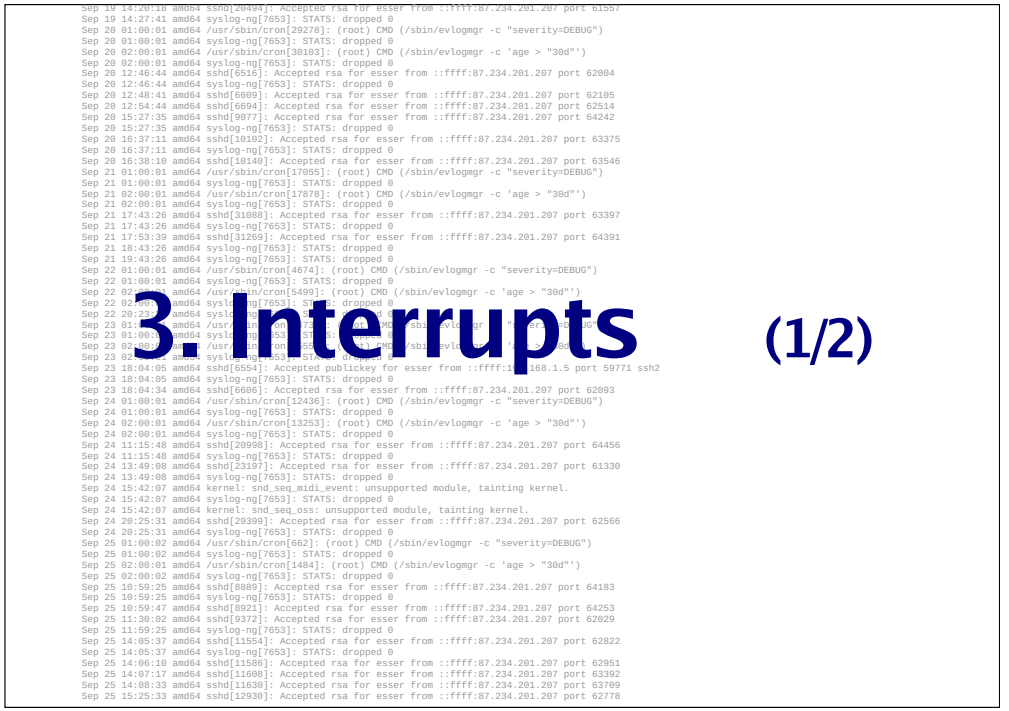
- `fork()` ruft `clone()` auf,
- `clone()` ruft `do_fork()` auf, und
- `do_fork()` ruft `copy_process()` auf, darin:

Prozesserzeugung (2/2)

`copy_process()` macht:

- `dup_task_struct()`: neuer Kernel Stack, `thread_info` Struktur, `task_struct`-Eintrag
- Kind-Status auf `TASK_UNINTERRUPTIBLE`
- `copy_flags()`: `PF_FORKNOEXEC`
- `get_pid()`: Neue PID für Kind vergeben
- Je nach `clone()`-Parametern offene Dateien, Signal-Handler, Prozess-Speicherbereiche etc. kopieren oder gemeinsam nutzen
- Verbleibende Rechenzeit aufteilen (➞ Scheduler)

Danach: aufwecken, starten (Kind kommt vor Vater dran)



Threads im Kernel

- Linux kennt keine Threads, sondern betrachtet diese als Prozesse
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt
- Jeder Thread hat `task_struct` und sieht für den Kernel wie ein normaler Prozess aus
- Fundamental anders als z. B. Windows und Solaris

Interrupts: Einführung

Interrupt-Klassen

- **Software-Interrupts (Exception, synchroner Interrupt)**
  Falscher Speicherzugriff, Division durch 0, falsche CPU-Instruktion, ...
- **Timer**
- **I/O (Eingabe/Ausgabe, asyonchr. Interrupt)**
  Vom I/O-Controller: Aktion abgeschlossen
- **Hardware-Fehler**
  Stromausfall, RAM-Paritätsfehler

Interrupts: Einführung

Wozu Interrupts?

