Embedded-Linux Treiber

Andreas Klinger
ak@it-klinger.de

IT - Klinger
http://www.it-klinger.de

Linuxhotel
06.03.2018

linuxhotel

Teil I

Gerätetreiber

Gerätetreiber

Linux-System I

Userspace

- niedrigste Privilegstufe auf CPU
- virtueller Adressraum
  → geschützt mittels MMU
  → Default: 3 GB Adressbereich (32-Bit)
- Startpunkt für Programme und Dämonen

Linux-System II

Linux-Kernel

- höchste Privilegstufe
  → Hardware-Zugriff
- logischer Adressraum
  → konstanter Offset zu physikalischen Adressen
  → kein Schutzkonzept
  → Default: 1 GB Adressbereich (32-Bit)
- Eintritt durch Prozeß- oder Interruptkontext

linuxhotel

Linux-System III

Process-Control-Subsystem

- Memory-Management
- Scheduling
- Interprozess-Kommunikation

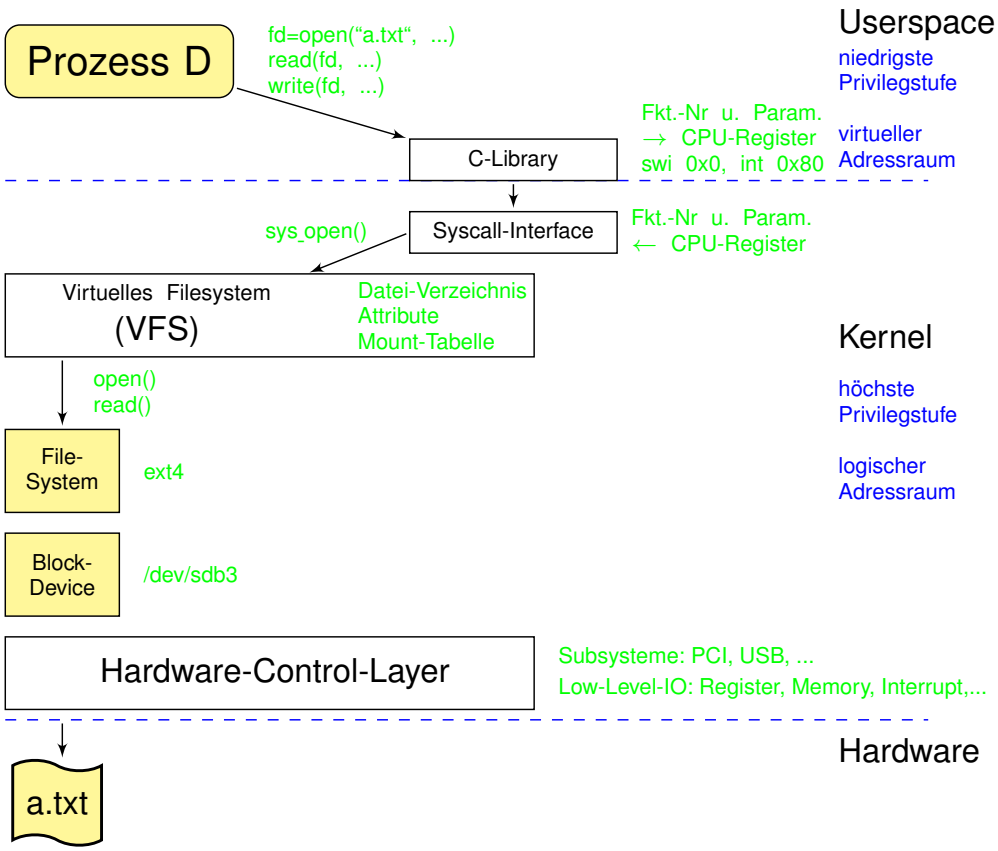
Virtuelles Filesystem

Abbildung von Treibern, Sockets, IPC, Dateien, ...
über die Dateischnittstelle

„Alles ist eine Datei.“

linuxhotel

Kernel-Aufbau — Dateizugriff



linuxhotel

- Prozess D greift auf die Datei a.txt zu
- Funktion `open()` ist in der C-Library (meist glibc) implementiert
- Aufgabe der C-Library ist es, den Syscall zur Ausführung zu bringen; dieser wird jedoch nicht als Funktion ausgeführt, da ein Wechsel der Privilegstufe erforderlich ist
- Im Syscall wird die aufzurufende Funktion samt Parametern auf CPU-Registern abgelegt und anschließend auf die höchste Privilegstufe gewechselt (mittels Software-Interrupt)
- im Syscall-Interface im Kernel wird die aufgerufene Funktion wieder hergestellt und jetzt auf der höchsten Privilegstufe aufgerufen
- im VFS werden die Dateiattribute angeschaut und dadurch bestimmt, welcher Treiber für das Öffnen der Datei zuständig ist (Filesystem, Character-Device, ...)
- im betreffenden Treiber wird nun die Funktion `open()` aufgerufen
- siehe auch:
  `syscall(2)`

Dateien und Device-Nodes

- reguläre Datei; binär oder ASCII

d Verzeichnis

s symbolischer Link

p Named Pipe; FIFO

c zeichenorientiertes Gerät (Character Device)

b blockorientiertes Gerät (Block Device)

- Socket; Netzwerkschnittstelle
- Halve-Duplex-Pipe
- Dateiüberwachung (inotify)

linuxhotel

Dateizugriff

- Zugriff und Verwendung von Dateien erfolgt mit einem Satz an Funktionen, der sog. Datei-Schnittstelle:
  `open(), read(), write(), lseek(), select(), mmap(), close(), ...`
- Identifikation erfolgt über den Dateideskriptor; einer Nummer welche die zu einem Prozeß geöffneten Dateien identifiziert, beginnend bei `0`
- zum Prozeßstart vergebene Nummern:

```
0 FILENO_STDIN (stdin)
1 FILENO_STDOUT (stdout)
2 FILENO_STDERR (stderr)
```

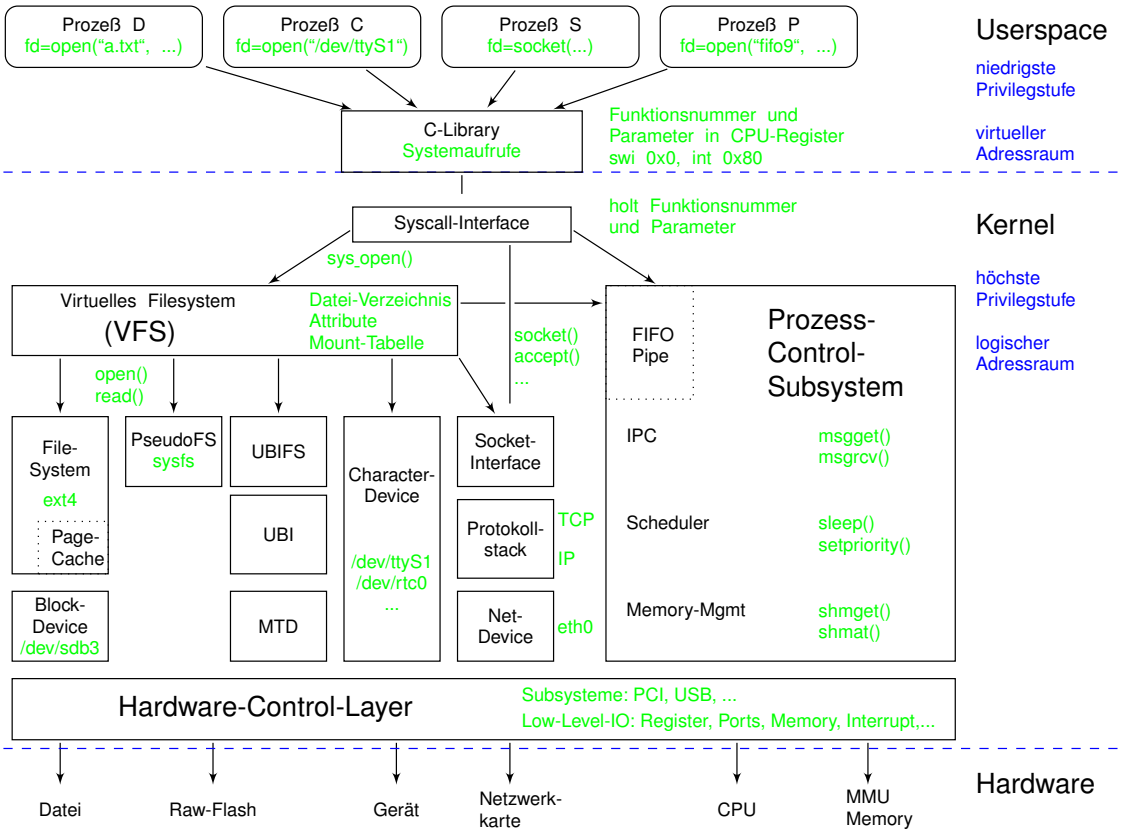
linuxhotel

Virtuelles Filesystem (VFS)

- unterschiedliche Dateiarten mit gleichem Zugriff müssen auf die korrespondierenden Objekte umgesetzt werden (Demultiplexing)
- es wird nach der zu öffnenden Datei gesucht und deren Attribute betrachtet
- für den Dateideskriptor wird ein Objekt vom Typ `struct file` angelegt
- entsprechend dem Dateityp wird im betreffenden Treiber die dazugehörige `open()`-Funktion aufgerufen
- bei weiteren Zugriffen nach dem Öffnen wird mittels der `struct file` direkt die entsprechende Funktion aufgerufen

linuxhotel

Kernel-Aufbau — reguläre Datei



linuxhotel

reguläre Datei I

- reguläre Dateien befinden sich auf einem Dateisystem und werden von diesem dargestellt
- VFS ruft die `open()`-Funktion des Dateisystems auf und dieses gibt den Inhalt der Datei wieder
- das Dateisystem wiederum benutzt einen Blockgeräte-Treiber, um den Inhalt der Datei von der Hardware zu lesen (Ausnahmen: Raw-Flash, procfs, sysfs, debugfs)
- Blockgeräte-Treiber hat die Aufgabe den Inhalt von Blöcken an das Dateisystem zu übergeben; dazu wird diesem lediglich die Blocknummer übergeben
- die Interpretation des Dateiinhaltes ist dem Dateisystem-Treiber vorbehalten

linuxhotel

reguläre Datei II

- Zuordnung, welches Dateisystem mit welchem Blockgerät verwendet werden soll findet sich in der Mounting-Tabelle
- Eintrag in Mounting-Tabelle wird beim mounten (einhängen) des Dateisystems mit dem Blockgerät in ein Unterverzeichnis des Root-Dateisystems generiert

linuxhotel

Blockorientiertes Gerät (Block Device)

- Block Devices haben einige wenige spezielle Funktionen zum Zugriff
- Block-Geräte werden fast immer im Stapel mit einem Dateisystem verwendet und nicht direkt aus dem Userspace angesprochen
- Ausnahme:
  `dd if=/dev/sdb3 of=/tmp/image.sav`

linuxhotel

FTL-Flash

- Flash-Speicher mit eigenem Controller, welcher den File-Translation-Layer (FTL) abbildet werden genauso wie „normale“ Blockgeräte angesprochen
- Linux-Dateisysteme können auf ihnen verwendet werden
- *Beispiele:*
  SD-Card, CF-Card, MMC, eMMC, SSD
- jedoch sollte auf häufiges Schreiben und Lesen verzichtet werden
  → Optionen beim Mounten

linuxhotel

Raw-Flash

- Raw-Flash-Speicher werden direkt vom Betriebssystem als Flash-Speicher angesprochen
- *Beispiele:*
  NOR-Flash, NAND-Flash
- Linux-Flash-Treiber muß Wear-Leveling, Bad-Block-Mgmt., Löschen in Erase-Blocks, ... durchführen
- nur spezielle Flash-Filesysteme wie JFFS2 oder UBIFS können verwendet werden
- diese wiederum benötigen kein Block-Device sondern spezielle Gerätetreiber, zusammengefaßt als Memory-Technology-Devices (MTD)

linuxhotel

Device-Node I

- ein Device-Node ist ein Eintrag in einem Dateisystem mit speziellen Attributen, um ihn von regulären Dateien zu unterscheiden
- der Device-Node setzt nicht das Vorhandensein eines Treibers voraus, sondern stellt lediglich einen Eintrittspunkt in den Treiber mit den korrespondierenden Attributen dar
- der Name und das Verzeichnis sind frei wählbar und es kann auch viele Device-Nodes mit den gleichen Attributen (Device-Typ, Major-Nr, Minor-Nr) geben
- erst beim Zugriff (öffnen) eines Device-Nodes wird vom VFS die Verbindung zum korrespondierenden Treiber mit der gleichen Majornummer hergestellt

linuxhotel

Device-Node II

- Beispiel:
  Character-Device-Node mit Majornummer 240 und Minornummer 0 anlegen:

```
mknod /dev/kio c 240 0
```

linuxhotel

Zeichenorieniertes Gerät (Character Device) I

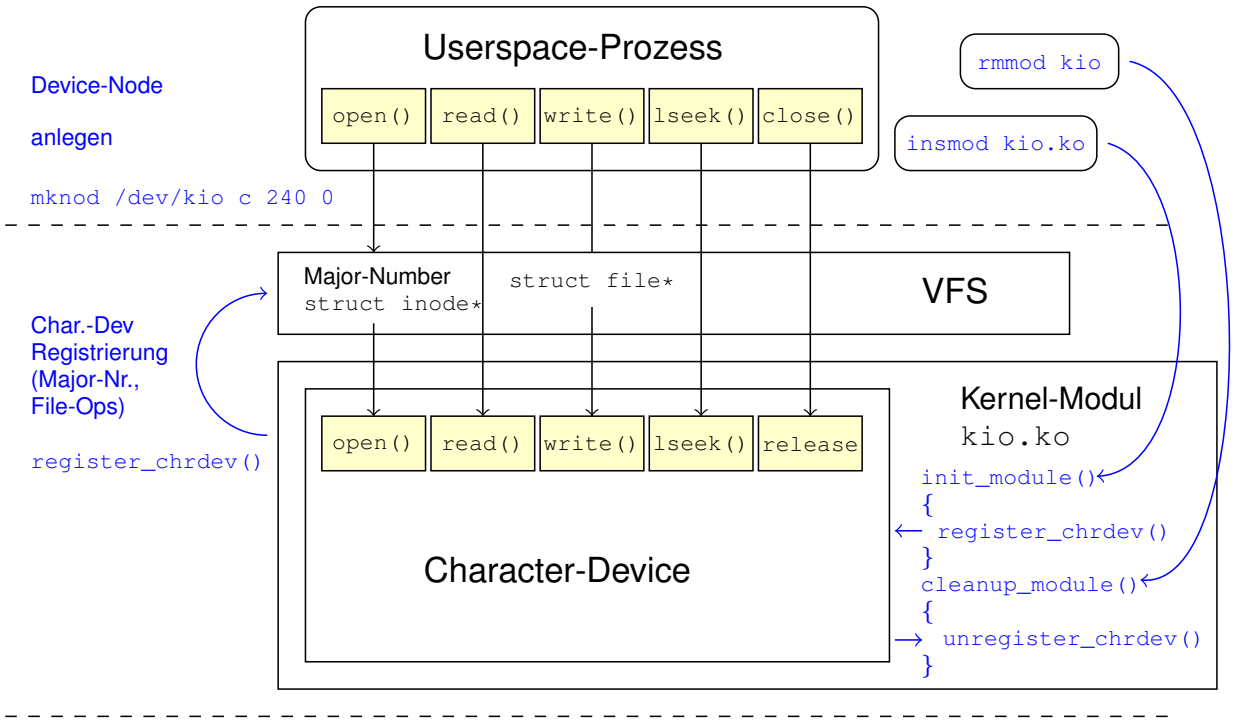
- Character Devices müssen im Kernel beim VFS registriert werden
- dazu wird neben einer freien Majornummer eine Referenz auf eine Struktur mit Zeigern auf die implementierten Dateioperationen `open()`, `read()`, `write()`, ... benötigt (`struct file_operations`)
- die Angabe des Namens des Character Devices dient der Zuordnung registrierter Treiber; diese hat jedoch keinen steuernden Einfluß
- stellt das VFS fest, daß es sich bei der Datei um einen Device-Node vom Typ Character Device handelt, wird die entsprechende Majornummer in der Liste der registrierten Character Devices gesucht

linuxhotel

Zeichenorieniertes Gerät (Character Device) II

- der zur Majornummer passende Zeichengeräte-Treiber wird aufgerufen
- Liste der registrierten Majornummers und zugehöriger Zeichengeräte-Treiber abfragen:
  `cat /proc/devices`
- im Treiber sind die eigentlichen Dateioperationen implementiert
- Minornummer dient ausschließlich dem Treiber zur Implementierung unterschiedlicher Funktionalitäten oder unterschiedlicher Ports

Character-Device



linuxhotel

- Kernelmodul wird auf der Shell mit `insmod kio.ko` zum laufenden Linux-Kernel hinzugelinkt
- im Modul wird Einsprungfunktion ausgeführt
- in dieser Funktion wird im Bild ein Character-Device beim VFS registriert mit der Funktion `register_chrdev()`
- nun ist das Character-Device verfügbar und sollte in der Datei `/proc/devices` sichtbar sein
- wird im Userspace ein passender Devicenode angelegt und mit `open()` verwendet, dann löst das VFS die Attribute auf und ruft vom registrierten Treiber die Funktion `open()` auf

Netzwerkgerät (Net Device)

- Netzwerkgeräte werden ausschließlich im Zusammenspiel mit dem Netzwerk-Protokoll-Stack verwendet
- dabei übernehmen diese die Aufgabe, das eigentliche Netzwerkgerät anzusprechen, die Daten zu übertragen und zu empfangen
- erkannte Netzwerkgeräte anzeigen:
  `cat /proc/net/dev`
- an der Schnittstelle zum Protokoll-Stack können diese ein- und ausgehängt werden
- Beispiel: Netzwerkgerät einhängen und IP-Adresse zuweisen
  `ifconfig eth0 192.168.0.81`

linuxhotel

Prozess-Control-Subsystem

- Zugriff auf Memory-Management, Scheduling und Interprozesskommunikation über Syscall-Schnittstelle
- zentraler Kernelteil

linuxhotel

Wann benötige ich welchen Treiber? I

Block-Device
wahlfreier Zugriff auf Massenspeicher mit einem darauf aufbauenden Dateisystem
Bsp.: `/dev/sdb5`

File-System
Dateisystem aufbauend auf Blockgerätetreiber
Bsp.: `ext2`

Pseudo-File-System
Dateisystem ohne Hardwareanbindung liefert Informationen aus dem Betriebssystem
Bsp.: `proc`, `sysfs`, `debugfs`

linuxhotel

Wann benötige ich welchen Treiber? II

Memory-Technology-Device
unmanaged Flash (NAND, NOR, ...) ohne FTL-Controller
Bsp.: `/dev/mtd3`

Flash-Filesystem
Dateisystem aufbauend auf MTD-API
Bsp.: `jffs2`, `ubi/ubifs`

Net-Device
Netzwerkgerät, welches vom Protokollstack verwendet werden kann
Bsp.: `eth0`, `ppp0`

Character-Device
Zeichenorientiertes Gerät sowie sonstige Funktionalität
Bsp.: `/dev/ttyS0`, `/dev/mem`

linuxhotel

Kernel-Module I

- Kernel-Module sind Objektdateien mit der speziellen Eigenschaft, zur Laufzeit dem Kernel hinzugelinkt und auch wieder entfernt werden zu können
- das Hinzufügen erfolgt mit dem Shell-Programm `insmod` oder `modprobe` und das Entfernen mittels `rmmod`
- beim Hinzufügen wird die `init_module()`-Funktion durch den Kernel aufgerufen; in ihr kann dann wiederum ein Character-Device registriert werden
- beim Entfernen ruft der Kernel die `cleanup_module()`-Funktion auf; hier wird dann das registrierte Character-Device wieder entfernt

Kernel-Module II

- Makros `module_init()` und `module_exit()` kennzeichnen die Lade- und Entladefunktionen des Kernel-Moduls
- Shell-Programm `lsmod` oder auch `cat /proc/modules` listen alle geladenen Kernel-Module inkl. Verwendungszähler und Grösse auf

linuxhotel

Beispiel: Kernel-Modul mit Character-Device I

```
static int kio_major = 240;
static char* kio_name = "kio";

static struct file_operations kio_fops = {
    .owner   = THIS_MODULE,
    .open    = kio_open,
    .release = kio_release,
    .read    = kio_read,
    .write   = kio_write,
    ...      // weitere Funktionen
             // nicht benoetigte weglassen
};
```

linuxhotel

1 statisch vergebene Majornummer 240

2 Name, welcher für Resourcenanforderungen verwendet wird

4 `struct file_operations` enthält die Zeiger auf die Einsprungfunktionen des Character-Device-Drivers

5 `owner`-Element dient der Zuordnung zwischen Character-Device-Driver und Kernelmodul (Referenzzählung)

6-9 Zuweisung der Treiberfunktionen an die entsprechenden Elemente in der `struct file_operations`

- Funktionszeigernamen entsprechen den Namen im Userspace einzig die Funktion `close()` heißt im Kernel `release()`

Beispiel: Kernel-Modul mit Character-Device II

```
static int kio_init (void)
{
    int err;

    err = register_chrdev (kio_major, kio_name, &kio_fops);
    if (err < 0) {
        printk (KERN_ERR "kio_init(): err=%d\n", err);
        return err;
    }
    return 0;
}
module_init (kio_init);

static void kio_exit (void)
{
    unregister_chrdev (kio_major, kio_name);
}
module_exit (kio_exit);
```

linuxhotel

13 Funktion `kio_init()` dient als Einsprungfunktion in das Kernelmodul, welche beim Laden des Moduls aufgerufen wird (in der Shell mit `insmod` oder `modprobe`

17 Anmelden des Character-Devices mit seiner Major-Nummer, seinem Namen und der Treiberfunktionen beim VFS

19 im Fehlerfall wird String an Kernel-Log-Buffer übergeben
`KERN_ERR` ist Makro, welches sich in `"<3>"` auflöst und die Priorität der Logausgabe darstellt
String muß mit `"\n"` abgeschlossen sein

24 mit dem Makro `module_init` wird die Einsprungfunktion als solche definiert

26 `kio_exit()` ist die Aussprungfunktion, welche beim Shellaufruf `rmmod` aufgerufen wird

28 Abmelden des Character-Devices beim VFS

30 mit dem Makro `module_exit` wird die Aussprungfunktion definiert

Kernel-Modul-Parameter I

- Modul-Parameter können beim Laden mitgegeben werden und im Modul selber mit dem Makro `module_param()` deklariert werden
- globale Deklaration im Kernel-Modul:

```
#include <linux/moduleparam.h>

static int kio_major = 240;
static char* kio_name = "kio-default";

module_param(kio_major, int, S_IRUGO);
module_param(kio_name, charp, S_IRUGO);
```

linuxhotel

Kernel-Modul-Parameter II

- Variablen sind global in allen Funktionen des Moduls verwendbar und sichtbar im Verzeichnis `/sys/module/<Module-Name>/`

```
static int kio_init (void)
{
  ...
  err = register_chrdev (
            kio_major, kio_name, &kio_fops);
  ...
}
```

- Hinzufügen des Moduls in der Shell:

```
insmod kio.ko kio_major=199 kio_name="new-kio"
```

linuxhotel

Character-Device im 2.6-er Treibermodell I

- seit Kernel 2.6 steht das neue Character-Device-Modell zur Verfügung; `register_chrdev()` kapselt diese Funktionen
- zu verwenden bei: dynamischer Majornummer, sysfs-Support
- globale Deklaration im Kernel-Modul:

```
#include <linux/cdev.h>

static dev_t kio_dev;
static struct cdev kio_cdev;
```

linuxhotel

Character-Device mit dynamischer Majornummer I

```
static int __init kio_init(void)
{
    int ret = 0;
    ret = alloc_chrdev_region(&kio_dev, 0, 1, kio_name);
    if (ret < 0) {
        printk(KERN_ERR "kio_init(): Major-Nr\n");
        goto out;
    }
    cdev_init(&kio_cdev, &kio_fops);
    if ((ret = cdev_add(&kio_cdev, kio_dev, 1)) < 0) {
        printk(KERN_ERR "kio_init(): Char-Dev\n");
        goto out_unalloc_region;
    }
    return ret;
out_unalloc_region:
    unregister_chrdev_region(kio_dev, 1);
out:
    return ret;
}
```

linuxhotel

1 mit `__init` wird die Funktion in der Section `.init.text` erstellt
→ wird entladen, wenn nicht mehr im Speicher benötigt

4 Anfordern einer dynamischen Majornummer sowie eines Minornummer-Bereiches, hier:
1. Minornummer = 0
nur 1 Minornummer anfordern

9 initialisieren der `struct cdev` mit den Einsprungfunktionen in das Character-Device

10 Anmelden beim VFS unter Angabe, wie viele Minornummern angemeldet werden sollen

Character-Device mit dynamischer Majornummer II

```
20 module_init (kio_init);

   static void __exit kio_exit (void)
   {
       cdev_del(&kio_cdev);
25     unregister_chrdev_region(kio_dev, 1);
   }
   module_exit (kio_exit);
```

linuxhotel

22 `__exit` bewirkt, daß die Funktion in der Section `.exit.text` erstellt wird
→ als ladbares Modul vorhanden
→ als statisch zum Kernel dazugelinktes Modul weggelassen

printk-Debugging im Kernel-Modul I

- Ausgabe eines Kernel-Moduls erscheint nicht auf der Shell
- Debugging-Ausgaben können mittels `printk()` in das Kernel-Log geschrieben werden (Ringspeicher)
- der erste Wert in der Formatangabe ist ein Makro des Schweregrades; kein Parameter
- der Dämon `klogd` liest diese Meldungen aus und gibt sie an den `syslogd`-Dämon weiter
- `syslogd` schreibt die Ausgaben in das Systemlog `/var/log/messages`

linuxhotel

printk-Debugging im Kernel-Modul II

- auf neue Systemlog-Einträge in eigener Shell warten:
  ```
  tail -f /var/log/messages
  tailf /var/log/messages
  ```
- Inhalt von Kernel-Log-Speicher:
  ```
  dmesg -w
  ```

linuxhotel

Dateischnittstelle I

- Schnittstelle zum Userspace in Form von Funktionen für Dateioperationen
- über das VFS werden die Funktionen im *Prozess-Kontext* aufgerufen
- daher dürfen die Funktionen schlafen und blockieren; User-Task wird entsprechend schlafen gelegt

Dateischnittstelle II

Rückgabewerte

< 0 negativer Fehlercode
$\rightarrow$ `-1` als Rückgabe für den Userspace und
$\rightarrow$ `errno` auf positiven Fehlercode gesetzt

≥ 0 Funktion erfolgreich

linuxhotel

Dateischnittstelle - `open()`

- wird vom VFS beim Öffnen des Filedeskriptors durch den Userspace aufgerufen
- `struct file*` ist bereits gefüllt und ist die Datenstruktur, welche hinter einem Userspace-Filedeskriptor steht
- Element `private_data` ist `void*`-Zeiger und kann im Treiber frei verwendet werden
  $\rightarrow$ Zuweisung von Struktur mit treiberinstanzspezifischen Daten
- `struct inode*` repräsentiert die I-Node des Gerätetreiber-Device-Nodes, z. B. `/dev/kio1`
  $\rightarrow$ enthält unter anderem Minor- und Major-Number

linuxhotel

Beispiel: `open()`

```
int kio_open (struct inode* in, struct file* f)
{
  // Minor-Number holen und für weitere Aufrufe merken
  int minor_nr = iminor (in);

  // besser: Struktur anlegen und darin Feld mit minor_nr
  // zur Zuweisung an *private_data
  f->private_data = (void*) minor_nr;

  // = 0: alles OK  ==>  Userspace bekommt fd
  // < 0: Fehler    ==>  Userspace bekommt keinen fd
  return 0;
}
```

linuxhotel

Dateischnittstelle - `read()` I

- Funktion kopiert Daten von der Schnittstelle in den übergebenen Puffer
- übergebene Zeiger auf Puffer liegt im virtuellen Adressbereich des Userspace-Tasks und darf nicht direkt verwendet werden
- sie müssen mit speziellen Funktionen bearbeitet werden; `copy_to_user()` und `copy_from_user()`
- übergebene Offset-Wert `loff_t*` ist die Position des Schreib-/Lese-Zeigers in der Datei
- vom Treiber richtig zu setzen
  → ansonsten kein „Dateiende“ (`return 0;`) durchführbar

linuxhotel

Dateischnittstelle - `read()` II

- Rückgabewert: Anzahl an zurückgegebenen Zeichen oder Fehler (`< 0`)
- Defaultverhalten: blockierend, bis mindestens ein Zeichen gelesen werden konnte
- `__user` dokumentiert, daß Zeiger im virtuellen Adressbereich des Userspaces erwartet wird

linuxhotel

Beispiel: `read()` I

```
int kio_read (struct file* f, char __user * buf,
                          size_t cnt, loff_t* off)
{
  char kern_buf[50];    // schlecht: Stack-Verbrauch
  int nread;

  // Read-Write-Pointer prüfen ==> Abbruchbedingung
  if ( (*off) >= sizeof(kern_buf))
    return 0;

  if (cnt > sizeof(kern_buf))
    cnt = sizeof(kern_buf);

  // eigene Funktion zum Lesen von einem Gerät
  if ( (nread = kio_read_device (kern_buf, cnt)) <= 0)
    return nread;
```

linuxhotel

Beispiel: `read()` II

```
  // kopieren vom Kernel-Adressraum in
  //   den virtuellen Userspace-Adressraum
  // Rückgabe: Anzahl nicht kopierter Zeichen

  if ( !copy_to_user(buf, kern_buf, nread) )
  {
    // Read-Write-Pointer müssen wir selber mitführen,
    //   um bei einem folgenden Aufruf in
    //   Abbruchbedingung zu kommen
    (*off) += nread;
    // Rückgabe: Anzahl gelesener Zeichen
    return nread;
  }
  else
    return -EFAULT;
}
```

linuxhotel

Dateischnittstelle - `write()`

- Schreiben von Daten auf die Schnittstelle
- übergebene Puffer liegt im Userspace
- Userspace ist für erneuten Aufruf zuständig, wenn nur ein Teil der Daten geschrieben wurde
- Rückgabewert: Anzahl an weggeschriebenen Zeichen oder Fehler (`< 0`)
- Defaultverhalten: blockierend, bis mindestens ein Zeichen weggeschrieben werden konnte

linuxhotel

Beispiel: `write()` I

```
int kio_write (struct file* f, const char __user * buf,
                               size_t cnt, loff_t* off)
{
  char kern_buf[50];    // schlecht: Stack-Verbrauch
  int nwritten;

  if (cnt > sizeof(kern_buf))
    cnt = sizeof(kern_buf);

  if ( !copy_from_user(kern_buf, buf, cnt) )
  {
    // eigene Funktion zum Schreiben auf Gerät
    if ((nwritten = kio_write_dev(kern_buf, cnt)) <= 0)
      return nwritten;
```

linuxhotel

Beispiel: `write()` II

```
    // Read-Write-Pointer mitpflegen, sofern notwendig
    (*off) += nwritten;

    // Rückgabe: Anzahl weggeschriebener Zeichen
    return nwritten;
  }
  else
    return -EFAULT;
}
```

linuxhotel

Dateischnittstelle - `ioctl()`

- Kontrollanweisungen für den Treiber
- Für jedes Kommando kann es ein anderes Argument geben
  $\Rightarrow$ sämtliche Funktionalität kann damit generiert werden
- `unlocked_ioctl()` hält keinen BKL
- `compat_ioctl()` hält keinen BKL und dient der Kompatibilität zwischen 32- und 64-bit-Systemen; wird aufgerufen, wenn 32-Bit-Prozeß auf einem 64-Bit-System `ioctl()` aufruft
- Kommandovergabe, siehe:
  `Documentation/ioctl/ioctl-number.txt`
- Unterschied `unlocked_ioctl()` und `compat_ioctl()`, siehe:
  https://lwn.net/Articles/119652

linuxhotel

Beispiel: `ioctl()` I

```
#define FUNC_27 _IOW ('x', 0x01, short)

#define FUNC_74 _IOWR('x', 0x02, struct kio74)

long kio_ioctl (struct file* f,
                unsigned int cmd, unsigned long arg)
{
  int ret = 0;

  switch (cmd)
  {
    case FUNC_27:
      // eigene Funktion erwartet short-Argument
      ret = kio_fkt_27 ( (short) arg);
      break;
```

linuxhotel

Beispiel: ioctl() II

```
    case FUNC_74:
      // eigene Funktion erwartet Zeiger auf Struktur
      ret = kio_fkt_74 ( (struct kio74*) arg);
      break;

    default:
      return -EINVAL;
  }

  return ret;
}
```

linuxhotel

Beispiel: release() = close()

```
int kio_release (struct inode* in, struct file* f)
{

  // Aufräumarbeiten durchführen

  return 0;
}
```

linuxhotel

4 Hardware-Zugriff
- IO-Ports und IO-Memory
- Managed Device Support
- GPIO's
- I^2C
- SPI
- Industrial-IO-Subsystem (IIO)

Zugriff auf IO-Ports

- Adressbereich (typ. 0x0000 ... 0xFFFF) parallel zum physikalischen Speicher mit Hardware-Registern
- IO-Port-Zugriff wird reserviert mit der Funktion `request_region()` und mit `release_region()` wieder freigegeben
- Schreiben von Registern in der Breite 8, 16, 32 Bit mit den Funktionen `outb()`, `outw()` und `outl()`
- Lesen von Registern in der Breite 8, 16, 32 Bit mit den Funktionen `inb()`, `inw()` und `inl()`
- Reservierte IO-Port-Bereiche anschauen: `cat /proc/ioports`

Beispiel: IO-Ports schreiben I

```
static int __init kio_init (void)
{
  ...

  if (!request_region (0x378, 3, "kio"))
  {
    return -1;
  }
}

static void __exit kio_exit (void)
{
  ...

  release_region (0x378, 3);
}
```

linuxhotel

Beispiel: IO-Ports schreiben II

```
int kio_write (struct file* f, const char __user * buf,
                               size_t cnt, loff_t* off)
{
  char kbuf[10];

  if (cnt > sizeof(kbuf))
    cnt = kbuf;


  if (!copy_from_user (kbuf, buf, cnt))
  {
    for (i = 0; i < cnt; i++)
    {
      outb (kbuf[i], 0x378);
      usleep (50);
    }
```

linuxhotel

Beispiel: IO-Ports schreiben III

```
    (*off) += cnt;
    return cnt;
  }

  return (-EFAULT);
}
```

linuxhotel

Zugriff auf IO-Memory I

- physikalischer IO-Speicher sollte reserviert und mit entsprechenden Funktionen angesprochen werden
- IO-Memory-Zugriff wird reserviert mit der Funktion `request_mem_region()` und mit `release_mem_region()` wieder freigegeben
- Schreiben von Registern in der Breite 8, 16, 32, 64 Bit mit den Funktionen `writeb()`, `writew()`, `writel()` und `writeq()`
- Lesen von Registern in der Breite 8, 16, 32, 64 Bit mit den Funktionen `readb()`, `readw()`, `readl()` und `readq()`
- Reservierte IO-Memory-Bereiche anschauen: `cat /proc/iomem`

linuxhotel

Managed Devices

- Funktionen mit Präfix `devm_`
- die von diesen Funktionen belegten Resourcen werden beim Abmelden des Treibers automatisch freigegeben
  → wenn zugehörige `struct device` freigegeben wird
- normalerweise müssen die angeforderten Resourcen in der `remove()`-Funktion wieder freigegeben werden
- Achtung: nicht für Resourcen verwenden, welche Gerät abschalten
  ⇒ nicht alle Resourcen können freigegeben werden
  Beispiel: Regulator

linuxhotel

General Purpose Input Output I

- GPIO's sollten sowohl im Kernel als auch im Userspace unter Verwendung der entsprechenden Schnittstellen verwenden werden
- im Kernel gibt es das klassische GPIO-Interface mit Funktionsnamen `gpio_*`
- dieses wurde durch das deskriptorbasierte Interface (`gpiod_*`) erneuert
  → für Neuimplementierungen ist dieses vorzuziehen
- Definition des GPIO-Controllers sowie der GPIO's im Device-Tree
- direkte Initialisierung hieraus
- für den Userspace gibt es das einfach zu verwendende Interface im sysfs mit ein paar Einschränkungen:

linuxhotel

General Purpose Input Output II

  - pro Zugriff kann nur ein GPIO geschalten werden
  - Übermittlung der Information als Strings
- performanter und universeller ist der Zugriff mittels Device-Nodes
  → C-Programm für `ioctl()`-Aufrufe notwendig
- Diagnose über bekannte und verwendete GPIO-Controller sowie deren GPIO's ist mit dem Programm `lsgpio` sowie mit dem debug-FS (`/debug/gpio`) möglich

linuxhotel

GPIO-Controller

Definition des GPIO-Controllers im Device-Tree
⇒ verwendbar im sysfs

```
gpio2: gpio@481ac000 {
    compatible = "ti,omap4-gpio";
    reg = <0x481ac000 0x1000>;
    ti,hwmods = "gpio3";
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    interrupts = <32>;
};
```

linuxhotel

1 *gpio2:*
Alias-Name für Node im Device-Tree
→ zum Referenzieren durch anderen Node
→ zum Vererben, Ändern und Erweitern

1 *gpio@481ac000*
beliebig wählbarer Name für Node im Device-Tree
→ Konvention ist es, die Adresse des Konfigurationsregisters oder der Busadresse in den Namen einzubauen

2 *compatible*
spezifiziert den Treiber, welcher initialisiert werden soll
→ Kernel-Treiber registriert sich mit genau diesem Namen

...
Treiberspezifische Einstellungen, welche vom Treiber übergeben und von ihm ausgewertet werden

Referenzierung auf GPIO-Controller durch Treiber
$\Rightarrow$ GPIO kann durch Treiber belegt und genutzt werden

```
srf04@0 {
    compatible = "devantech,srf04";
    trig-gpios = <&gpio2  7 GPIO_ACTIVE_HIGH>;
    echo-gpios = <&gpio2  8 GPIO_ACTIVE_HIGH>;
};
```

linuxhotel

3 *trig-gpios*
deskriptorbasierter GPIO
$\rightarrow$ in dieser Form von `gpiod_` -Funktionen im Kernel-Treiber direkt verwendbar

3 *&gpio2*
Referenzierung des Alias-Namens des entsprechenden GPIO-Controllers

GPIO-Treiber als Platform-Device

```
static const struct of_device_id of_srf04_match[] = {
    { .compatible = "devantech,srf04", },
    {},
};

MODULE_DEVICE_TABLE(of, of_srf04_match);

static struct platform_driver srf04_driver = {
    .probe      = srf04_probe,
    .driver     = {
        .name           = "srf04-gpio",
        .of_match_table = of_srf04_match,
    },
};

module_platform_driver(srf04_driver);
```

linuxhotel

1 `struct of_device_id` definiert implementierte Compatible-Strings, welche unterstützt werden

6 `MODULE_DEVICE_TABLE` erstellt Tabelle mit allen im Modul implementierten Device-Tree-Treibern
→ kann mit `modinfo` angezeigt werden

9 Funktion `probe()` dient als Einsprung in den Treiber, sobald im Device-Tree der Compatible-String gematched wird

16 Makro `module_platform_driver()` beinhaltet `module_init()`, `module_exit()` und Registrierung von Platform-Driver
→ erspart es den immer wieder gleichen `init()`- und `exit()`-Code zu schreiben

probe() mit GPIO-Deskriptor I

```
struct srf04_data {
    struct device       *dev;
    struct gpio_desc    *gpiod_trig;
    struct gpio_desc    *gpiod_echo;
    int                 irqnr;

    [...]
};

static int srf04_probe(struct platform_device *pdev)
{
    int ret;
    struct device       *dev = &pdev->dev;
    struct srf04_data *data = devm_kzalloc(...);

    [...]
```

linuxhotel

3,4 `struct gpio_desc` dient als Referenz auf deskriptorbasiertes GPIO-Objekt

13 Referenz auf `struct device` wird bereits vom Platform-Device geliefert

14 `struct srf04_data` muß alloziert werden

probe() mit GPIO-Deskriptor II

```
    data->gpiod_echo = devm_gpiod_get(dev, "echo",
20                                                GPIOD_IN);

    if (IS_ERR(data->gpiod_echo)) {

        dev_err(dev, "echo-gpios: err=%ld\n",
25                              PTR_ERR(data->gpiod_echo));

        return PTR_ERR(data->gpiod_echo);
    }
}
```

linuxhotel

19 Funktion `devm_gpiod_get()` holt innerhalb des Device-Tree-Nodes, welcher zum Aufruf der `probe()`-Funktion führte (angegeben durch `struct device* dev`) ein Attribut mit dem Namen `echo-gpios` und versucht daraus direkt einen deskriptorbasierenden GPIO zu initialisieren

20 es erfolgt gleich die Festlegung der Datenrichtung (hier Input)

22 mit `IS_ERR` Prüfung darauf, ob Zeiger einen Fehlercode enthält

25 Fehlercode wird mit `PTR_ERR` aus dem Fehlerzeiger geholt

probe() mit GPIO-Deskriptor III

```
    if (gpiod_cansleep(data->gpiod_echo)) {

        dev_err(dev, "cansleep-GPIOs not supported\n");
40      return -ENODEV;
    }

    data->irqnr = gpiod_to_irq(data->gpiod_echo);

45  ret = devm_request_irq(dev, data->irqnr,
            srf04_handle_irq,
            IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
            pdev->name, data);

50  [...]

}
```

linuxhotel

37 `gpiod_cansleep()` prüft, ob es sich um einen GPIO handelt, der beim Setzen oder Abfragen möglicherweise wartet
→ mittels I2C angeschlossener GPIO-Expander führt bei jedem Zugriff Telegrammverkehr aus, auf den gewartet werden muß

43 `gpiod_to_irq()` liefert die zur GPIO-Nummer gehörende Interrupt-Nummer

45 `devm_request_irq()` registriert die Interrupt-Service-Routine

IRQ mit GPIO-Deskriptor

```
static irqreturn_t srf04_handle_irq(int irq,
                                    void *dev_id)
{
    struct srf04_data *data =
5                           (struct srf04_data*)dev_id;

    if (gpiod_get_value(data->gpiod_echo)) {

        [...]
10  }

    [...]

}
```

linuxhotel

5 mithilfe der `dev_id` wird devicespezifische Datenstruktur `struct srf04_data` in die ISR reingereicht

7 mit `gpiod_get_value()` und `gpiod_set_value()` werden GPIO's abgefragt oder gesetzt

GPIO im debugfs

```
root@target: cat /sys/kernel/debug/gpio

gpiochip0: GPIOs 0-31, parent: platform/44e07000.gpio, gpio:
 gpio-6   (                    |cd                  ) in  lo IRQ
 gpio-26  (                    |dout                ) in  lo

gpiochip1: GPIOs 32-63, parent: platform/4804c000.gpio, gpio:
 gpio-45  (                    |sck                 ) out lo

gpiochip2: GPIOs 64-95, parent: platform/481ac000.gpio, gpio:
 gpio-72  (                    |trig                ) out lo
 gpio-73  (                    |echo                ) in  lo IRQ
 gpio-82  (                    |set-get-gpio        ) out hi
 gpio-83  (                    |sysfs               ) in  lo
```


Anzeige der GPIO-Nummern (`gpio-NN`), des Owners, Input oder Output (`in out`), aktueller Zustand (`hi lo`) sowie ob IRQ mit dem GPIO verbunden ist

9 GPIO, welcher von Treiber mit Device-Tree-Attribut `trig-gpios` genutzt wird

10 GPIO für `echo-gpios`

11 GPIO von Userspace-Anwendung, welche GPIO's mittels `ioctl()` nutzt

12 GPIO, welcher durch sysfs-Schnittstelle genutzt wird

GPIO im sysfs

in der Shell wird GPIO 20 mit Interrupt zur Erkennung der fallenden Flanke eingestellt:

```
cd /sys/class/gpio
echo 20 > export
cd gpio20
echo in > direction
echo falling > edge
cat value
```

zwei zusätzliche Terminalfenster zur Diagnose öffnen:

1. `watch 'cat /sys/kernel/debug/gpio | grep gpio-20'`
2. `watch 'cat /proc/interrupts | grep gpio'`

Was kann man beobachten, wenn der GPIO (S1) gesetzt wird?

Warten auf Interrupt im Userspace I

```
root@target: cat mygpio.c

int main (int argn, char* argv[], char* envp[])
{
    int fd, ret;
    struct pollfd pollfd0;
    char buf[100];

    fd = open ("/sys/class/gpio/gpio20/value", O_RDWR, 0);
    ret = read (fd, buf, sizeof(buf));
```

linuxhotel

Verzeichnis `/sys/class/gpio20` muß für das C-Programm bereits existieren, der GPIO auf Eingang und die Triggerung auf steigende/fallende Flanke eingerichtet sein (registriert Interrupt)
→ all dies wurde hier der einfachheit halber per Shell durchgeführt

10 mit `open()` wird Value-Datei geöffnet

11 es muß einmalig dummy-gelesen werden
→ Triggerung funktioniert sonst nicht

Warten auf Interrupt im Userspace II

```
    while (1)
20  {
        pollfd0.fd = fd;
        pollfd0.events = POLLPRI | POLLERR;
        ret = poll (&pollfd0, 1, 100000);
        if (ret > 0)
25      {
            if (pollfd0.revents)
                printf ("Event for GPIO-82");

            ret = lseek (fd, SEEK_SET, 0);
30          ret = read (fd, buf, sizeof(buf));
        }
        else if (ret == -1)
            break;
    }
35  close (fd);
    return 0;
}
```

linuxhotel

23 mit `poll()` wird maximal 100 Sekunden gewartet, bis eine fallende Flanke am Eingang mittels Interrupt registriert wird

29 es muß auf Position 0 mit `lseek()` positioniert und wiederholt mit `read()` gelesen werden

GPIO als Device-Node mit `ioctl()`

```
int ret, i, fd;
struct gpiohandle_request req;
struct gpiohandle_data data;

fd = open("/dev/gpiochip2", O_RDWR, 0);

req.flags = GPIOHANDLE_REQUEST_OUTPUT;
strcpy(req.consumer_label, "set-get-gpio");
memset(req.default_values, 0, sizeof(req.default_values));

req.lines = 1;
req.lineoffsets[0] = 18;
ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);

data.values[0] = 1;
ret = ioctl(req.fd,GPIOHANDLE_SET_LINE_VALUES_IOCTL,&data);
```

linuxhotel

Source: `gpiochip.c`

5 Device-Node `/dev/gpiochip2` wird geöffnet

12 im Beispiel wird lediglich ein einzelner GPIO mit der Nummer 18 bezogen auf den GPIO-Controller geschaltet

13 mit `ioctl(GPIO_GET_LINEHANDLE_IOCTL, ...)` wird ein Filedeskriptor für eine ganze Gruppe an GPIO's angefordert
→ spart Syscall-Aufrufe ein

16 `ioctl(GPIO_SET_LINE_VALUES_IOCTL, ...)` setzt den Wert des Ausganges im Beispiel auf eins

Warten auf Interrupt-Event aus dem Userspace heraus:
`linux/tools/gpio/gpio-event-mon.c`

Siehe auch:
`linux/tools/gpio`
`include/uapi/linux/gpio.h`
`cat /debug/gpio`

GPIO-Events als Device-Node mit `ioctl()`

```
int fd;
struct gpioevent_request req;
struct pollfd pollfd;
char buf[100];

fd = open("/dev/gpiochip0", O_RDWR, 0);
req.handleflags = GPIOHANDLE_REQUEST_INPUT;
req.eventflags = GPIOEVENT_EVENT_RISING_EDGE;
req.lineoffset = 20;
strncpy(req.consumer_label, "poll-gpio",
                            sizeof(req.consumer_label)-1);
ret = ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &req);

memset(&pollfd, 0, sizeof(struct pollfd));
pollfd.fd = req.fd; pollfd.events = POLLIN;
if (poll(&pollfd, 1, -1) >=1) {
    printf("Button pressed\n");
    lseek(req.fd, 0, SEEK_SET);
    read(req.fd, buf, sizeof(buf));      }
```

linuxhotel

Source: `gpioevent.c`

6 Device-Node `/dev/gpiochip0` wird geöffnet

7-10 Eigenschaften (Input, steigende Flanke, GPIO-Nr 20 und Bezeichnung) werden festgelegt

12 mit `ioctl(GPIO_GET_LINEEVENT_IOCTL, ...)` wird ein Filedeskriptor für einen GPIO-Event angefordert

16 mit `poll()` wird auf die steigende Flanke, über einen Interrupt bekanntgegeben, gewartet