- **Effizienz**
  I/O-Zugriff sehr langsam ➞ sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist
- **Programmierlogik**
  Nicht immer wieder Gerätestatus abfragen, sondern abwarten, bis passender Interrupt kommt
- **Kein Polling**
  Polling: BS fragt regelmäßig bei allen Geräten nach, ob ein Ereignis stattgefunden hat

Interrupts: Einführung

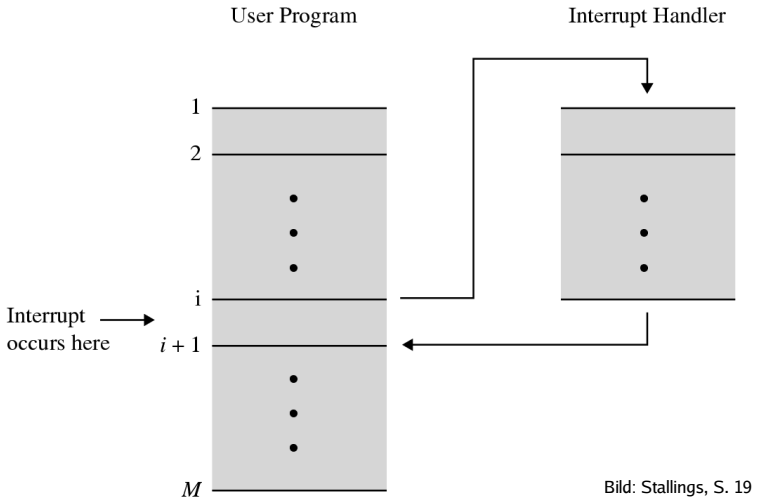


Bild: Stallings, S. 19

Interrupts: Einführung

Grundsätzlich

- Interrupt tritt auf
- Laufender Thread wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
- BS speichert Daten des Threads (wie bei Prozesswechsel ➞ Scheduler)
- BS ruft Interrupt-Handler auf
- Danach: Scheduler wählt Prozess aus, der weiterarbeiten darf (z. B. den unterbrochenen)

Interrupts: Einführung

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

Interrupts: Einführung

Was tun bei Mehrfach-Interrupts?

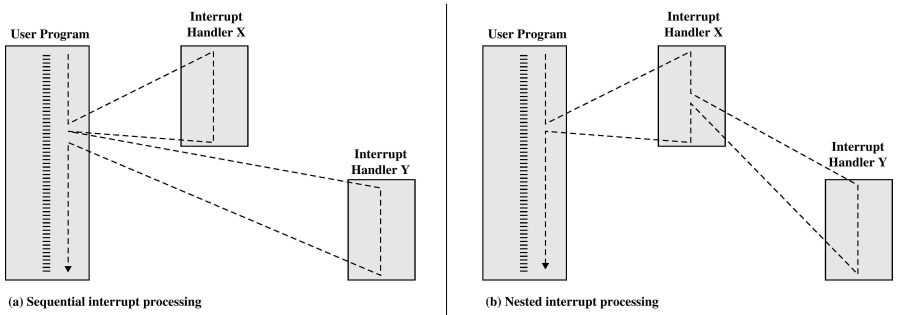
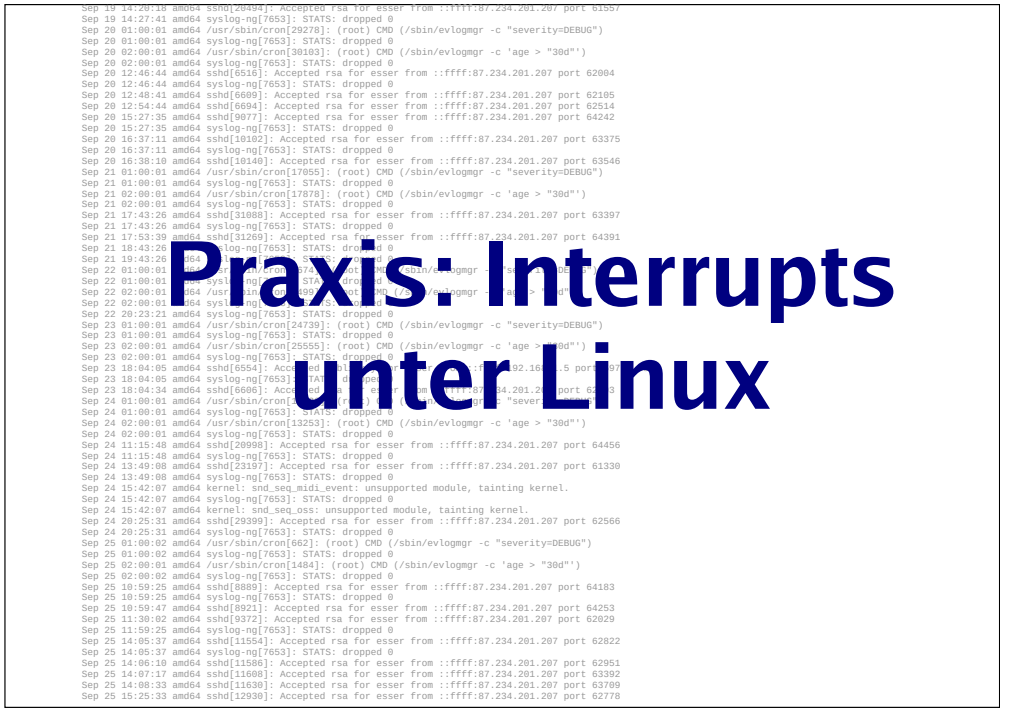