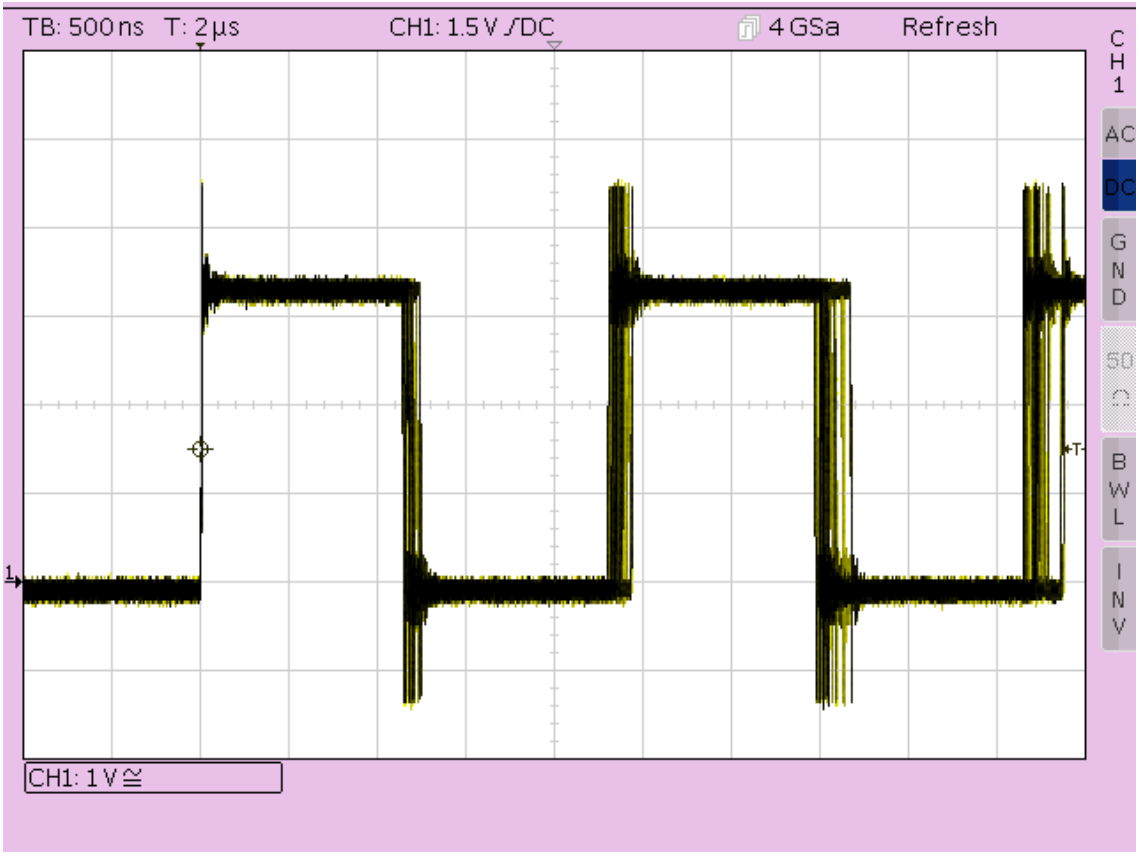
18,19 es muß nach jedem `poll()` dummy-gelesen werden, da ansonsten der nächste `poll()` ohne zu warten gleich zurückkehrt und nicht auf den Interrupt wartet

Vergleich von internen GPIO und GPIO-Expander

- TI-AM3354 (phyBoard-WEGA) mit 800 MHz
- linux-4.12 mit Preemption ohne RT-Patch
- Vergleich:
  - interner GPIO von SOC
  - GPIO-Expander PCA9555 mit I^2C-Anbindung von 400 kBit
- toggeln des GPIO mit max. Frequenz mittels
  - Kernel-Thread im Treiber (`gpio-kthread.c`)
  - `ioctl()` von `/dev/gpiochipN` im Userspace (`gpio-toggle.c`)
  - C-Programm verwendet sysfs (`gpio-sysfs.c`)
  - Shell-Skript verwendet sysfs-Schnittstelle (`gpio-expander.sh`)
- es wurden keine Worst-Case-Werte gemessen

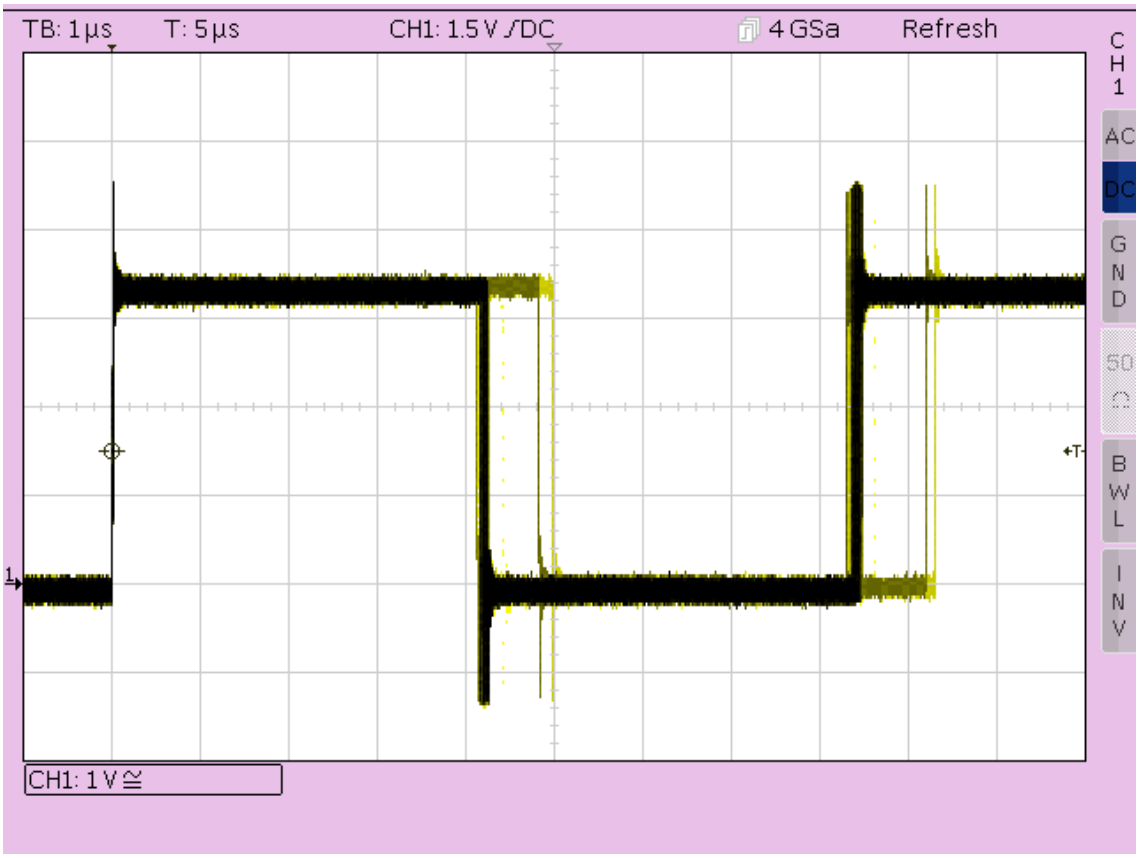
linuxhotel

GPIO im Treiber toggeln



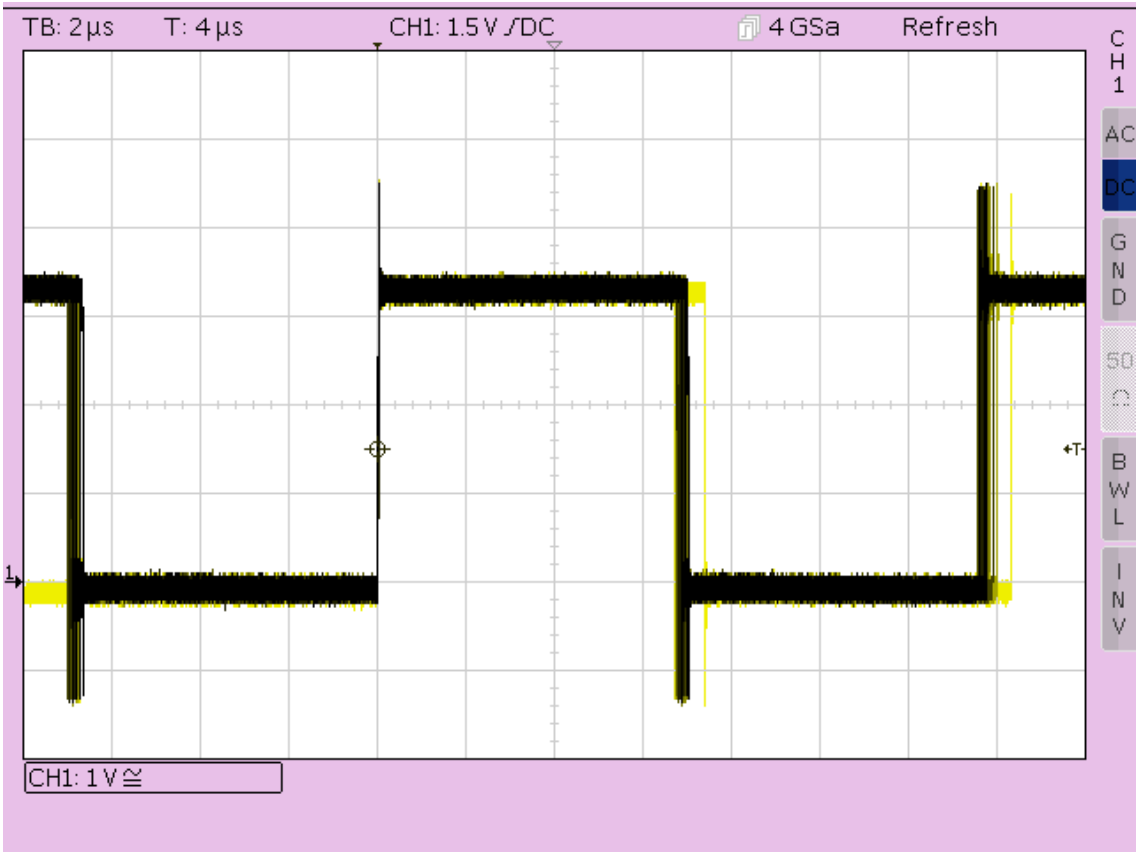
linuxhotel

GPIO im Userspace mit `ioctl()` toggeln



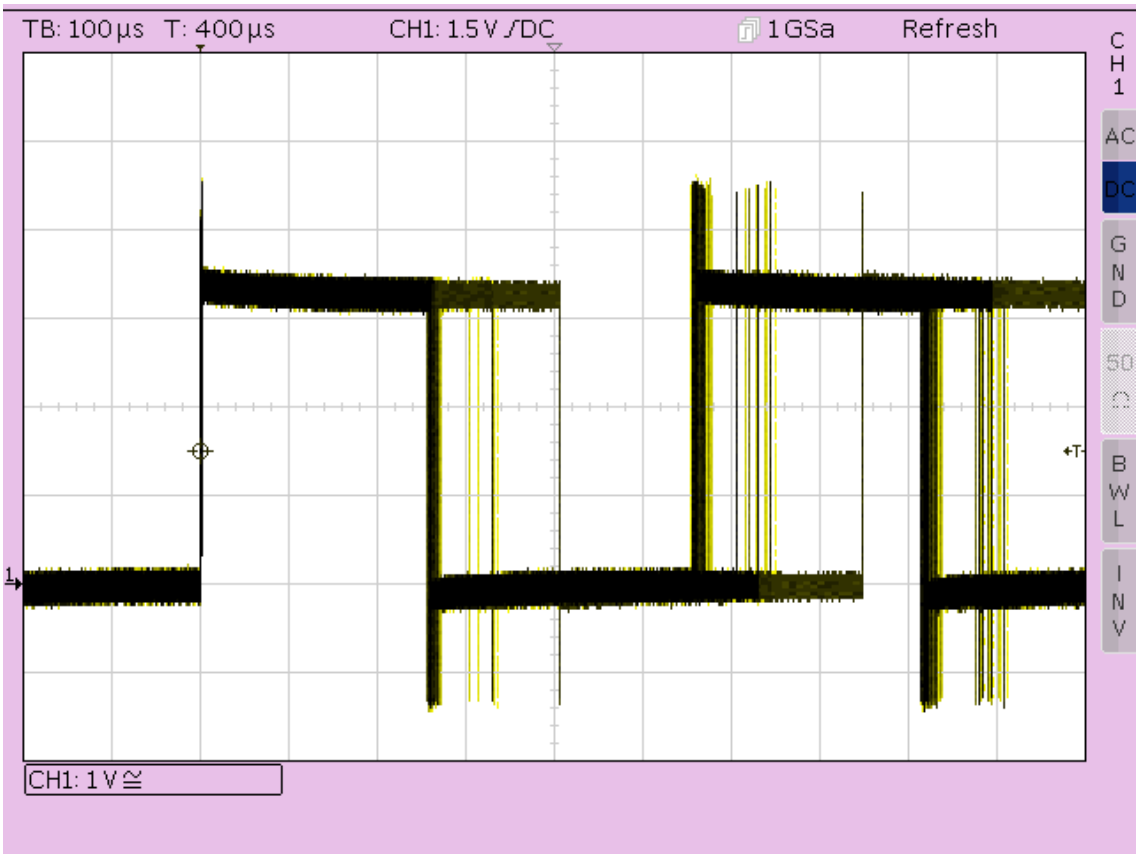
linuxhotel

GPIO im sysfs durch C-Programm toggeln



linuxhotel

GPIO im sysfs durch Shell toggeln



linuxhotel

Toggeln von GPIO auf SOC

- Kernel-Treiber: $1,2\ \mu s$
- `ioctl()`: $4,5\ \mu s$
- sysfs mit C-Programm: $7,0\ \mu s$
- sysfs mit Shell-Skript: $250 - 400\ \mu s$

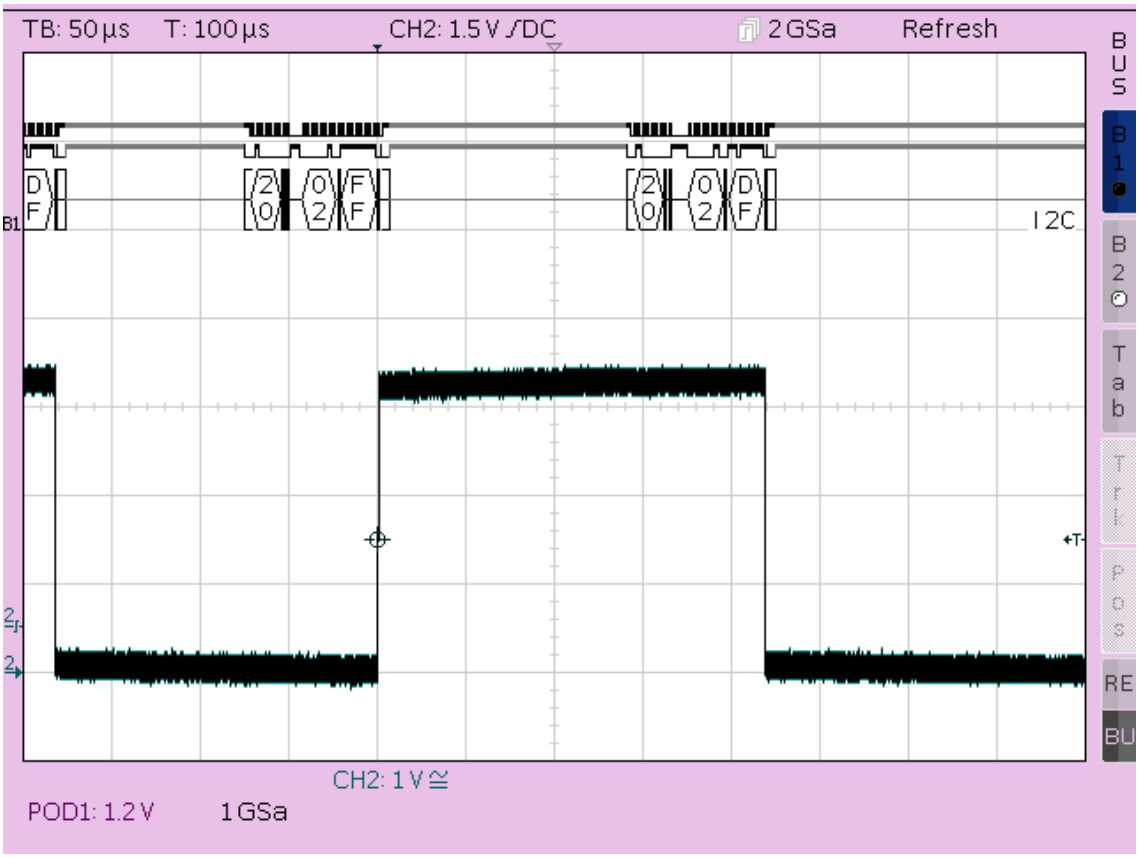
Wie lange dauert ein Syscall?

1.000.000 x Lesen eines Bytes und Schreiben desselben
$\rightarrow$ 2.000.000 Syscalls mit minimalster Rechenzeit (vernachlässigbar)

```
dd if=/dev/zero of=/dev/null bs=1 count=1000000
```

Ergebnis: ca. $3,5\ \mu s$ pro Syscall
$\Rightarrow$ `ioctl()`-Interface ohne signifikanten Overhead (siehe oben)

Toggeln von GPIO im Treiber auf I2C-Expander



linuxhotel

Bild Kernel-Treiber: 240 μs

- `ioctl()`: 250 μs
- sysfs mit C-Programm: 270 μs
- sysfs mit Shell-Skript: 550 μs

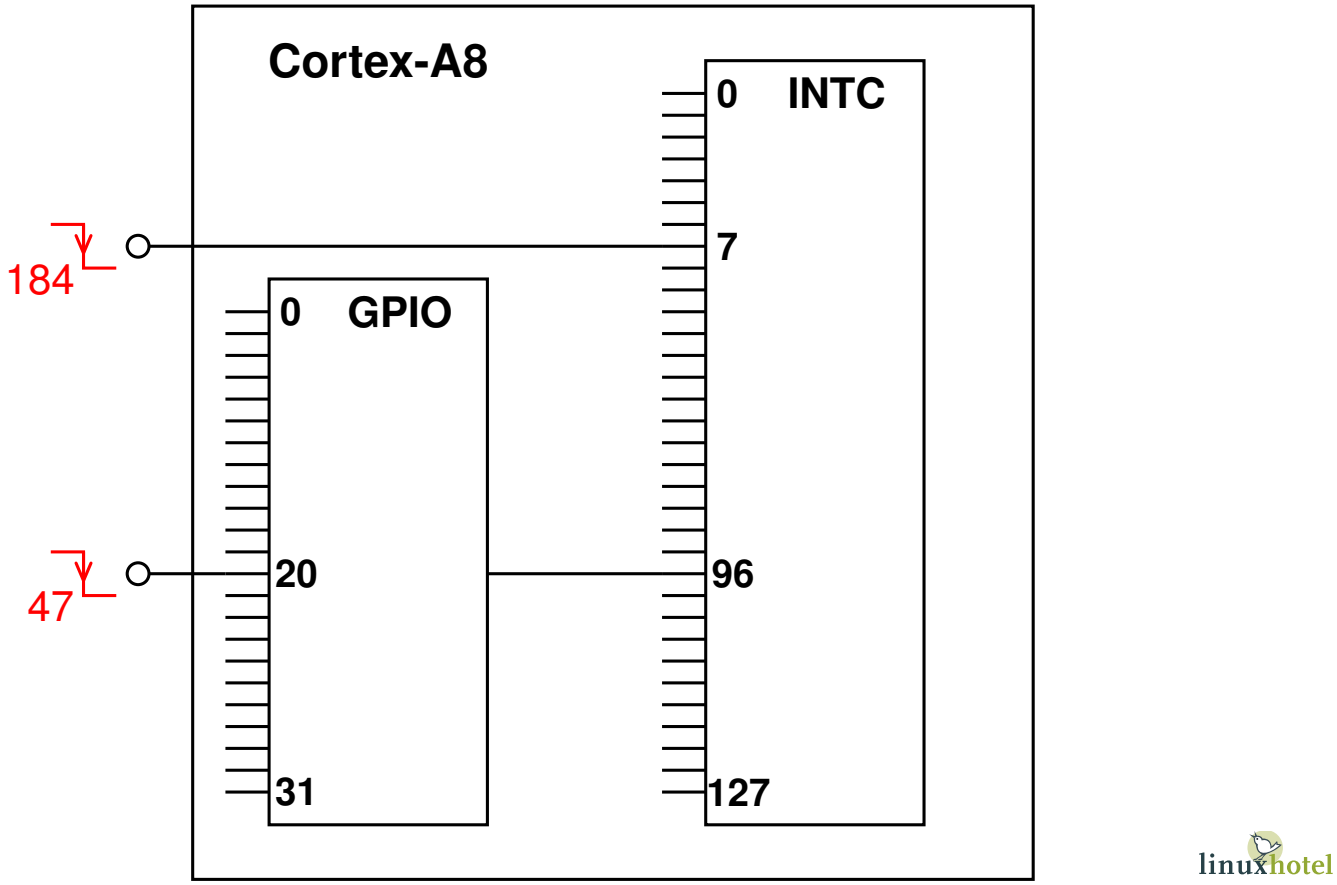
$\Rightarrow$ keine signifikanten Unterschiede feststellbar;
Overhead durch I^2C-Bus dominiert Geschwindigkeit

Vergleich von internem Interrupt und GPIO-Interrupt

- TI-AM3354 (phyBoard-WEGA) mit 800 MHz
- linux-4.12 mit Preemption ohne RT-Patch
- Vergleich:
  - interner Interrupt von SOC
  - GPIO als Interrupt, welcher wieder an internem Interrupt hängt
- es wurden keine Worst-Case-Werte gemessen

linuxhotel

GPIO-Interrupts bei Cortex-A8



- direkt am Interrupt-Controller wird Interrupt-Nummer 7 (bezogen auf Domäne des Interrupt-Controllers INTC) verwendet; im Linux-System (unter `/proc/interrupts`) hat er die Nummer 184
- GPIO-Nummer 20 wird auch als Interrupt-Eingang verwendet; in Linux hat dieser die Nummer 47
- ein GPIO-Controller (`gpio0`) nutzt die Interrupt-Nummer 96 vom Interrupt-Controller für alle seine Interrupts
  $\Rightarrow$ kaskadierte Interrupts

interner Interrupt-Controller — INTC

Ausschnitt aus dem Device-Tree:

```
myinterrupt {
    compatible = "itk,intr1";
    toggle-gpio = <&gpio0 7 GPIO_ACTIVE_HIGH>;
    interrupt-parent = <&intc>;
    interrupts = <7>;
};
```

Ausschnitt aus dem Kerneltreiber:

```
irqnr = irq_of_parse_and_map(dev->of_node, 0);

ret = devm_request_irq(dev, irqnr, intr1_handle_irq,
                                   0, pdev->name, data);
```

linuxhotel

GPIO-basierender Interrupt-Controller

Ausschnitt aus dem Device-Tree:

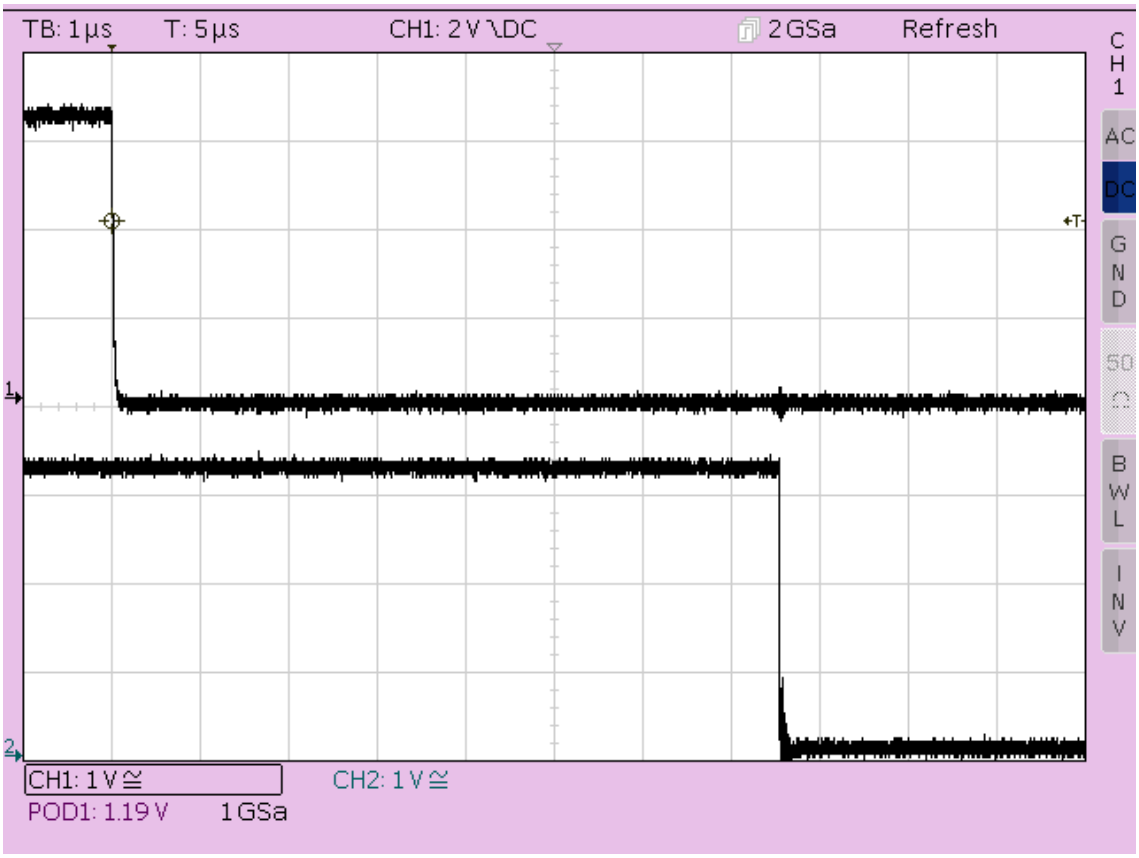
```
myinterrupt {
    compatible = "itk,intr1";
    toggle-gpio = <&gpio0 7 GPIO_ACTIVE_HIGH>;
    interrupt-parent = <&gpio0>;
    interrupts = <20 2>;
};
```