Bild: Stallings, S. 26



Praxis: Interrupts unter Linux

Interrupts: Einführung

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)

Praxis

```
> 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

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 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 ➞ bottom half

bottom half

- startet später, macht die eigentliche Arbeit
- mehr dazu in der nächsten Vorlesung

Treiberprogrammierung

Treiber registrieren mit Interrupt handler:

```
int request_irq(
  unsigned int irq,              /* Welche IRQ-Nummer?     */
  irqreturn_t (*handler)(int, void *, struct pt_regs *),
  unsigned long irqflags,
  const char * devname,          /* Gerätename->/proc/int..*/
  void *dev_id);
```

- Interrupt mit IRQ `irq` wird ausgelöst
- BS ruft Interrupt handler `handler()` auf
- Flags:

SA_SHIRQ:	Interrupt für mehrere Treiber
SA_INTERRUPT:	Lokale Interrupts werden gesperrt
SA_SAMPLE_RANDOM:	Interrupts treten „zufällig“ auf, nutzen; Entropie-Vergrößerung, Zufallszahlen

Treiberprogrammierung

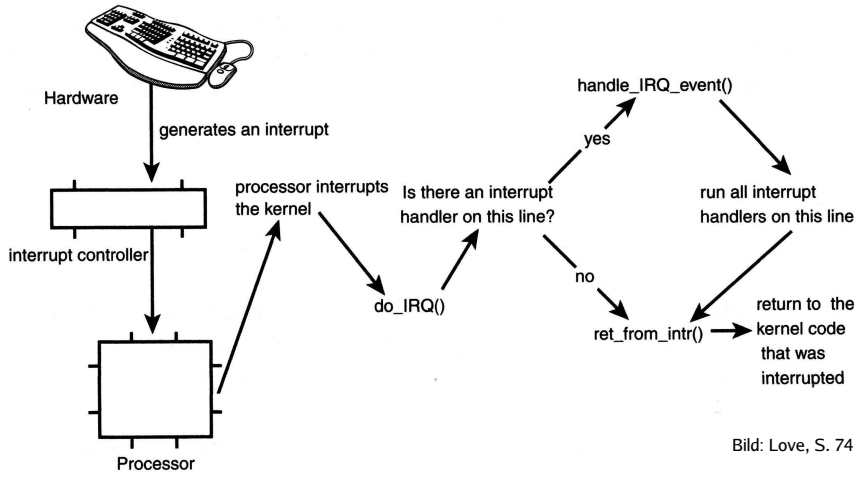


Bild: Love, S. 74

Treiberprogrammierung

Beispiel: Timer

`drivers/char/rtc.c`, `rtc_init()`

```
if (request_irq(rtc_irq, rtc_interrupt, SA_INTERRUPT,
                "rtc", (void *)&rtc_port)) {
   printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
   return -EIO;
}
```

- `rtc_irq` = 8
- `rtc_interrupt`: der eigentliche Handler
- Flag: `SA_INTERRUPT`

Treiberprogrammierung

```
irqreturn_t rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs) {
        spin_lock (&rtc_lock);
        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

        if (rtc_status & RTC_TIMER_ON)
                mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

        spin_unlock (&rtc_lock);

        /* Now do the rest of the actions */
        spin_lock(&rtc_task_lock);
        if (rtc_callback)
                rtc_callback->func(rtc_callback->private_data);
        spin_unlock(&rtc_task_lock);
        wake_up_interruptible(&rtc_wait);

        kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

        return IRQ_HANDLED;
}
```


Treiberprogrammierung

Eigener Handler wird aufgerufen in

`kernel/irq/handle.c`, `handle_IRQ_event()`:

```
int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                     struct irqaction *action) {
        int ret, retval = 0, status = 0;

        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();

        do {
                ret = action->handler(irq, action->dev_id, regs);
                if (ret == IRQ_HANDLED)
                        status |= action->flags;
                retval |= ret;
                action = action->next;
        } while (action);

        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);
        local_irq_disable();

        return retval;
}
```


Vorschau

Nächste Vorlesung:

- Mehr zu Interrupts und Treibern
- Sys Calls (ähnlich wie Interrupts)