Ausschnitt aus dem Kerneltreiber:

```
irqnr = irq_of_parse_and_map(dev->of_node, 0);

ret = devm_request_irq(dev, irqnr, intr1_handle_irq,
                                   0, pdev->name, data);
```

linuxhotel

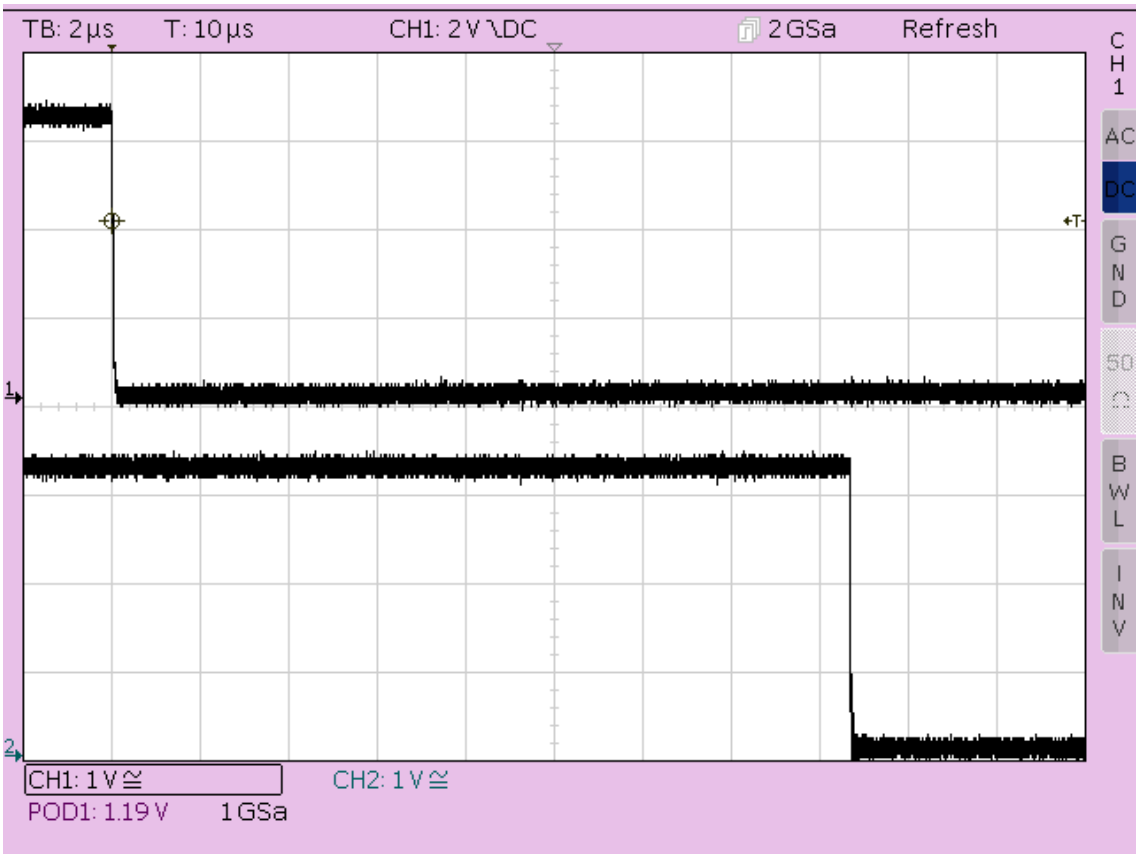
Interrupt toggelt GPIO — Variante INTC



linuxhotel

- interner Interrupt von Cortex-A8
- Interrupt wird an das System angelegt
- in der ISR wird ein GPIO getoggelt
- gemessene Latenz: $6 - 16\ \mu s$

Interrupt toggelt GPIO — Variante GPIO

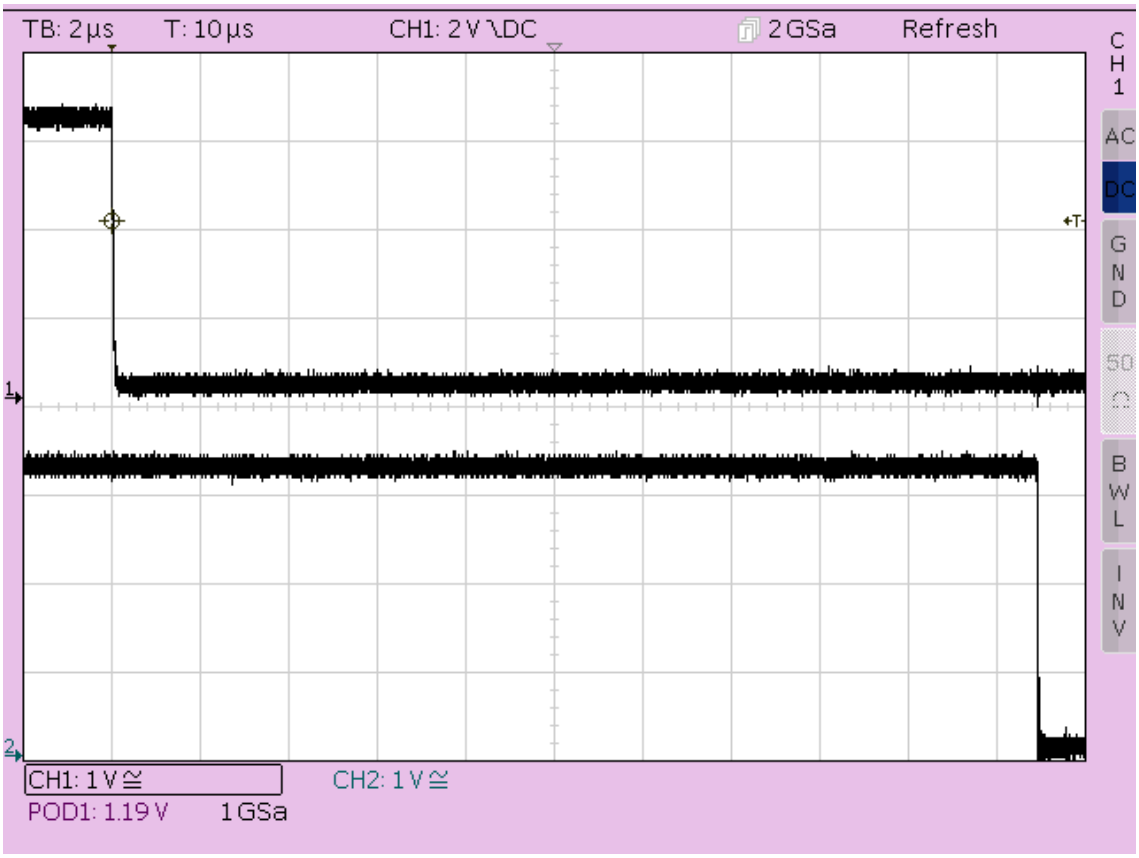


linuxhotel

- GPIO-basierender Interrupt
- gemessene Latenz: 12 – 20 μs
- Diagnose:

```
trace-cmd record -e 'irq:*' -e 'gpio:*'          \
                                ./gpioevent 1

trace-cmd report
```

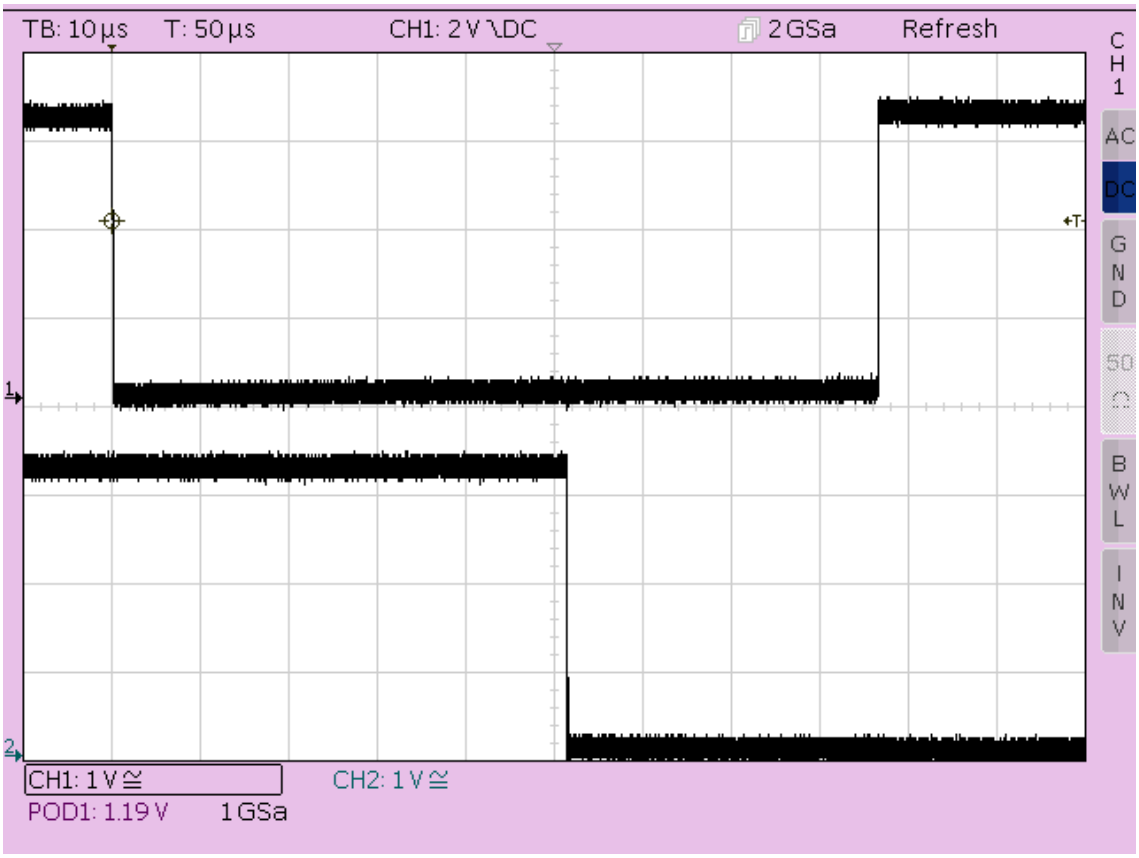
Interrupt toggelt GPIO — INTC und threadirqs



linuxhotel

- interner Interrupt
- Kernel-Kommandozeile enthält: `threadirqs`
  $\Rightarrow$ ISR wird durch Kernelthread ausgeführt
- gemessene Latenz: $20 - 30\ \mu s$

Interrupt toggelt GPIO — GPIO und threadirqs



linuxhotel

- GPIO-basierender Interrupt
- Kernel-Kommandozeile enthält: `threadirqs`
  ⇒ ISR wird durch Kernelthread ausgeführt
- es werden in der Default-Konfiguration zwei Kernelthreads hintereinander ausgeführt
  ⇒ mit dem Flag `IRQF_NO_THREAD` bei der Registrierung des Interrupts im Kerneltreiber änderbar
- gemessene Latenz: $45 - 70\ \mu s$

Interrupts im procfs

```
root@target: cat /proc/interrupts
        CPU0
 16:   21453      INTC  68 Level     gp_timer
 18:       0      INTC   3 Level     arm-pmu
 20:     760      INTC  12 Level     49000000.edma_ccint
 22:       8      INTC  14 Level     49000000.edma_ccerrint
 26:       0      INTC  96 Level     44e07000.gpio
 33:       0  44e07000.gpio   6 Edge      48060000.mmc cd
 59:       0      INTC  98 Level     4804c000.gpio
 92:       0      INTC  32 Level     481ac000.gpio
125:       0      INTC  62 Level     481ae000.gpio
158:     390      INTC  72 Level     44e09000.serial
160:     306      INTC  70 Level     44e0b000.i2c

[...]

184:   23874      INTC   7 Level     ocp:myinterrupt
```

linuxhotel

Anzeige der in Linux registrierten Interrupts

Spalte 1 Interrupt-Nummer im Betriebssystem Linux

Spalte 2 Anzahl der bisher aufgetretenen Interrupts

Spalte 3 Name des Interrupt-Controllers

Spalte 4 Interrupt-Nummer bezogen auf Domäne des Interrupt-Controllers

Spalte 5 Level- oder Flankentriggerung

Spalte 6 Name der registrierten Interrupt-Service-Routine

16 Beispieltreiber verwendet Interrupt des internen Interrupt-Controllers (INTC) mit der Nummer 7

Interrupts im procfs

```
root@target: cat /proc/interrupts
           CPU0
 16:        2159      INTC  68 Level     gp_timer
 18:           0      INTC   3 Level     arm-pmu
 20:         747      INTC  12 Level     49000000.edma_ccint
 22:           7      INTC  14 Level     49000000.edma_ccerrint
 26:          70      INTC  96 Level     44e07000.gpio
 33:           0  44e07000.gpio   6 Edge      48060000.mmc cd
 47:          70  44e07000.gpio  20 Edge      ocp:myinterrupt
 59:           0      INTC  98 Level     4804c000.gpio
 92:           0      INTC  32 Level     481ac000.gpio
125:           0      INTC  62 Level     481ae000.gpio
158:         304      INTC  72 Level     44e09000.serial
160:         306      INTC  70 Level     44e0b000.i2c

[...]
```

linuxhotel

6 GPIO-Controller ist zugleich auch ein Interrupt-Controller und hängt am internen Interrupt Nummer 96 (vgl. Device-Tree)

8 Beispieltreiber verwendet Interrupt des GPIO-Controllers (44e07000.gpio) mit der Nummer 20

Ergebnis

- bei GPIO-basierendem Interrupt wird zunächst IRQ vom GPIO-Controller ausgelöst; anschließend wird Interrupt-Behandlung des internen Interrupt-Controllers ausgeführt
  → Unterschied am Beispielsystem ca. Faktor 2
- bei threaded IRQ's wird für jeden Interrupt ein Kernelthread aufgeweckt und im Schedulingverfahren ausgeführt
  → Unterschied ca. Faktor 2 - 3
- bei GPIO-basierendem Interrupt und threaded IRQ's werden zwei Kernelthreads nacheinander ausgeführt
  → nocheinmal ca. Faktor 2 langsamer
- nachweisbar mit dem ftrace-Framework
- mit dem Flag `IRQF_NO_THREAD` kann die Generierung des Kernelthreads für laufzeitkritische Interrupts unterbunden werden

I^2C — Inter-Integrated-Circuit I

- serieller Bus mit Clock und Daten
- Adressierung mit 7- oder 10-Bit
- bei 7-Bit-Adressierung wird das 8. Bit der Slave-Adresse als RW-Kennung verwendet:
  0 Schreiben
  1 Lesen
- z. B. Geräteadresse $0x20$
  $\Rightarrow 0x40$ Slaveadresse für Schreiben
  $\Rightarrow 0x41$ Slaveadresse für Lesen
- Geschwindigkeit meist 100 kHz oder 400 kHz
  max. 3,4 MHz sind möglich

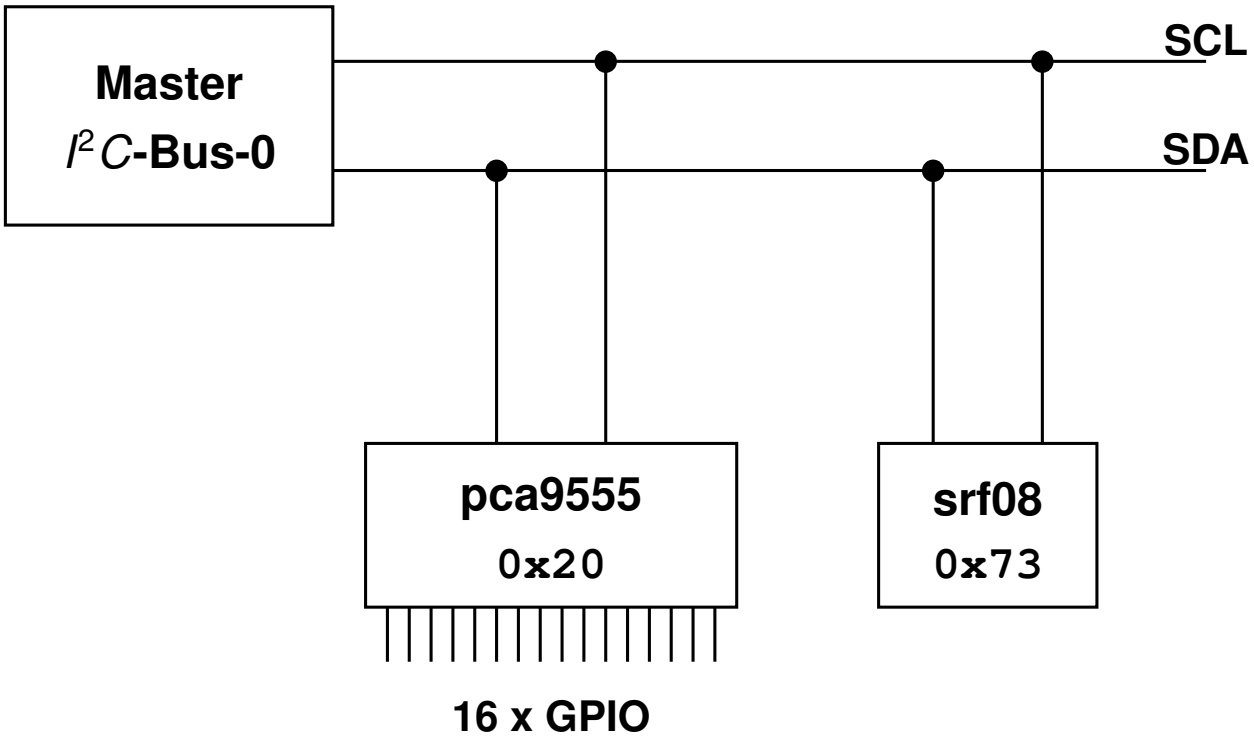
linuxhotel

I^2C — Inter-Integrated-Circuit II

- SMBus implementiert ein Subset von I^2C
  $\rightarrow$ viele, aber nicht alle Geräte sind kompatibel
- Siehe auch:
  `Documentation/i2c/`

linuxhotel

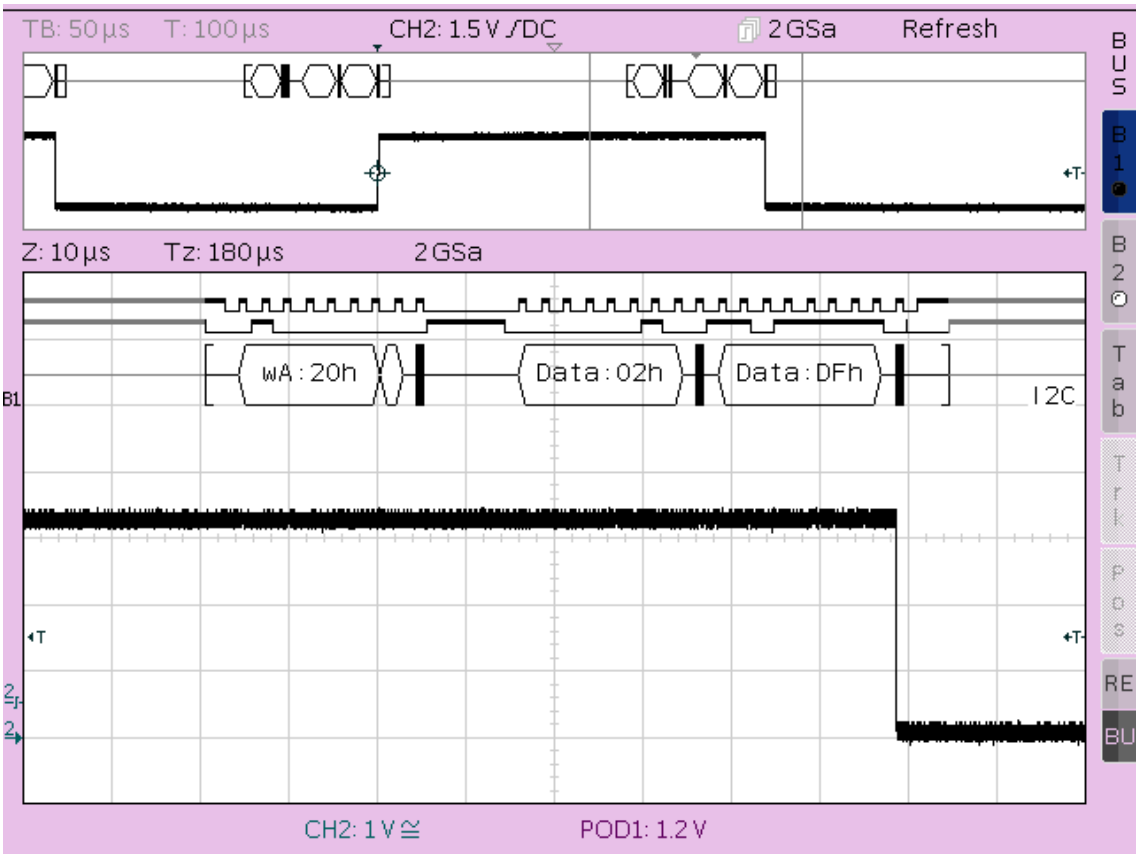
I^2C-Bus



linuxhotel

- beim I^2C-Bus gibt es einen Master und viele Slaves
- jeder Slave hat eine (7-Bit) Busadresse
- wenn Master Daten zu den Slaves schreiben möchte, sendet er die entsprechende Busadresse zusammen mit dem Schreiben-Bit (`0x40`)
- wenn Master Daten vom einem Slave lesen möchte, sendet er das Lese-Bit (`0x41`) und liest die Daten, welche der Slave schreibt

I^2C-Telegrammverkehr



linuxhotel

Aufzeichnung im ftrace

```
echo 'gpio_value i2c:*' > /debug/tracing/set_event

cat /debug/tracing/trace | cut -c 35-

[...]

16472.079819: smbus_write: i2c-0 a=020 f=0000 c=2 BYTE_DATA l=1 [ff]
16472.079846: i2c_write: i2c-0 #0 a=020 f=0000 l=2 [02-ff]
16472.080017: i2c_result: i2c-0 n=1 ret=1
16472.080022: smbus_result: i2c-0 a=020 f=0000 c=2 BYTE_DATA wr res=0
16472.080026: gpio_value: 499 set 0
16472.080029: smbus_write: i2c-0 a=020 f=0000 c=2 BYTE_DATA l=1 [df]
16472.080050: i2c_write: i2c-0 #0 a=020 f=0000 l=2 [02-df]
16472.080211: i2c_result: i2c-0 n=1 ret=1
16472.080216: smbus_result: i2c-0 a=020 f=0000 c=2 BYTE_DATA wr res=0
16472.080220: gpio_value: 499 set 1
```

linuxhotel

I^2C instantiieren im Device-Tree

```
i2c0: i2c@44e0b000 {
    compatible = "ti,omap4-i2c";
    reg = <0x44e0b000 0x1000>;
    clock-frequency = <400000>;
    #address-cells = <1>;
    #size-cells = <0>;
    [...]
};

&i2c0 {
    distance@70 {
        compatible = "devantech,srf08";
        reg = <0x70>;
    };
};
```

linuxhotel

1 Node `i2c@44e0b000` mit Alias `i2c0` definiert den Treiber für den I^2C-Bus-0 sowie dessen Einstellungen

4 `clock-frequency` stellt die Geschwindigkeit auf 400 kHz ein

5,6 `#address-cells` gibt an, daß die I^2C-Adresse innerhalb des I^2C-Busses mit einem 32-Bit-Wert im Setting `reg` der Subnodes angegeben werden; es gibt keine Größenangabe im Setting `reg`, da `#size-cells` = 0

11 Node `distance@70` definiert das I^2C-Gerät auf dem I^2C-Bus-0

13 Setting `reg` stellt die Busadresse auf `0x70` ein

I^2C instantiieren im sysfs

```
echo srf08 0x70 > /sys/bus/i2c/devices/i2c-0/new_device
```

```
echo 0x70 > /sys/bus/i2c/devices/i2c-0/delete_device
```

linuxhotel

1 I^2C-Gerät mit dem Namen `srf08` und der Busadresse `0x70` wird am I^2C-Bus-0 als neues Gerät angemeldet

4 Name ist derjenige, welche in der Treiberregistrierung als I^2C-Gerät in `i2c_device_id` verwendet wurde

Verfahren ist geeignet für Test- und Entwicklungszwecke

beim Abmelden reicht die Busadresse, da diese bereits eindeutig ist

I^2C-Treiber am Subsystem anmelden I

```
enum srf08_sensor_type {
    SRF02,
    SRF08,
    SRF_MAX_TYPE
};

static const struct of_device_id of_srf08_match[] = {
    { .compatible = "devantech,srf02", (void *)SRF02},
    { .compatible = "devantech,srf08", (void *)SRF08},
    {},
};
MODULE_DEVICE_TABLE(of, of_srf08_match);
```

linuxhotel

1 `enum srf08_sensor_type` dient der Aufzählung der verwendeten Sensortypen

7 `struct of_device_id` listet die unterstützten Compatible-Strings zusammen mit der Nummerierung auf

12 mit dem Makro `MODULE_DEVICE_TABLE` wir eine Tabelle der unterstützen Device-Tree-Treiber erstellt und bekannt gemacht

I^2C-Treiber am Subsystem anmelden II

```
static const struct i2c_device_id srf08_id[] = {
    { "srf02", SRF02 },
    { "srf08", SRF08 },
    { }
};
MODULE_DEVICE_TABLE(i2c, srf08_id);

static struct i2c_driver srf08_driver = {
    .driver = {
        .name           = "srf08",
        .of_match_table = of_srf08_match,
    },
    .probe = srf08_probe,
    .id_table = srf08_id,
};

module_i2c_driver(srf08_driver);
```

linuxhotel

24 `MODULE_DEVICE_TABLE` legt als Makro einen Eintrag in der Device-Tabelle des Moduls an:

```
root@host: modinfo srf08.ko
[...]
alias: i2c:srf08
```

31 Funktion `probe()` dient als Einsprung in den Treiber, sobald im Device-Tree der Compatible-String gematched wird

35 Makro `module_i2c_driver()` beinhaltet `module_init()`, `module_exit()` und Registrierung von I^2C-Driver
→ erspart es den immer wieder gleichen `init()`- und `exit()`-Code zu schreiben

probe() im I^2C-Device

```
static int srf08_probe(struct i2c_client *client,
                       const struct i2c_device_id *id)
{
    if (!i2c_check_functionality(client->adapter,
                    I2C_FUNC_SMBUS_READ_BYTE_DATA |
                    I2C_FUNC_SMBUS_WRITE_BYTE_DATA |
                    I2C_FUNC_SMBUS_READ_WORD_DATA))
        return -ENODEV;

    [...]
}
```

linuxhotel

1 probe()-Funktion wird Referenz auf i2c_client als I^2C-Treiberobjekt sowie i2c_device_id des hinzugefügten Devices übergeben
→ aus i2c_device_id kann mit dem Data-Wert ermittelt werden, welches I^2C-Gerät genau hinzugefügt wurde, wenn Treiber mehrere I^2C-Device-Ids unterstützt

4 mit i2c_check_functionality kann geprüft werden, welche I^2C-Features von dem betreffenden I^2C-Bus unterstützt werden

Lesen und Schreiben von I^2C-Device

```
int ret;
struct i2c_client *client = data->client;

ret = i2c_smbus_write_byte_data(client, 0x00, 0x51);
if (ret < 0) {
    dev_err(&client->dev, "write - err: %d\n", ret);
    return ret;
}

ret = i2c_smbus_read_byte_data(data->client, 0x07);
if (ret < 0) {
    dev_err(&client->dev, "read - err: %d\n", ret);
    return ret;
}
```

linuxhotel

4 `i2c_smbus_write_byte_data()` schreibt auf das Register `0x00` den Wert `0x51`

5 Rückgabewert < 0 bedeutet Fehler

10 `i2c_smbus_read_byte_data()` liest vom Register `0x07`

11 Rückgabewert < 0 bedeutet Fehler

I^2C im Userspace

- mit dem Kernel-Modul `i2c-dev` gelangt man zu Character-Device-Nodes, welche den Zugriff auf I^2C vom Userspace aus ermöglichen
  `/dev/i2c-0`
  `/dev/i2c-1`
  `...`
- Paket `i2ctools` liefert Hilfsprogramme für I^2C:

  `i2cdetect`:
  Scannen von I^2C-Bus nach Devices

  `i2cdump, i2cget, i2cset`:
  Lesen und Schreiben von I^2C-Registern

linuxhotel

I^2C-Device mit i2c-dev verwenden

```
int fd;
char filename[20];
int addr = 0x0018;          /* I2C address: lower 7-Bit */
unsigned char buf[10];

sprintf(filename, "/dev/i2c-0");
if ((fd = open(filename, O_RDWR)) < 0)
    error(1, errno, "open()");

if (ioctl(fd, I2C_SLAVE, addr) < 0)
    error(2, errno, "ioctl()");

buf[0] = 0x05;
if (write (fd, buf, 1) < 0)
    error(3, errno, "write()");

if (read (fd, buf, 3) < 0)
    error(4, errno, "read()");
```

linuxhotel

3 I^2C-Adresse ist hier $0x18$

7 Öffnen des Device-Nodes zum Character-Device des Treibers i2c-dev

10 Einstellen der I^2C-Adresse des Gerätes am Bus

14,17 mit `read()`- und `write()`-Aufrufen kann vom Bus gelesen und geschrieben werden

SPI I

- serieller Bus mit Master-Slave-Konfiguration
- jedes Device hat eigenen Chip-Select-Eingang
- wird als 4-Draht-Bus immer vollduplex betrieben:
  - Chip-Select (CS)
  - Clock
  - Master-Out-Slave-In (MOSI)
  - Master-In-Slave-Out (MISO)
- wird als 3-Draht-Bus halbduplex betrieben:
  - Chip-Select (CS)

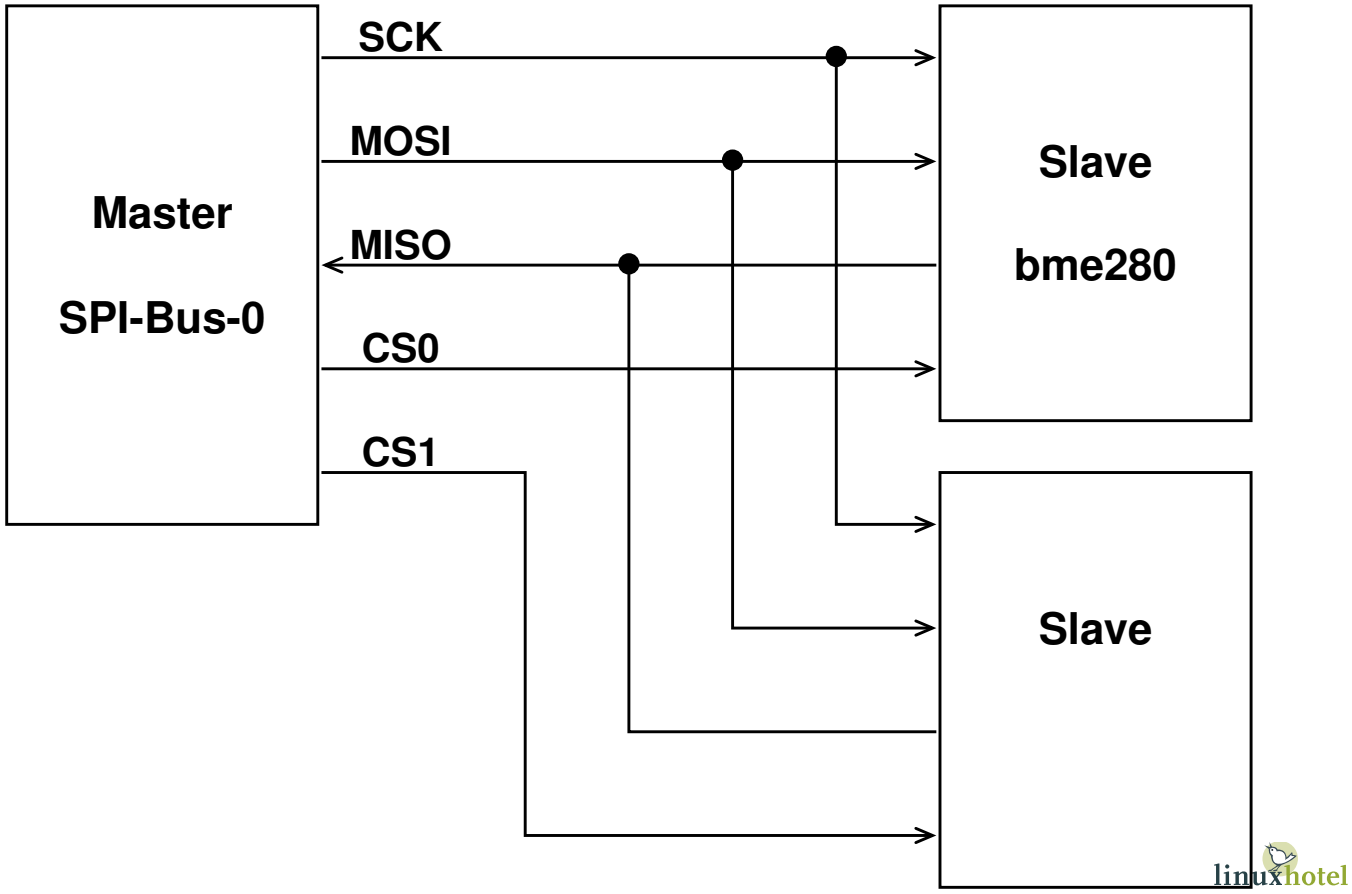
linuxhotel

SPI II

  - Clock
  - Data (bidirektional)
- Geschwindigkeit nicht festgelegt; > 1 MHz sind üblich
- schneller als I^2C

linuxhotel

SPI-Bus



- beim SPI-Bus gibt es einen Master der den Clock und den Chip-Select (in der Zeichnung CS0 oder CS1) vorgibt
- Übertragung erfolgt im Duplexmodus was bedeutet, daß sowohl vom Master zum ausgewählten Slave (MOSI) als auch in der Gegenrichtung (MISO) Daten übertragen werden
- bei nicht benötigten Datenbits werden Dummydaten übertragen; z. B. bei Frage-Antwort-Spiel wenn Master den Slave nach einem Meßwert frägt

SPI im Device-Tree

```
&spi0 {
    bme280: bme280 {
        compatible = "bosch,bme280";
        spi-max-frequency = <500000>;
        reg = <0x0>;
        status = "okay";
    };
};
```

linuxhotel

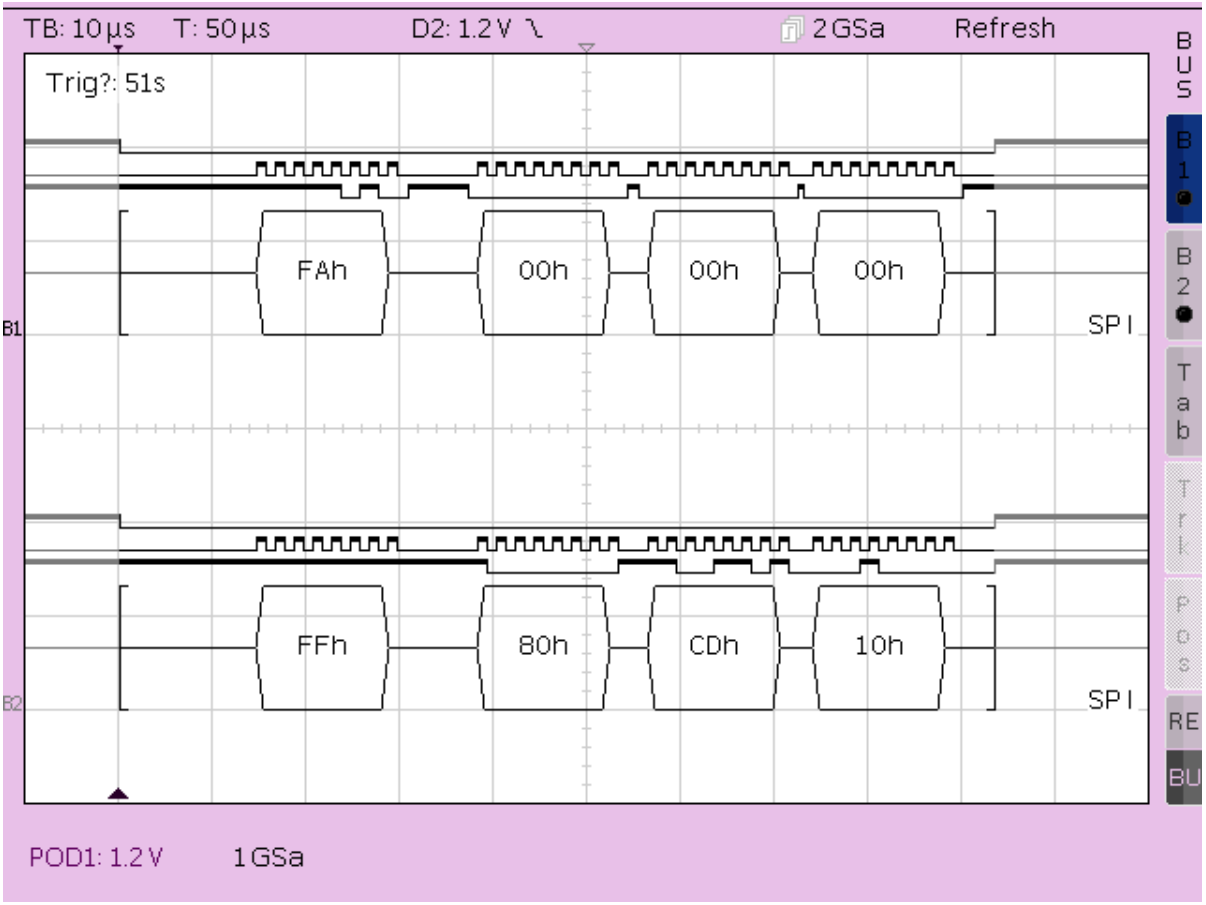
1 Referenzierung des 1. SPI-Busses

3 *compatible*
Angabe des Treibers für das Gerät

4 max. Geschwindigkeit

5 Chip-Select auf dem SPI-Bus

4-Draht-SPI-Datenaustausch mit BME-280



- pro Bus kann nur eine Datenleitung aufgezeichnet werden
  ⇒ zwei Busse (B1 und B2) definieren

Bus 1 Chip-Select (CS), Clock und Master-Out-Slave-In (MOSI)
→ Abfrage des Temperaturwertes (Register `0xFA`) vom Sensor

Bus 2 Chip-Select (CS), Clock und Master-In-Slave-Out (MISO)
→ Sensor liefert Temperaturwert (3 Byte) zurück

- Kommunikation läuft beim 4-Draht-Bus immer duplex
  → Dummydaten wenn nichts zu senden

spidev-Treiber im Device-Tree

Treiber spidev ist für den Test der SPI-Schnittstelle aus dem Userspace heraus gedacht

```
&spi0 {
    spidev0: spidev0 {
        compatible = "rohm,dh2228fv";
        spi-max-frequency = <500000>;
        reg = <0x0>;
        status = "okay";
    };
};
```

linuxhotel

1 Referenzierung des 1. SPI-Busses

3 Angabe des Treibers für das Gerät
der hier angegebene Treiber wird gar nicht verwendet, aber es ist ein Treiber aus der Compatible-Liste des spidev-Treibers zu wählen

seit v4.1 wird gewarnt (`WARN_ON`-Makro), wenn „`spidev`“ als Compatible im Device-Tree verwendet wird

Hintergrund:
spidev ist eine Software-Konfiguration und hat daher im Device-Tree nichts verloren
Kommentar:
eine nicht verwendete Hardware als Dummy anzugeben ist auch nicht schöner

4 max. Geschwindigkeit

5 Chip-Select auf dem SPI-Bus

spidev-Treiber mit Testprogramm nutzen

im Verzeichnis `tools/spi` der Kernel-Sourcen befindet sich Testprogramm für den Zugriff auf SPI aus dem Userspace heraus

```
spidev_test -D /dev/spidev1.0 -p "\xFA\x00\x00\x00"

spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
```

```
spidev_test --help
```

linuxhotel

Industrial-IO — Um was geht es?

- Vernetzung von Geräten wird zur Gewinnung von Informationen betrieben
- Abstraktion des Sensor- und Aktorinterfaces für den Userspace
- Sensoren liefern Meßwerte aus der Umgebung
- Implementierung als Linux-Kernel-Treiber
- schnelle Erfassung der Daten
- Nutzung kann buffered mit Device-Node, im sysfs oder durch Bibliotheken erfolgen

linuxhotel

Anwendungsfall: Bienenwaage

Messung des Gewichtes mit Waage

- Messung des Futterverbrauches (Honig)
  → hungert das Volk?
- eingelagerten Honig messen
  → wann kann der Imker schleudern?
- Bienenschwarm erkennen
  → Volk teilt sich und zieht aus
- Räuberei
  → ein Volk bricht bei einem anderen ein und klaut dort den fertigen Honig

Messung der Temperatur

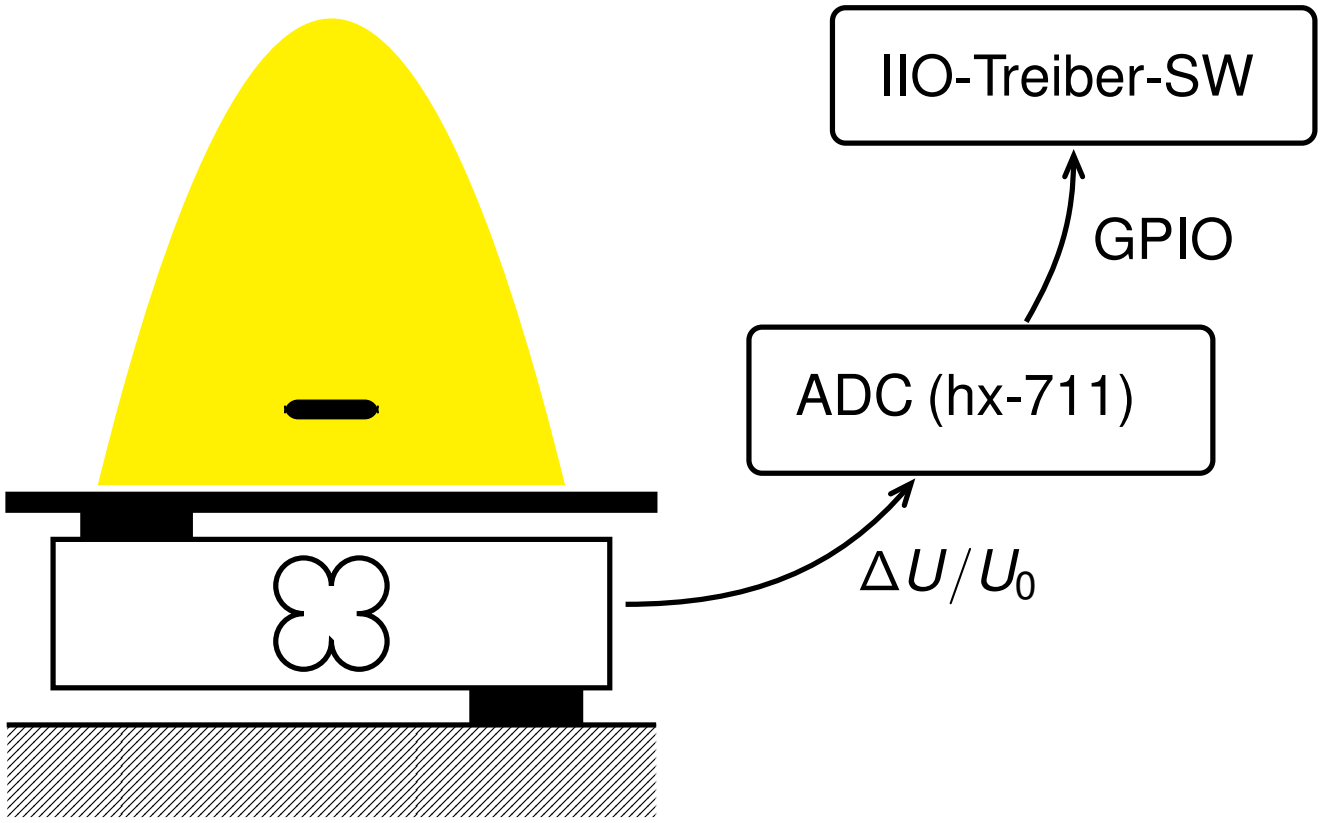
→ Flugaktivität ab 12°C

Messung des Luftdrucks

→ Bienen sind aggresiv bei Gewitterneigung (fallender Luftdruck)

linuxhotel

Beispiel: Bienenwaage mit IIO-Sensoren

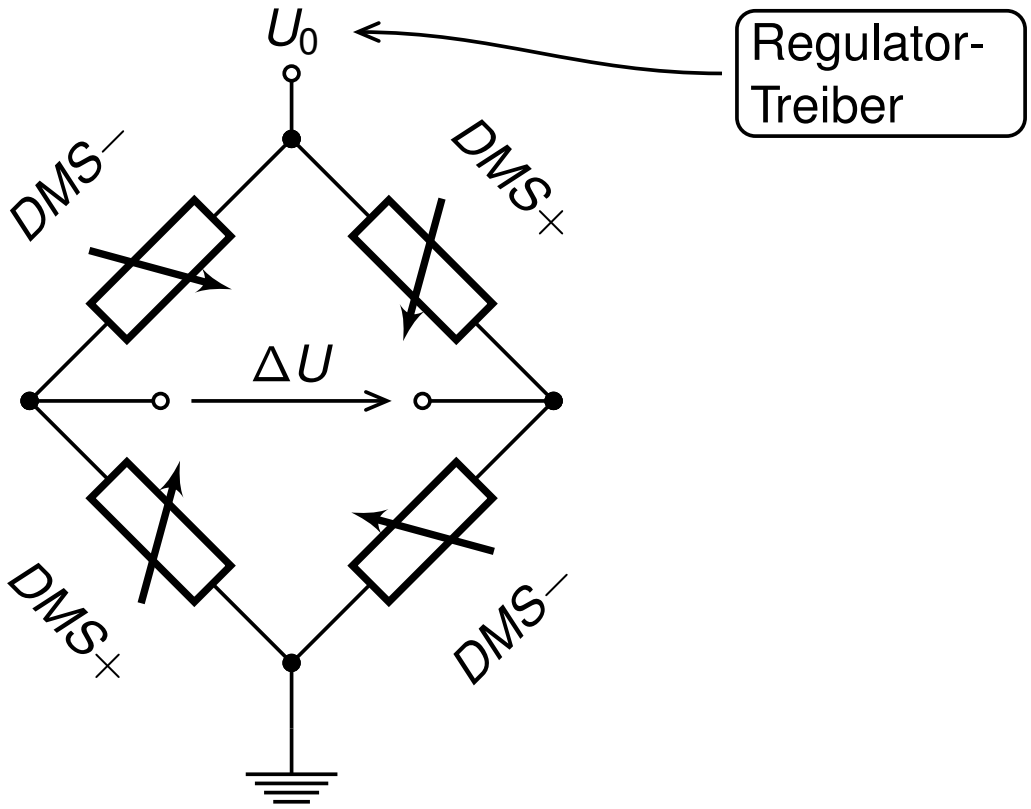


linuxhotel

- eine Wägezelle besteht aus Dehnungsmeßstreifen, welche bei einer Gewichtsänderung eine proportionale Widerstandsänderung liefern. Diese Widerstände werden zu einer Meßbrücke zusammengeschlossen und eine Spannung U_0 angelegt.
  Die Brückenspannung ΔU ist proportional zum Gewicht auf der Waage.
- ein Analog-Digital-Wandler (ADC), hier Typ hx-711, mißt die Brückenspannung und kann selber mit GPIO-Bitbanging abgefragt werden.
- IIO-Treiber übernimmt das Abfrageprotokoll und liefert gemessene Spannung im IIO-Framework.
- Spannungswert kann im sysfs abgefragt werden:

Verzeichnis	`/sys/bus/iio/devices/iio:deviceN/`
Rohwert (Ganzzahl)	`in_voltage0_raw, in_voltage1_raw`
Skalierung (Fließkomma)	`in_voltage0_scale, ...`

Wägezelle

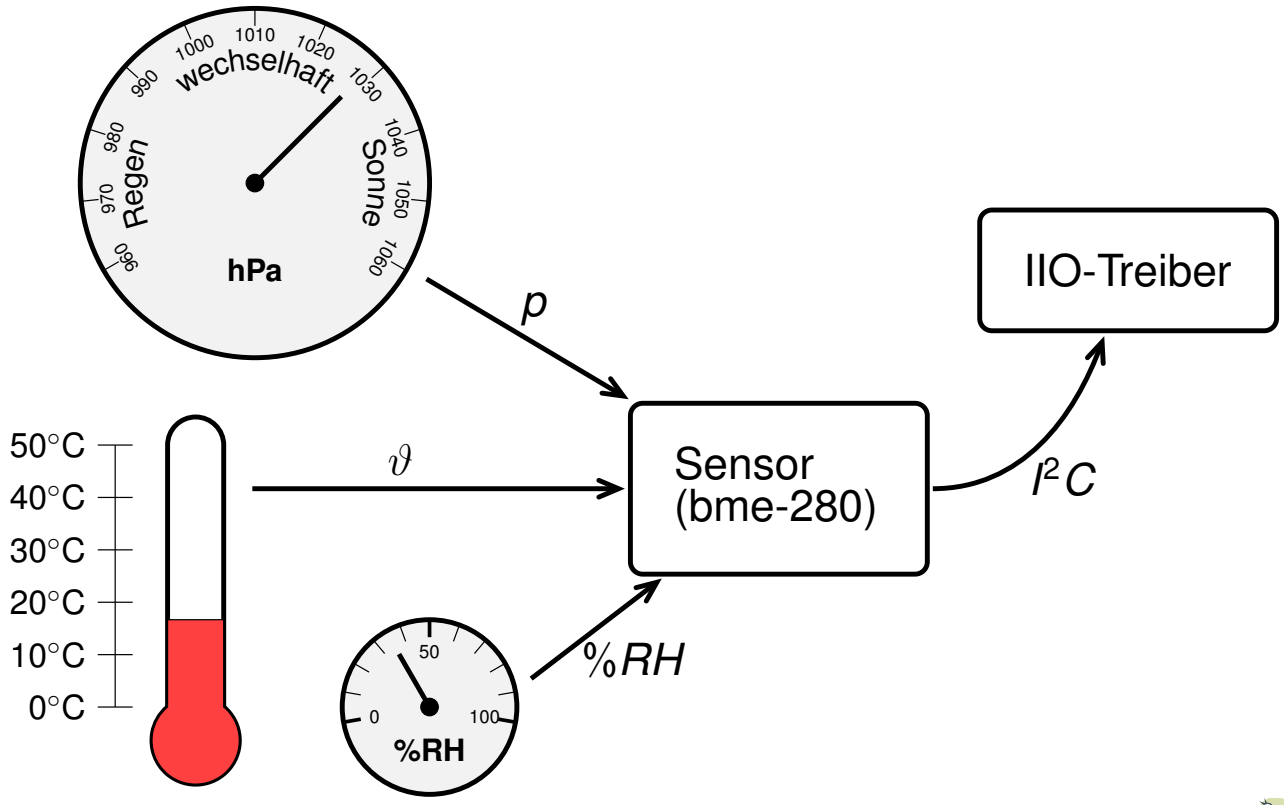


linuxhotel

Ersatzschaltbild der Wägezelle

- Wägezelle besteht aus Dehnungsmeßstreifen (DMS) welche sich nach außen hin wie eine Brückenschaltung darstellen
- Spannung U_0 wird angelegt und DMS verändern ihren Widerstandswert proportional zur Verformung
- gestauchte und gedehnte DMS bei einem Gewicht sind jeweils gegeneinander in der Brücke angeordnet, so daß die Brückenspannung ΔU möglichst groß wird
- Brückenspannung ΔU wird gemessen und als direkt proportional zum belasteten Gewicht angenommen

Beispiel: Umweltdatenmessung mit IIO-Sensoren



linuxhotel

- Messung von Umweltdaten, wie Temperatur θ, Luftdruck p, relative Luftfeuchtigkeit $\%RH$ mit einem Sensor, hier Modell BME-280
- Daten können mithilfe von I^2C vom Sensor abgefragt werden
- IIO-Treiber frägt Daten vom Sensor ab und hängt diese im IIO-Framework ein, wodurch sie im sysfs sichtbar werden.

Verzeichnis	`/sys/bus/iio/devices/iio:deviceN/`
Temperatur	`in_temp_input`
Feuchtigkeit	`in_humidityrelative_input`
Luftdruck	`in_pressure_input`

Sensor implementiert als Character Device I

- Device-Node:
  `/dev/waage`
  → Abfrage der Masse
  → Treiber kommuniziert mit Hardware
- mehrere Kanäle:
  `/dev/waage0`
  `/dev/waage1`
  `...`
- Eingangsverstärker im Sensor:
  `ioctl()` zum Setzen der Verstärkung

linuxhotel

Sensor implementiert als Character Device II

- Darstellung als $f(t)$
  → Userspace-Timer triggert Abfrage
  oder
  → Kernel-Timer triggert und schreibt in eigenen Buffer
- externer Event (Interrupt) triggert Aufzeichnung
  → Interrupt-Handler und `poll()` zum Warten aus dem Userspace heraus
- Event bei Grenzwertüberschreitung
  → `ioctl()` oder `poll()`

linuxhotel

Sensor implementiert als Character Device III

Sensoren als Character Device

jedes Feature muß jedesmal erneut implementiert werden

jeder Sensor hat sein eigenes Interface (viele `ioctl()`'s)

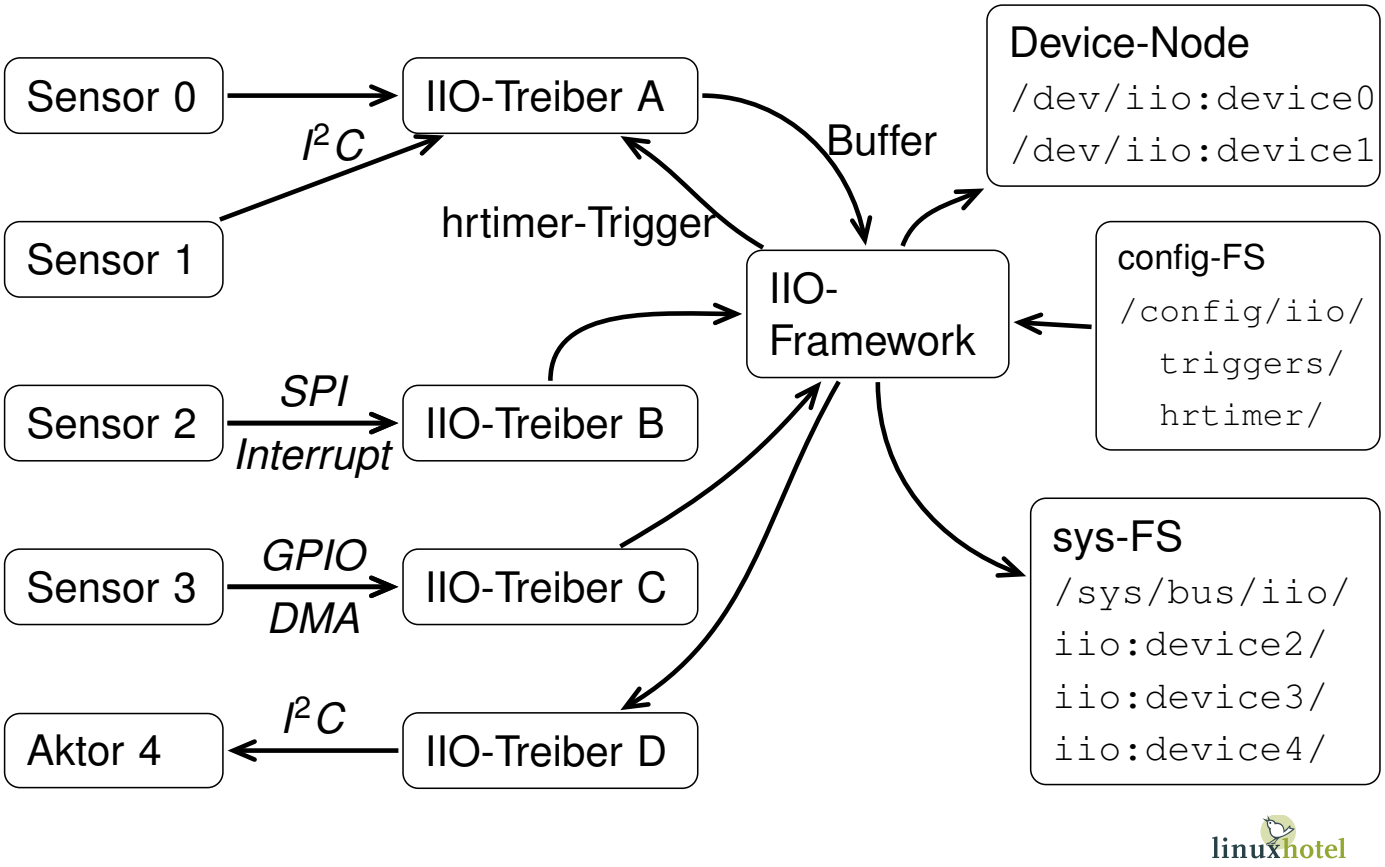
→ starke Bindung der Userspace-Anwendung an den Treiber

⇒ Austausch von Sensoren schwierig

⇒ Bibliotheken für alle Sensoren (fast) unmöglich

linuxhotel

Industrial-IO-Subsystem



- Sensoren liefern Daten an IIO-Treiber und Aktoren (z. B. Digital-Analog-Wandler) bekommen Daten vom Treiber.
- Anbindung an Kernel-Treiber kann mittels I^2C, SPI, GPIO, DMA, Interrupt, ... erfolgen
- IIO-Treiber meldet sich beim IIO-Framework an; er gibt an, welche Daten er zur Verfügung stellt, welche Kanäle er hat, welche Attribute verfügbar sind, ob und wie gepufferter Zugriff unterstützt wird, welche Events (z. B. Threshold) er selber liefert
- IIO-Framework liefert vereinheitlichtes Interface im sysfs für den synchronen Zugriff auf Meßdaten, Einstellung von Attributen, Abfrage und Einstellung von Triggern, Abfrage des Datenformates bei gepufferter Abfrage
- config-FS wird genutzt, um hrtimer-Trigger für einen Treiber anzulegen
- Zugriff auf Buffer mit Meßwerten erfolgt mittels Character Device Node (`/dev/iio:deviceN`)

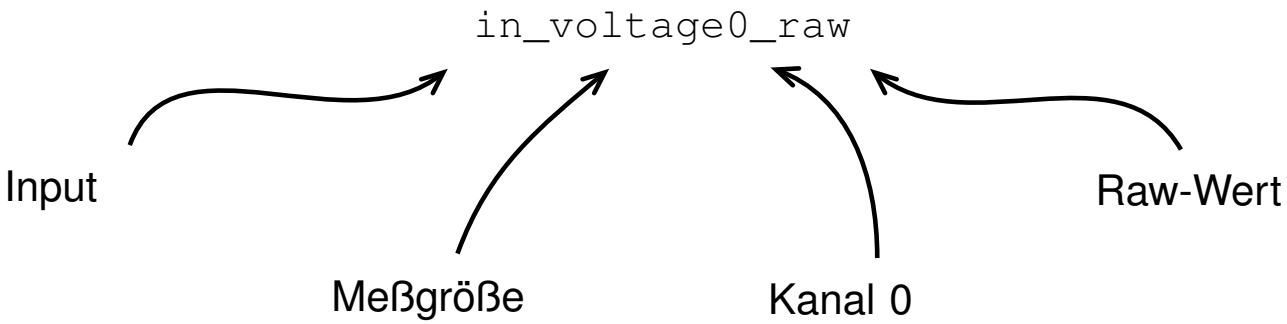
Industrial-IO-Subsystem (IIO) I

- Unterstützung für den Zugriff auf Sensoren
- standardisiertes Interface für Kernel-Treiber und Userspace-Zugriff
  $\Rightarrow$ Userspace-Applikation für unterschiedliche Sensoren verwendbar
- Hardware-Schnittstellen:
  I^2C
  SPI
  GPIO
  usw.
- Abfrage und Parametrisierung im sysfs:

linuxhotel

Industrial-IO-Subsystem (IIO) II

```
/sys/bus/iio/iio:device2/
```



- für jede Meßgröße ist (SI-) Einheit festgelegt und für alle Sensoren gleich
- Attribute (Verstärkung, Skalierung, Offset, ...) standardisiert im sysfs abgebildet

linuxhotel

Industrial-IO-Subsystem (IIO) III

- für hohe Abfragefrequenz geeignet
  z. B.: Zero-Copy bis zum Userspace:
  - Datenerfassung mittels DMA und Buffer-Swap
  - Einblenden in den Userspace mit `mmap()` und Memory-Swap

linuxhotel

`lsiio` — Liste registrierte IIO-Devices I

Tools im Verzeichnis `tools/iio` des Kernels

```
root@waage: lsiio -v

Device 000: bme280
   in_temp_input
   in_humidityrelative_input
   in_pressure_input

Device 001: hx711
   in_voltage0_raw
   in_voltage1_raw
```

linuxhotel

IIO-Device — Verzeichnisstruktur I

```
root@waage: tree /sys/bus/iio/devices/iio:device1
.
|-- buffer
|   |-- enable
|   |-- length
|   `-- watermark
|-- current_timestamp_clock
|-- dev
|-- in_voltage0_raw
|-- in_voltage0_scale_available
|-- in_voltage1_raw
|-- in_voltage1_scale_available
|-- in_voltage_scale
|-- name
|-- of_node -> [...] /firmware/devicetree/base/ocp/weight0
|-- power
|   |-- autosuspend_delay_ms
```

linuxhotel

IIO-Device — Verzeichnisstruktur II

```
|   |-- control
|   |-- runtime_active_time
|   |-- runtime_status
|   `-- runtime_suspended_time
|-- scan_elements
|   |-- in_timestamp_en
|   |-- in_timestamp_index
|   |-- in_timestamp_type
|   |-- in_voltage0_en
|   |-- in_voltage0_index
|   |-- in_voltage0_type
|   |-- in_voltage1_en
|   |-- in_voltage1_index
|   `-- in_voltage1_type
|-- subsystem -> ../../../../../bus/iio
|-- trigger
|   `-- current_trigger
`-- uevent
```

linuxhotel

IIO-Features I

Kanal-Beschreibung

- Struktur `struct iio_chan_spec` definiert vorhandene Kanäle und deren Eigenschaften
- `.type` legt fest, welche Größe eingelesen werden soll
  Beispiele:
  `IIO_VOLTAGE`, `IIO_ACCEL`, `IIO_DISTANCE`, `IIO_TEMP`, ...
- `.info_mask_separate` legt die Eigenschaften fest, welche es für diesen Kanal und diese Meßgröße gibt.

linuxhotel

IIO-Features II

Buffer-Modus

- im Direkt-Modus werden die Daten vom Gerät geholt, wenn im sysfs abgefragt wird (`INDIO_DIRECT_MODE`)
- Daten können auch getriggert eingelesen und gepuffert werden (`INDIO_BUFFER_TRIGGERED`)
- hrtimer-Events als Trigger
  → einstellbar mittels config-FS
- asynchrone Abfrage der Daten aus dem Userspace mit Device-Node `/dev/iio:deviceN`
- als Beispiel:
  `tools/iio/iio_generic_buffer.c`

linuxhotel

iio_generic_buffer — getriggerte Daten

```
root@waage: cd /sys/kernel/config/iio/triggers/hrtimer
root@waage: mkdir mytmr

root@waage: cd /sys/bus/iio/devices/trigger0
root@waage: echo 1 > sampling_frequency

root@waage: cd /sys/bus/iio/devices/iio\:device1/scan_elements
root@waage: echo 1 > in_voltage0_en
root@waage: echo 1 > in_timestamp_en

root@waage: iio_generic_buffer -c3 -n hx711 -t mytmr

iio device number being used is 1
iio trigger number being used is 0
/sys/bus/iio/devices/iio:device1 mytmr
10.062591 15118212303273263 20
11.024664 15118212313273243 60
10.025594 15118212323273238 00
```

linuxhotel

Welche Sensoren sind mainline? I

Sensoren (Stable v4.13, Treibervarianten und Staging nicht mitgezählt):

Sensortyp	Verzeichnis	Anzahl
Beschleunigung	accel	24
AD-Wandler	adc	74
Verstärker	amplifier	1
chem. Sensoren	chemical	3
Zähler	counter	1
DA-Wandler	dac	30
Frequency Synthesizer (PLL)	frequency	2
Beschleunigung	gyro	15
Herzfrequenz, Puls	health	4
Feuchtigkeit	humidity	8

linuxhotel

Welche Sensoren sind mainline? II

Trägheit	`imu`	6
Beleuchtung	`light`	37
Magnetfeld	`magnetometer`	9
Neigung	`orientation`	2
Potentiometer	`potentiometer`	6
Potentiostat	`potentiostat`	1
Luftdruck	`pressure`	13
Abstand	`proximity`	5
Temperatur	`temperature`	7
Summe		248

IIO-Device — Definition des Sensors

```
static const struct iio_info hx711_iio_info = {
    .driver_module     = THIS_MODULE,
    .read_raw          = hx711_read_raw,
    .write_raw         = hx711_write_raw,
    .write_raw_get_fmt = hx711_write_raw_get_fmt,
    .attrs             = &hx711_attribute_group,
};

static const struct iio_chan_spec hx711_chan_spec[] = {
    {
        .type                      = IIO_VOLTAGE,
        .channel                   = 0,
        .indexed                   = 1,
        .info_mask_separate        = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared_by_type  = BIT(IIO_CHAN_INFO_SCALE),
    },
    [...]
};
```

linuxhotel

1 `struct iio_info` definiert die Treiberzugriffsfunktionen, hier `read_raw(), write_raw(), ...`
→ werden für alle Kanäle genutzt

9 mit dem Strukturarray von `iio_chan_spec` wird definiert, welche Kanäle mit welchen Meßgrößen verfügbar sind und welche Daten dazu jeweils existieren.

11 `IIO_VOLTAGE`: AD-Wandler liefert Spannung

14 `IIO_CHAN_INFO_RAW`: gemessener Raw-Wert des AD-Wandlers für jeden Kanal

15 `IIO_CHAN_INFO_SCALE`: Skalierung des gelieferten Raw-Wertes auf die meßgrößenspezifische Standardeinheit;
z. B.: `IIO_VOLTAGE` wird auf mV skaliert

IIO-Device — Registrierung in `probe()` I

```
static int hx711_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct hx711_data *hx711_data;
5   struct iio_dev *indio_dev;
    int ret;

    indio_dev = devm_iio_device_alloc(dev,
                                      sizeof(struct hx711_data));
10  if (!indio_dev) {
        dev_err(dev, "failed to allocate IIO device\n");
        return -ENOMEM;
    }

15  hx711_data = iio_priv(indio_dev);
    hx711_data->dev = dev;
```

linuxhotel

8 `devm_iio_device_alloc()` alloziert eine `struct iio_dev` sowie Platz für eine private Datenstruktur des Treibers, hier `struct hx711_data`

15 mit `iio_priv()` wird der Zeiger auf die private Datenstruktur (unmittelbar nach `struct iio_dev` plus Alignment) geliefert

IIO-Device — Registrierung in `probe()` II

```
    [...]
20  platform_set_drvdata(pdev, indio_dev);

    indio_dev->name = "hx711";
    indio_dev->dev.parent = &pdev->dev;
    indio_dev->info = &hx711_iio_info;
25  indio_dev->modes = INDIO_DIRECT_MODE;
    indio_dev->channels = hx711_chan_spec;
    indio_dev->num_channels = ARRAY_SIZE(hx711_chan_spec);

    ret = iio_device_register(indio_dev);
30  if (ret < 0) {
        dev_err(dev, "Couldn't register the device\n");
        regulator_disable(hx711_data->reg_avdd);
    }

35  return ret;
}
```

linuxhotel

21 mit `platform_set_drvdata()` wird private Datenstruktur in Platform-Treiber-Struktur eingehängt
⇒ erreichbar in allen Funktionen mit `struct platform_device`

26 der Modus `INDIO_DIRECT_MODE` gibt an, daß der Treiber direkt abgefragt wird (SW-Trigger); bei der Abfrage mittels sysfs wird synchron auch die Datenabfrage am Gerät durchgeführt.

30 `iio_device_register()` registriert das neu definierte IIO-Device am IIO-Subsystem

IIO-Device — Zugriffsfunktion `read_raw()`

```
static int hx711_read_raw(struct iio_dev *indio_dev,
                const struct iio_chan_spec *chan,
                int *val, int *val2, long mask)
{
    struct hx711_data *hx711_data = iio_priv(indio_dev);
    int ret;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        [...]
        *val = hx711_read(hx711_data);
        return IIO_VAL_INT;

    case IIO_CHAN_INFO_SCALE:
        *val = 0;
        *val2 = hx711_get_gain_to_scale([...]);
        return IIO_VAL_INT_PLUS_NANO;
[...]
```

linuxhotel

- in der `read_raw()`-Funktion landen alle Lesezugriffe des Treibers
  ⇒ Unterscheidung erforderlich nach:
  Meßgröße (`IIO_VOLTAGE`, hier weggelassen),
  Kanal (hier weggelassen)
  und Attribut (`IIO_CHAN_INFO_RAW`, `IIO_CHAN_INFO_SCALE`)

- Rückgabewert gibt an, was in den beiden Rückgabewerten `val` und `val2` drinnen steht; hier:

13 `IIO_VAL_INT`:
in `val` steht Integerwert

17 `IIO_VAL_INT_PLUS_NANO`:
in `val` steht Integerwert als Vorkommastelle und in `val2` steht Integerwert als Nachkommastelle mit Wertigkeit 10^{-9}

```
unsigned int nrdev, nratt, nrchan, i, j;
struct iio_context *ctx;
struct iio_device *dev;
struct iio_channel *chan;

ctx = iio_create_default_context();

nrdev = iio_context_get_devices_count(ctx);

for (i = 0; i < nrdev; i++) {

  dev = iio_context_get_device(ctx, i);

  nratt = iio_device_get_attrs_count(dev);

  for (j = 0; j < nratt; j++) {
    /* read device specific attributes */
  }
```

linuxhotel

6 holen eines einfachen Kontextes;
alternativ könnte auch eine Verbindung zu einem `iiod`-Dämon hergestellt werden

8 Anzahl an Devices abfragen (`iio:device<N>`)

12 Device-Objekt holen

14 Anzahl an Attributen, welche zum Device gehören abfragen

16-18 Attribute abfragen; hier gekürzt

```
    nrchan = iio_device_get_channels_count(dev);
20
    for (j = 0; j < nrchan; j++) {
      chan = iio_device_get_channel(dev, j);
25    nratt = iio_channel_get_attrs_count(chan);
      /* get channel specific attributes */
    }
  }
```


19 Abfrage, wie viele Kanäle vom Device bereitgestellt werden

23 Kanal-Objekt holen

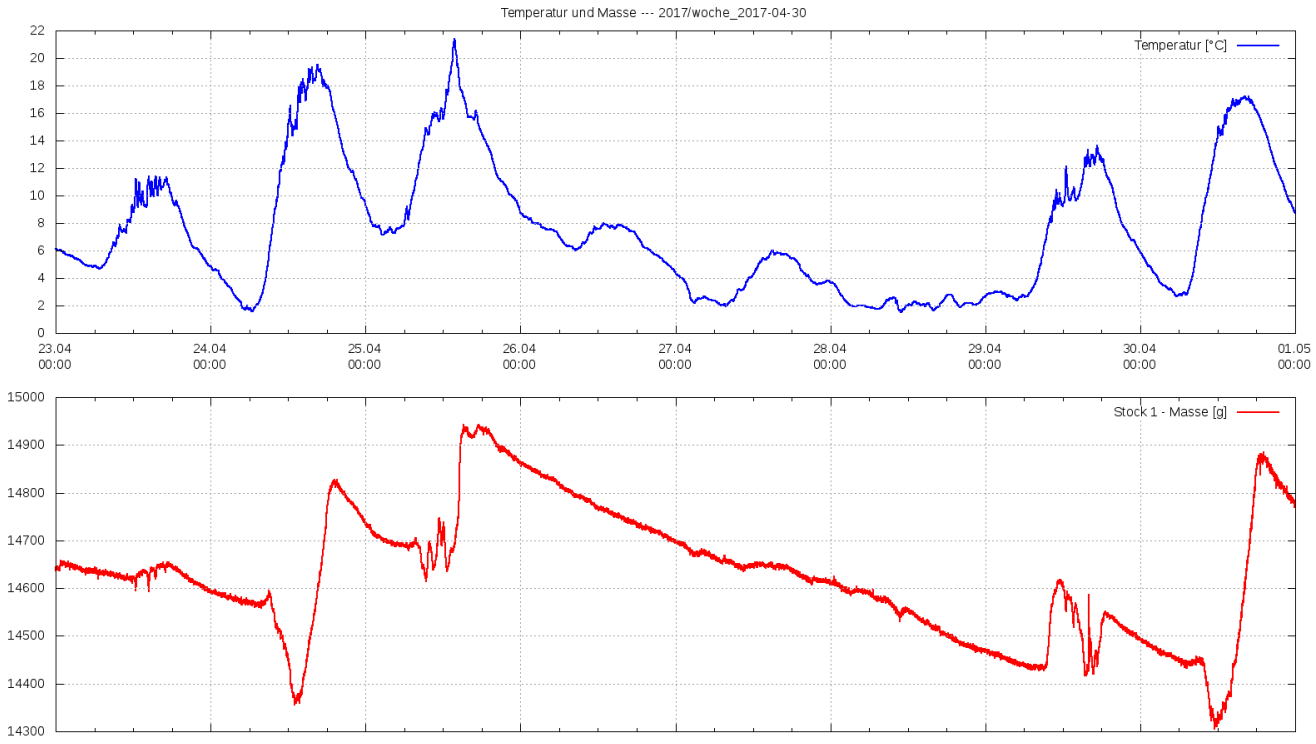
25 Abfrage der Anzahl an Attributen, welche zum Kanal gehören

27 Abfrage der kanalspezifischen Attribute; hier gekürzt

Community

- viel Aktivität auf Mailing-Liste linux-iio
- ursprünglicher Autor und Maintainer ist Jonathan Cameron
- gründlicher und zügiger Review
  → gute Ideen wie man es anders machen könnte
- starker Fokus auf Vereinheitlichung:
  In welche vorhandene Gruppe passt ein Treiber hinein?
  Verwendung etablierter Schnittstellen anstatt neue zu kreieren
- Danke an Lars-Peter Clausen für den Review der Folien

Auswertung der Daten mit GNU-Plot I



Datum Vorkommnisse

24.04. Temperatur > 12^{o}C
Bienen fliegen aus und Tracht vorhanden
$\Rightarrow$ Honig (Nektar) wird eingetragen

25.04. dito.

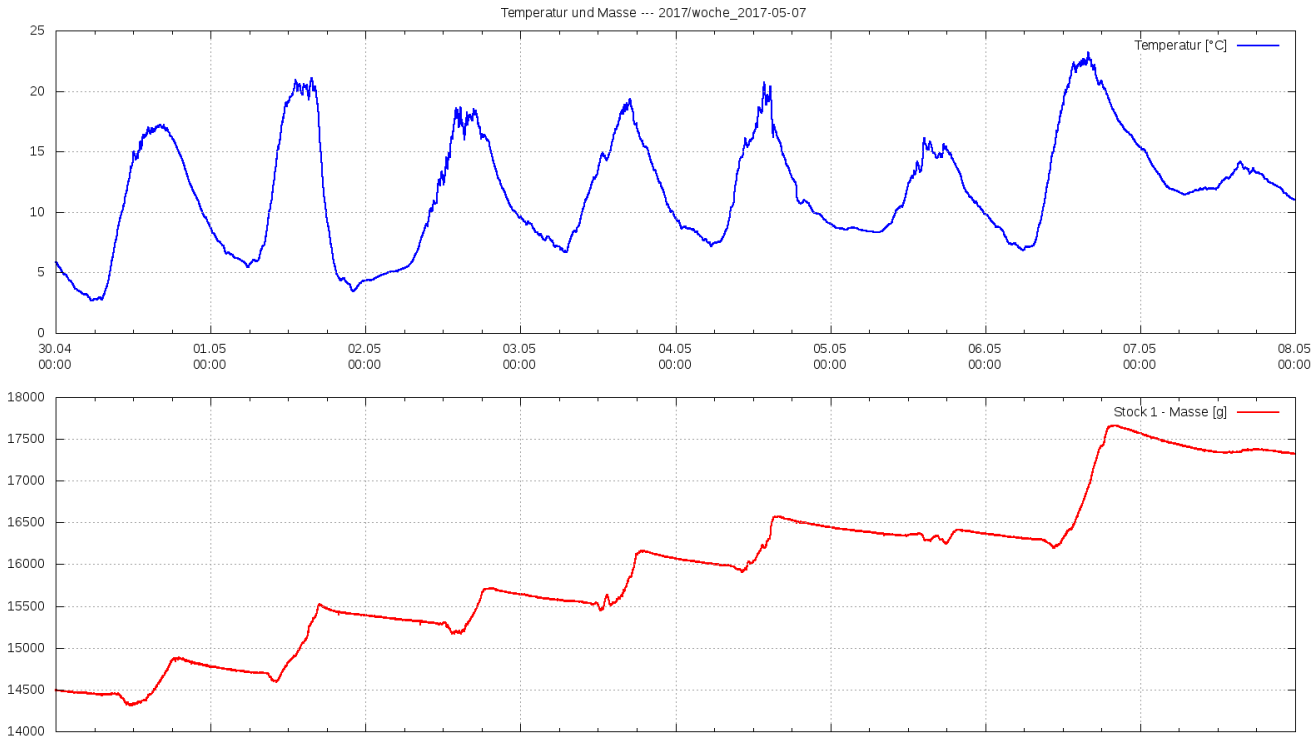
30.04. dito.

26.04. Temperatur < 12^{o}C
Honig wird verbraucht

27.04. dito.

28.04. dito.

Auswertung der Daten mit GNU-Plot II



30.04. Temperatur > 12^{o}C
Bienen fliegen aus und Tracht vorhanden
$\Rightarrow$ Honig (Nektar) wird eingetragen

01.05. dito.

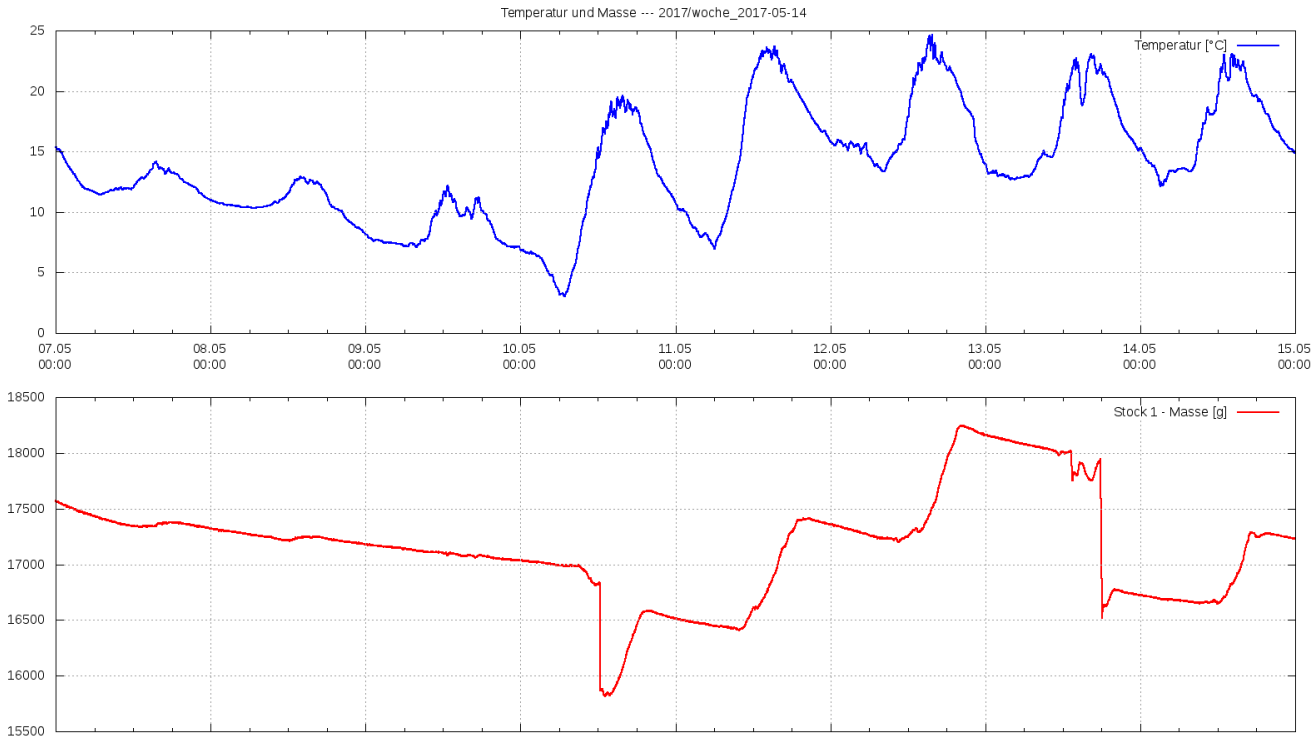
02.05. dito.

03.05. dito.

04.05. dito.

06.05. dito. (1300 g Nektar)

Auswertung der Daten mit GNU-Plot III



07.05. Temperatur < 12^{o}C
Honig wird verbraucht

08.05. dito.

09.05. dito.

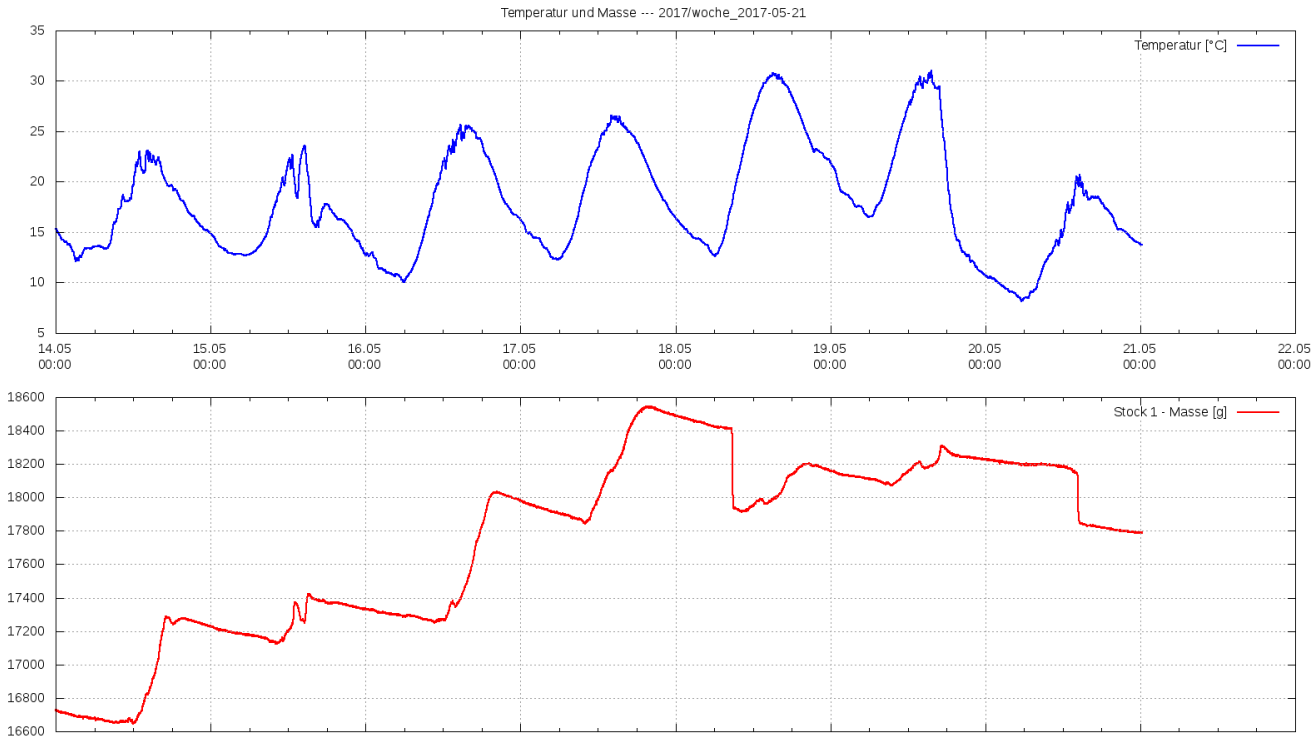
10.05. Bienenschwarm zieht um die Mittagszeit (12:00 Uhr) aus
→ natürliche Volksteilung (ca. 1000 g stark)

11.05. Temperatur > 12^{o}C
Bienen fliegen aus und Tracht vorhanden
⇒ Honig (Nektar) wird eingetragen

12.05. dito. (900 g Nektar)

13.05. Bienenvolk ist durch Schwarm kleiner geworden
→ nicht benötigte Rähmchen entfernt in manuellem Eingriff

Auswertung der Daten mit GNU-Plot IV



14.05. Temperatur > 12^oC
Bienen fliegen aus und Tracht vorhanden
$\Rightarrow$ Honig (Nektar) wird eingetragen

15.05. dito.

16.05. dito.

17.05. dito.

18.05. weiterer Bienenschwarm (Nachschwarm) zieht aus (ca. 500 g)

20.05. nochmals zieht Bienenschwarm (Nachschwarm) aus (ca. 350 g)

linuxhotel

Pseudo-Filesysteme

- procfs und sysfs für Systeminformationen aus dem Kernel
- debugfs für Debugging- und Tracing-Ausgaben aus dem Kernel
- ramfs dient als Filesystem im RAM mit fester Größe und Anzahl (Booten)
- tmpfs ist analog zu RAM-FS; jedoch zusätzlich variable Dateisystemgröße und Anzahl an Dateisystemen

linuxhotel

sys-Filesystem I

- strukturierte Darstellung des Device-Modells ab Kernel 2.6
- Einteilung nach Subsystemen, Geräteklassen und Treibern
  → spiegelt den internen Aufbau des Kernel wieder
- soll /proc-Filesystem in der Zukunft ersetzen

linuxhotel

sys-Filesystem II

Einteilung

- *block:* Block-Devices (loop, ram, sda, hda, ...)
- *bus:* Bus-Systeme (acpi, i2c, pci, usb, ...)
- *class:* Klassen von Gerätetreibern (input, net, tty, ...)
- *dev:* Gerätetreiber (Block- und Character-Devices) sortiert nach Major- und Minor-Number
- *devices:* Low-Level-Darstellung der Devices des Linux-Device-Modells
- *kernel:* Kernel-Informationen
- *module:* Kernel-Module

linuxhotel

sys-Filesystem III

Verwendung

- *Power-Management:* Geräte je nach Power-Zustand hierarchisch durchlaufen und benachrichtigen (z. B. USB-Gerät vor dem USB-Hub abschalten)
- *Hotplugging:* Hinzufügen oder Entfernen von Geräten generiert Events, die vom Hotplug-Dämon verarbeitet werden
- *Device-Nodes:* Events geben geladene Devices bekannt und `udevd`-Dämon kann entsprechende Device-Nodes anlegen
- *Userspace:* lesen und ändern von Treiber-Parametern aus dem Userspace

linuxhotel

sys-Filesystem IV

- jedes Verzeichnis hat im Kernel als Grundlage ein Objekt vom Typ `struct kobject`
- Dateien entsprechen den Attributen des Subsystems, der Klasse oder des Gerätetreibers
- Attribute können maximal mit Character-Daten von einer Page-Size (typ. 4 kiB) zurückgegeben oder beschrieben werden
- Attribute sollen einem Parameter mit Grunddatentyp (int, char*, ...) entsprechen
- sysfs ersetzt sowohl das procfs sowie die ioctl's von Gerätetreibern
- neue Gerätetreiber-Klasse kann mit `class_create()` angelegt werden

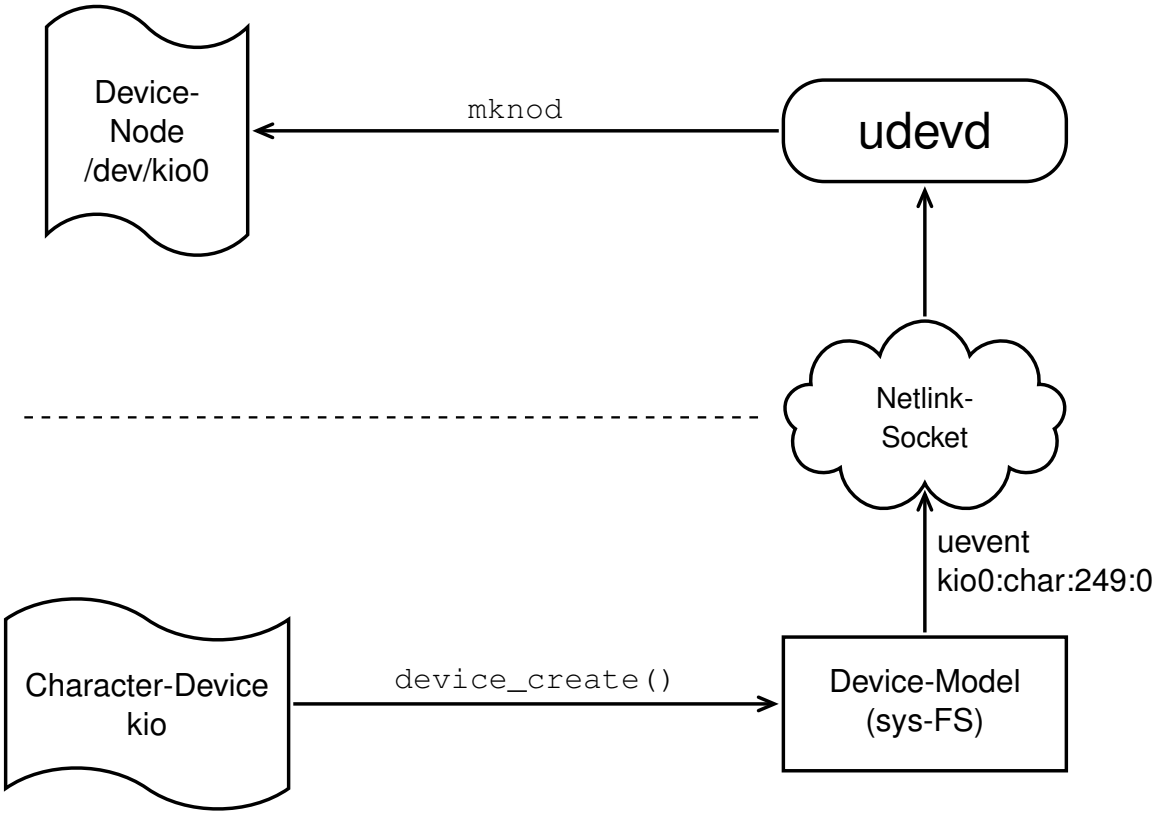
linuxhotel

sys-Filesystem V

- in der neuen Klasse kann ein Character-Device mit `device_create()` angelegt werden; die Major- und Minor-Numbers werden automatisch mittels Default-Attributen veröffentlicht und können vom `udevd`-Dämon verwendet werden
- Attribute (Parameter) des Treibers werden mittels `DEVICE_ATTR` deklariert und eine entsprechende Datei mittels `device_create_file()` im sysfs erzeugt
- *Beispiel tracing:* Verwendung von dynamischen Major-Nummern mittels uevent für `udevd`-Dämon; Treiber-Parameter anzeigen und ändern
- Beobachten von Events des `udevd`-Dämon: `udevadm monitor`
- Eintragen des Gerätes in `/etc/udev/rules.d/92-kio.rules`

linuxhotel

udevd - dynamische Device-Nodes



linuxhotel

Beispiel: uevents mittels sysfs unterstützen I

```
static struct class* kio_class;
static struct device* kio_device;

static int __init kio_init(void)
{
  ...

  // Eintrag fuer uevent in /sys
  kio_class = class_create (THIS_MODULE, "kio");
  if (IS_ERR(kio_class))
  {
    printk(KERN_ERR "class_create() -err:%ld\n",
                    PTR_ERR(kio_class));
    return ((-1) * PTR_ERR(kio_class));
  }
```

linuxhotel

Beispiel: uevents mittels sysfs unterstützen II

```
  // kio_dev von alloc_chrdev_region(), ...
  kio_device = device_create (kio_class, NULL,
      kio_dev, NULL, "kio%d", MINOR(kio_dev));
}


static void __exit kio_exit(void)
{
  device_destroy (kio_class, kio_dev);
  class_destroy (kio_class);
}

// soll bestehende Klasse verwendet werden,
//   wird die dazugehörige globale Variable verwendet
//   z. B.: "input_class"  für  "/sys/class/input"
```

linuxhotel

Beispiel: Erweiterung um Attribute im sysfs I

```
static int led_value = 0;
ssize_t led_store (struct device* dev,
                   struct device_attribute* attr,
                   const char* buf, size_t count)
{
  unsigned int val;
  if ((ret = kstrtouint(buf, 10, &val)))
    return ret;
  led_value = val;
  return count;
}
ssize_t led_show (struct device* dev,
                  struct device_attribute* attr, char* buf)
{
  sprintf (buf, "%d\n", led_value);
  return strlen(buf);
}
```

linuxhotel

Beispiel: Erweiterung um Attribute im sysfs II

```
// Device-Attribute definieren
//   Makro DEVICE_ATTR generiert "dev_attr_led"
DEVICE_ATTR (led, 0664, led_show, led_store);

// init_module() u. cleanup_module() erweitern
//   --> Anlegen der Attribut-Datei
//       "/sys/class/kio/kio0/led"

static int __init kio_init(void)
{
  ...

  ret = device_create_file (kio_device, &dev_attr_led);
  if (ret)
    printk(KERN_ERR "device_create_file(): ret=%d\n", ret);
  ...
}
```

linuxhotel

Beispiel: Erweiterung um Attribute im sysfs III

```
static void __exit kio_exit(void)
{
  ...

  device_remove_file (kio_device, &dev_attr_led);

  ...
}
```

linuxhotel

Dateien synchronisieren I

- Dateisystem synchron mounten (Zugriff langsamer)
  Option beim Mounten:
  `sync`
- einzelne Datei synchron machen
  → sync- / dirsync-Flag wird bei neu angelegten Dateien / Verzeichnissen geerbt
  In der Shell:
  `chattr +S`
- Datei synchron öffnen
  `open(... , O_SYNC);`

linuxhotel

Dateien synchronisieren II

- Synchronisierungspunkt bei Linux-Filedeskriptor
  `fsync();`
  `fdatasync();`

- Synchronisierungspunkt bei Streams (`FILE*`)
  `fflush();`

- Synchronisierung aller Dateisysteme
  In der Shell:
  `sync`
  → Dateisysteme werden aufgefordert zu synchronisieren
  → Wann ist Synchronisierung abgeschlossen?

linuxhotel

Teil II

Kernel-Architektur

Kernel-Architektur

Ausprägungen einer Task

Userspace Prozeß

- laufende Instanz eines Programmes
- isoliert von anderen Prozessen im Adreßbereich

Userspace Thread (Posix-Thread)

- separater Ausführungspfad innerhalb eines Prozesses
- eigener Stack aber gemeinsamer Adreßbereich

Kernel-Thread

- eigener Ausführungspfad innerhalb des Kernels
- Kernel-Adreßbereich
- kein virtueller Adressbereich

Scheduling-Klassen

Deadline-Task (ab 3.14)

haben Ablaufzeit und Zeitscheibe pro Zeiteinheit

Realtime-Task (RT-Task)

prioritätsbasiert unterbrechend

normale Task

im Round-Robin-Verfahren gescheduled
$\rightarrow$ überwiegende Mehrheit

Idle-Task

wenn keine anderen Tasks Rechenzeit benötigen; System wäre idle

Task-Status

Running

Task rechnet auf der CPU

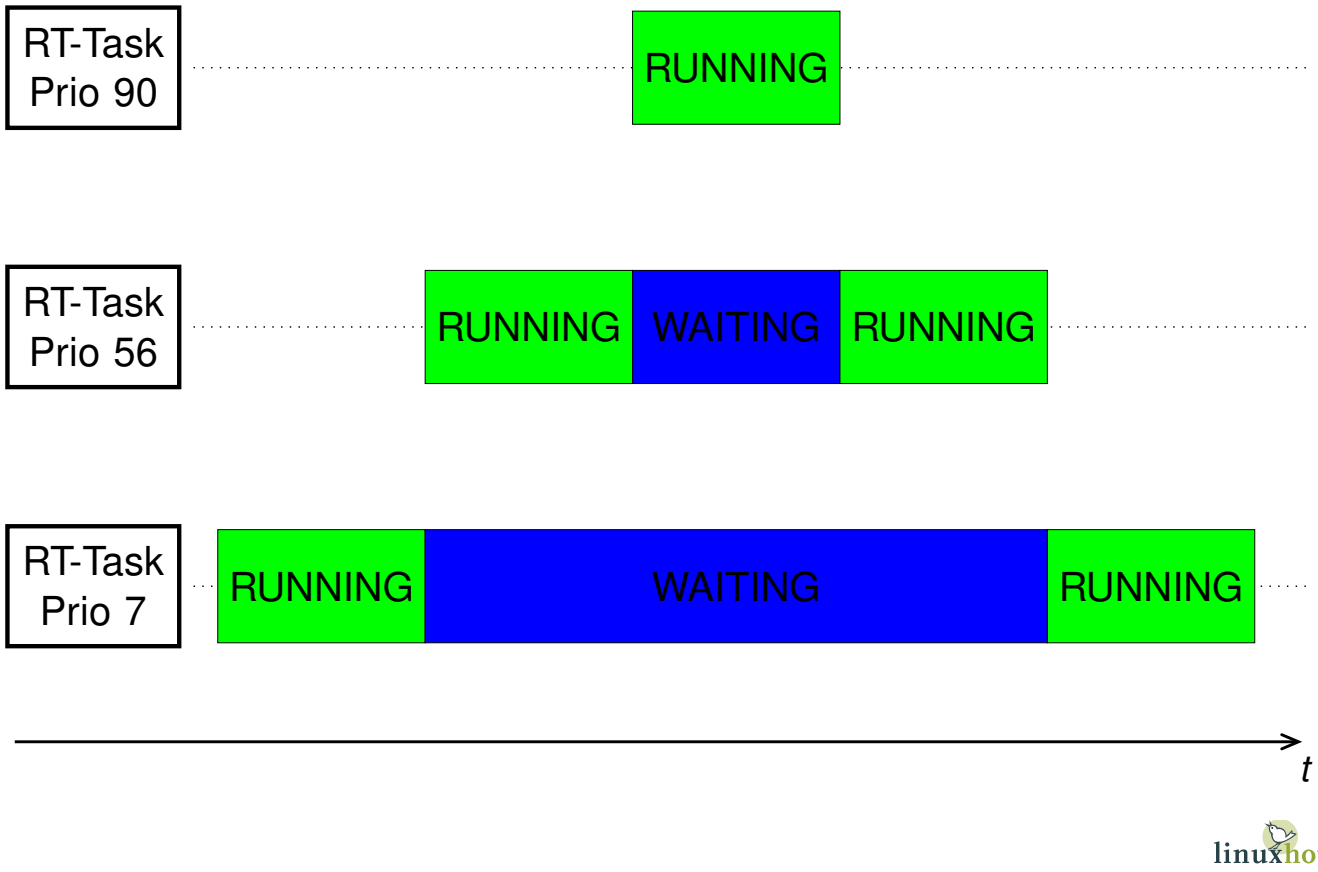
Waiting

Task ist rechenbereit, wartet aber auf Rechenzeit auf der CPU

Sleeping

Task ist nicht rechenbereit (blockiert, schläft)

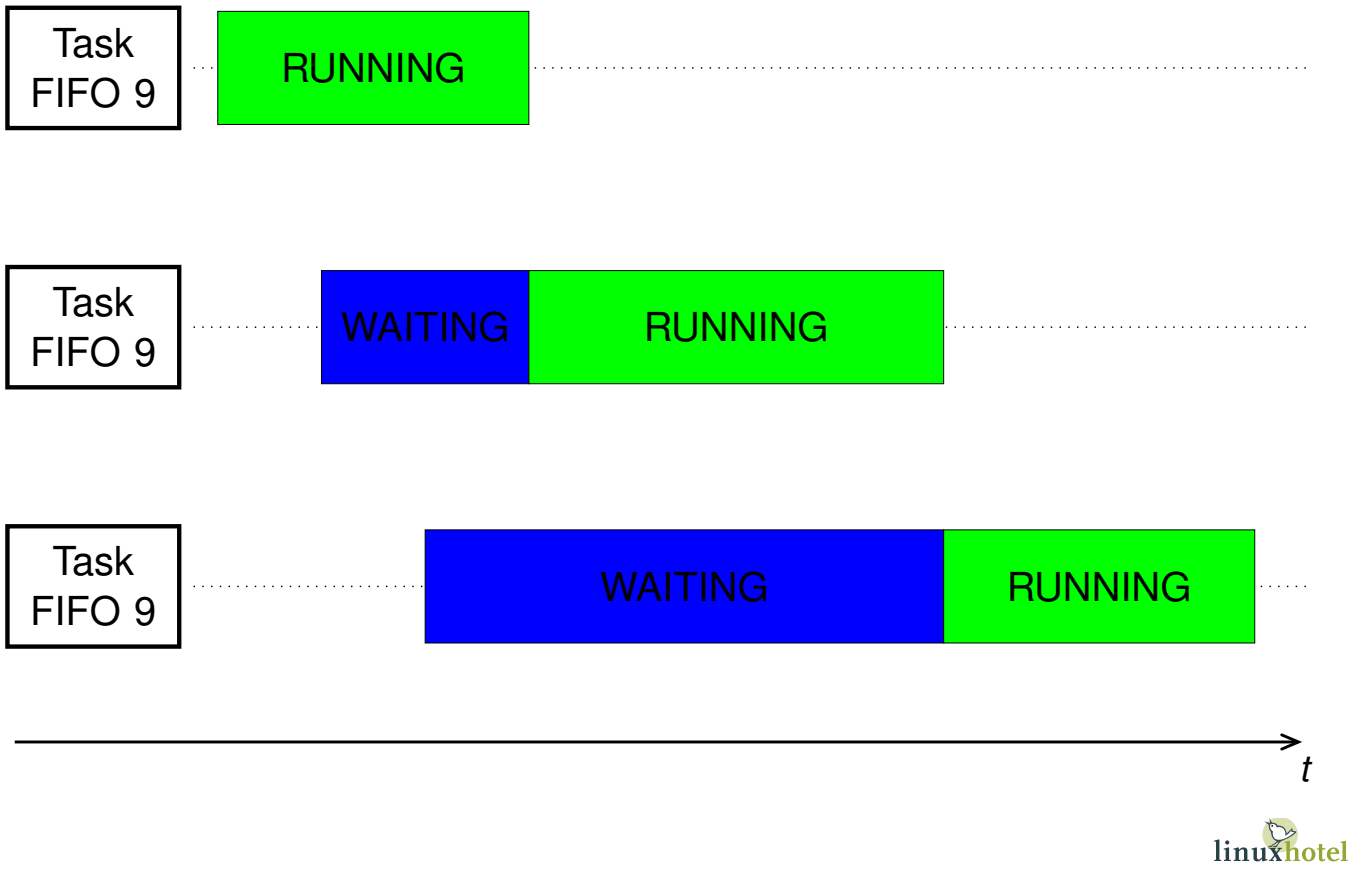
RT Task



linuxhotel

- sobald mindestens eine RT-Task rechenbereit ist, werden die RT-Tasks gescheduled
- Priorisierung bei mehreren RT-Tasks mittels einer Prioritätsstufe von 1 bis 99
  $\rightarrow$ höherer Zahlenwert = höhere Priorität
- höhere Priorität unterbricht niedrigere
  $\rightarrow$ Priority-Based-Preemptive-Scheduling
- innerhalb einer Prioritätsstufe zwei Verfahren möglich:
  - FIFO
    Tasks rechnen nacheinander und unterbrechen sich nicht
  - Round-Robin
    Tasks unterbrechen sich gegenseitig im Zeitscheibenverfahren

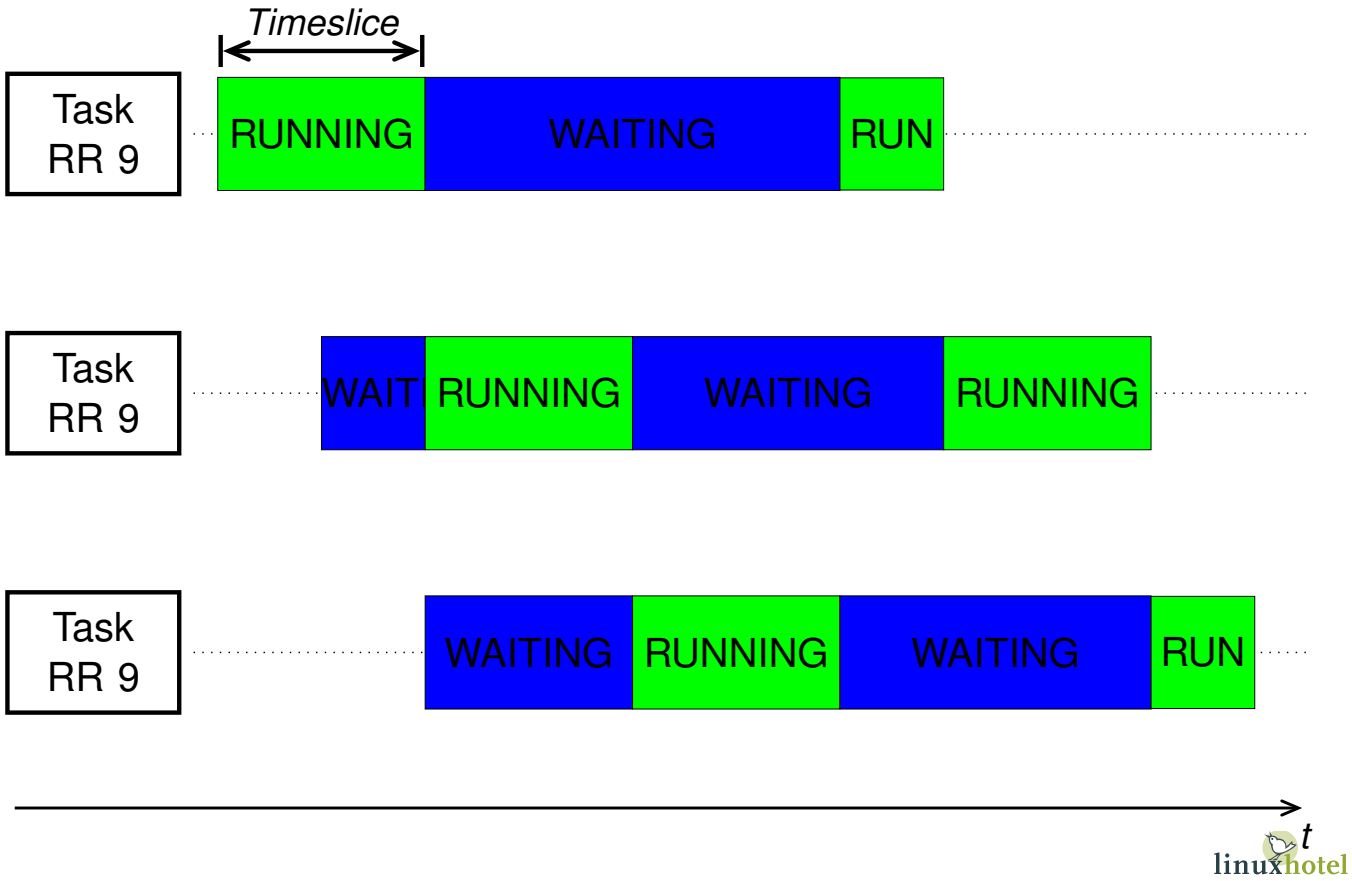
RT-Task-FIFO — gleiche Priorität



linuxhotel

- drei RT-Tasks mit der Scheduling-Policy `SCHED_FIFO` und gleicher Priorität werden nacheinander rechenbereit
- alle drei Tasks rechnen so lange, bis sie ihre Rechenzeit wieder abgeben
- sie unterbrechen sich nicht gegenseitig
- wird ein höher priorisierter RT-Task rechenbereit, unterbricht er den gerade laufenden

RT-Task-RR — gleiche Priorität



- drei RT-Tasks mit der Scheduling-Policy `SCHED_RR` und gleicher Priorität werden nacheinander rechenbereit
- jedem der Tasks wird eine Zeitscheibe zugewiesen, welcher er maximal rechnen darf
- nach Ablauf der Zeitscheibe kommt der nächste Task dran
- wird ein höher priorisierter RT-Task rechenbereit, unterbricht er den gerade laufenden
- Tasks wechseln sich gegenseitig ab
  → gegenseitige Unterbrechbarkeit bringt zusätzliche Komplexität im Design

Task einstellen I

Scheduling-Policy in der Shell einstellen

Task mit `SCHED_FIFO / 20` starten:

```
chrt -f 20 /bin/ls
```

Task mit *pid* = 367 auf `SCHED_FIFO / 80` einstellen:

```
chrt -f -p 80 367
```

Policy und Priorität abfragen:

```
chrt -p 367
```

linuxhotel

Task einstellen II

Was passiert bei nachfolgenden Aufrufen?

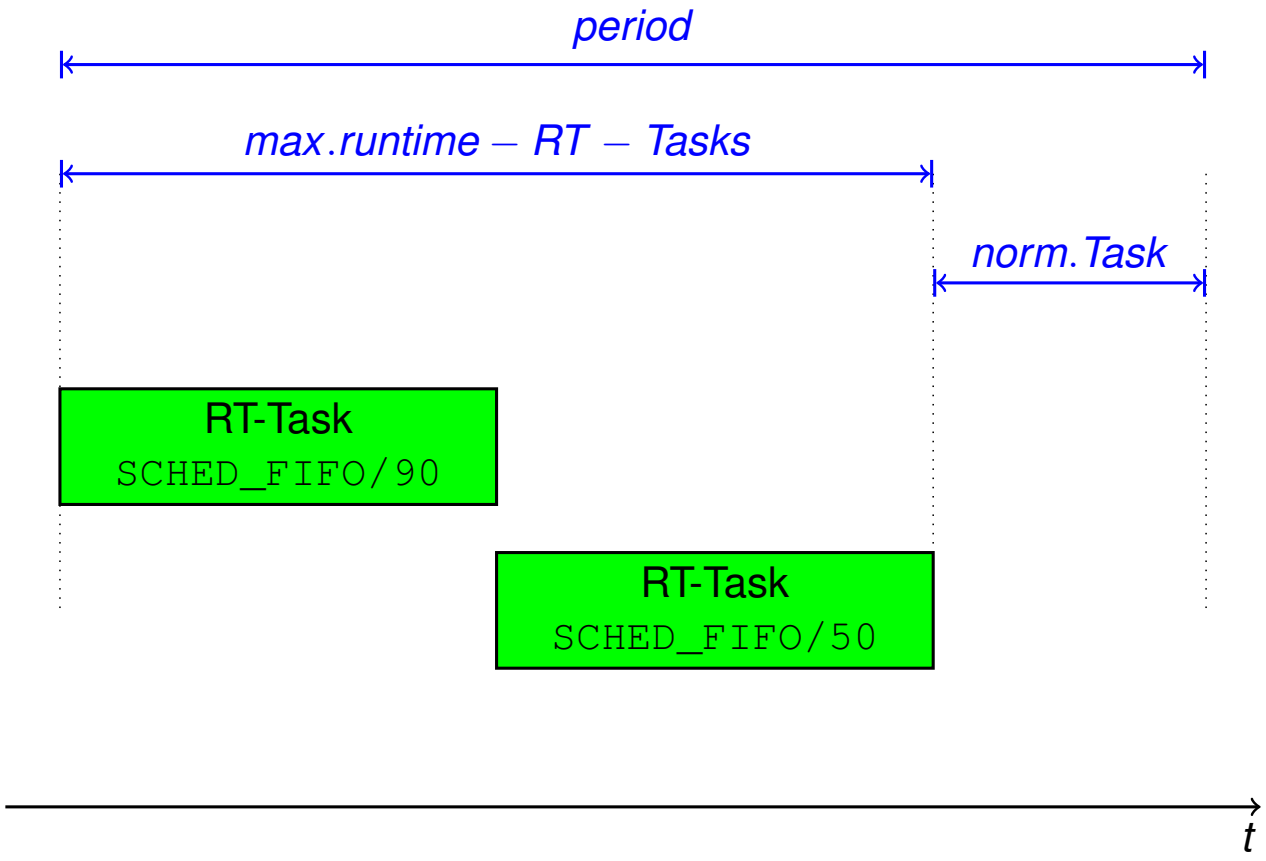
```
chrt -f 30 yes we can
```

```
chrt -f 30 dd if=/dev/zero of=/dev/null
```

```
chrt -f 70 gzip </dev/urandom >/dev/null
```

linuxhotel

RT-Throttling



- im betrachteten Zeitintervall (*period*) dürfen RT-Tasks eine maximale Zeit ununterbrochen rechnen;
  überschreiten diese zusammen die eingestellte Zeit werden Sie unterbrochen und Tasks aus dem normalen Round-Robin-Scheduling bekommen Rechenzeit
- Zeitperiode einstellbar in Datei
  `/proc/sys/kernel/sched_rt_period_us` in `[`μ`s]`
- maximale Laufzeit von RT-Tasks einstellbar in Datei
  `/proc/sys/kernel/sched_rt_runtime_us` in `[`μ`s]`
- Defaulteinstellung:
  950000μs Runtime
  1000000μs Period
  $\Rightarrow$ 95 % der CPU für RT-Tasks und Deadline-Tasks (ab 3.14) zusammen
- Abschalten von RT-Throttling:
  `echo -1 > /proc/sys/kernel/sched_rt_runtime_us`

Deadline-Task I

- Earliest-Deadline-First-Scheduling (EDF) erweitert um Constant-Bandwidth-Server (CBS)
- je näher die Deadline rückt, umso höher ist die Priorität (EDF)
- Begrenzung der Rechenzeit pro Zeitperiode (CBS)
- Zielsetzung:
  - direkte Übertragung zeitlicher Randbedinungen in Design
    $\rightarrow$ in der Regel werden zeitliche Bedinungen gestellt und keine Prioritätenlisten vorgegeben
  - bessere Nutzung der verfügbaren CPU-Rechenzeit durch Echtzeit-Tasks

linuxhotel

Deadline-Task II

- spezifiziert werden:

$t_{runtime}$ maximale Rechenzeit pro Zeiteinheit ($\geq$ *WCET*)

$t_{deadline}$ max. Zeit bis zur Fertigstellung des Tasks

t_{period} betrachtete Zeiteinheit; minimale Zeit innerhalb derer der Task erneut *runtime* Zeit bekommt ($\leq 1/f_{max}$)

- Zeiteinheit beginnt mit dem Zeitpunkt zu dem Task rechenbereit ist; also aufgeweckt wird
- wenn WCET nicht überschritten wird:
  $\rightarrow$ garantierte Zuteilung der Zeitscheibe bevorzugt vor RT-Tasks
- bei Überschreiten der spezifizierten runtime:
  $\rightarrow$ Task wird abgebrochen

linuxhotel

Deadline-Task III

- RT-Throttling wird von RT-Tasks und Deadline-Tasks gemeinsam genutzt
- anteilige Rechenzeit kann maximal der anteiligen Runtime aus dem Throttling entsprechen:
  $\Rightarrow$ Default max. 95 % Rechenzeit für Deadline-Tasks zusammen
- forken von Deadline-Tasks ist nicht zulässig
  $\Rightarrow$ würde anteilige Rechenzeit vervielfachen
- siehe auch:
  `linux/Documentation/scheduler/sched-deadline.txt`

linuxhotel

Deadline-Task IV

Scheduling-Policy in der Shell einstellen

`SCHED_DEADLINE`

$t_{runtime}$	50 μs
$t_{deadline}$	500 μs
t_{period}	1000 μs

```
chrt -d -T 50000 -D 500000 -P 1000000 0          \
        /bin/dd if=/dev/zero of=/dev/null
```

linuxhotel

Zeitermittlung im Userspace I

CPU-Rechenzeit

Messung von $t_{cpu,i}$:

```
struct timespec ts;
clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
```

daraus Maximalwert unter Worst-Case-Bedingungen ermitteln:

$$t_{runtime} = max(..., t_{cpu,i}, ...) \qquad i = 0 \ldots n$$

linuxhotel

Zeitermittlung im Userspace II

Gesamtlaufzeit (Rechenzeit plus Wartezeit)

Messung von $t_{ges,i}$:

```
clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
```

$\Rightarrow$ Korrekturwert für $t_{deadline}$ ergibt sich zu:

$$\Delta t_{deadline} = max(..., (t_{ges,i} - t_{cpu,i}), ...) \qquad i = 0 \ldots n$$

linuxhotel

Zeitermittlung im Userspace III

Achtung

$t_{runtime}$ wird über- oder
t_{period} unterschritten:

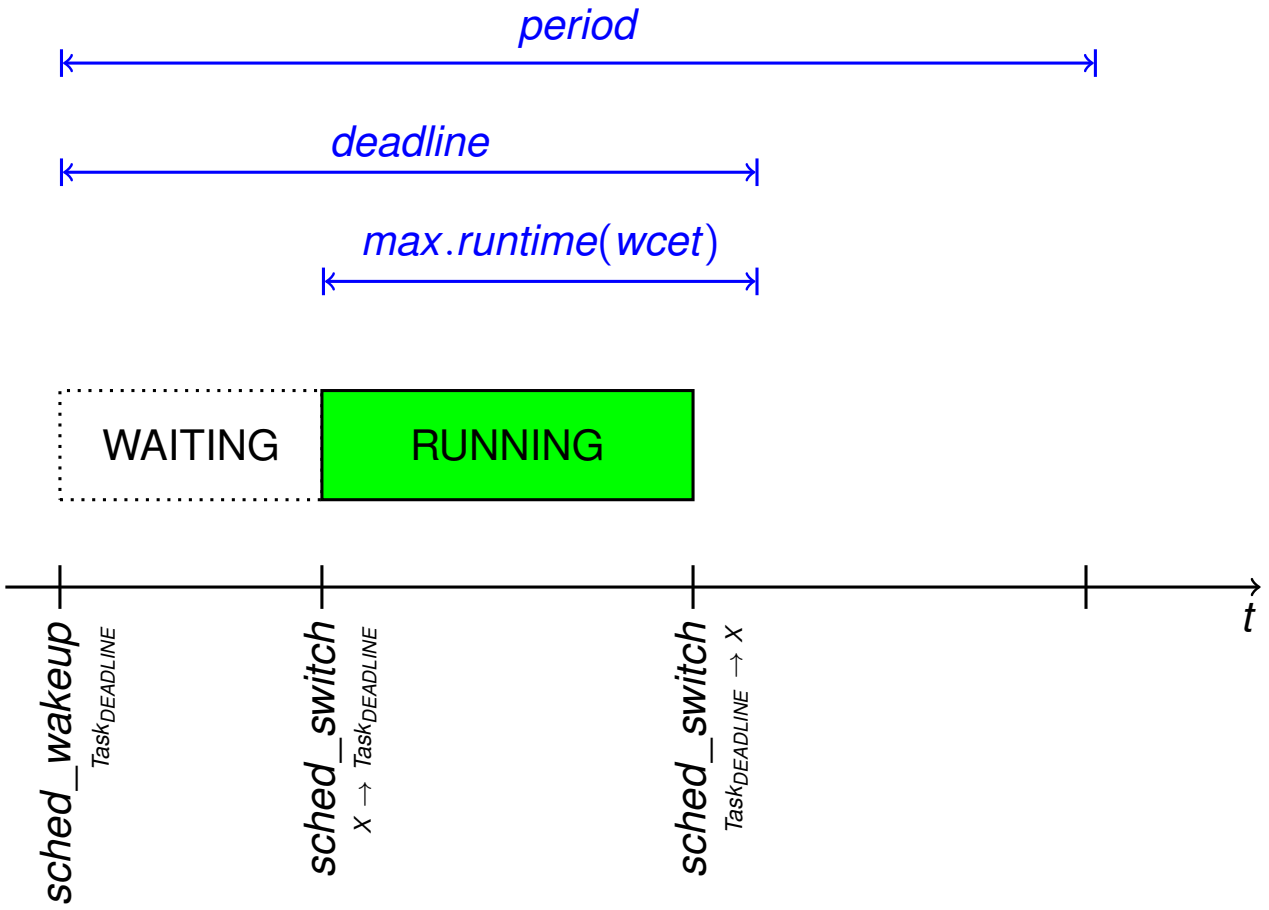
$\Rightarrow$ Task verliert Determinismus!

$\rightarrow$ Zeitparameter müssen für Worst-Case-Fall ermittelt werden

$\rightarrow$ System kann sogar idle werden, obwohl Deadline-Tasks da wären!

linuxhotel

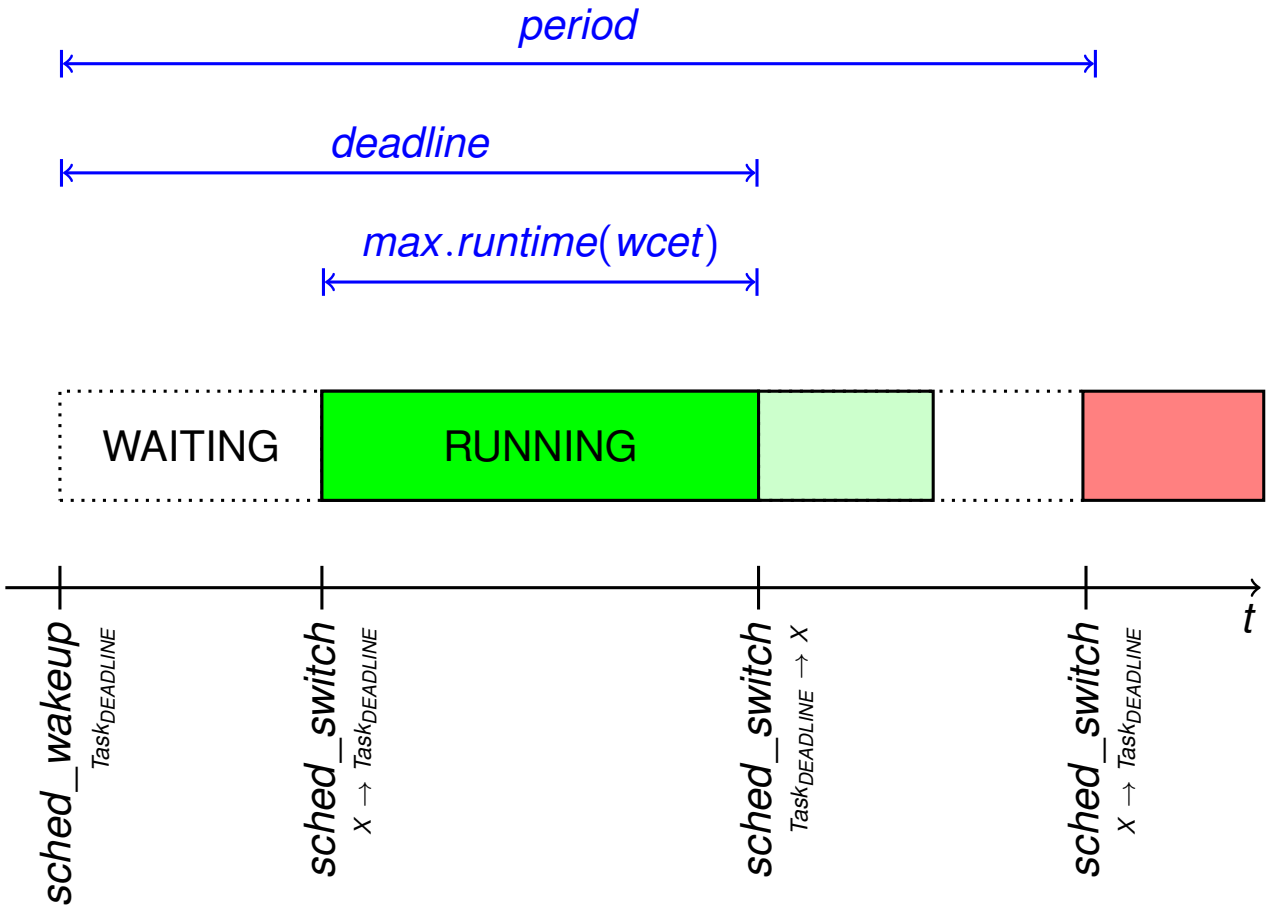
Deadline-Task



- Deadline-Task erhält vom Scheduler, gemessen ab dem Zeitpunkt des Wakeup's (sched_wakeup) innerhalb der Deadline-Zeitspanne maximal die spezifizierte Rechenzeit
- ein Wakeup kann z. B. sein:
  - es wird blockierend auf ein Timer-Ereignis gewartet und der entsprechende Timer ist abgelaufen
  - Task wurde beim Versuch einen Mutex zu locken blockiert; der Mutex ist nun wieder frei
- Task gibt am Ende wieder freiwillig Rechenzeit ab
- erst nach Ablauf der Periode kann der Zyklus wieder vollständig mit einer „frischen Runtime“ von vorne beginnen

  $\Rightarrow$ Periodendauer darf maximal so groß spezifiziert sein, wie die minimale Zeitspanne zweier aufeinander folgender Ereignisse

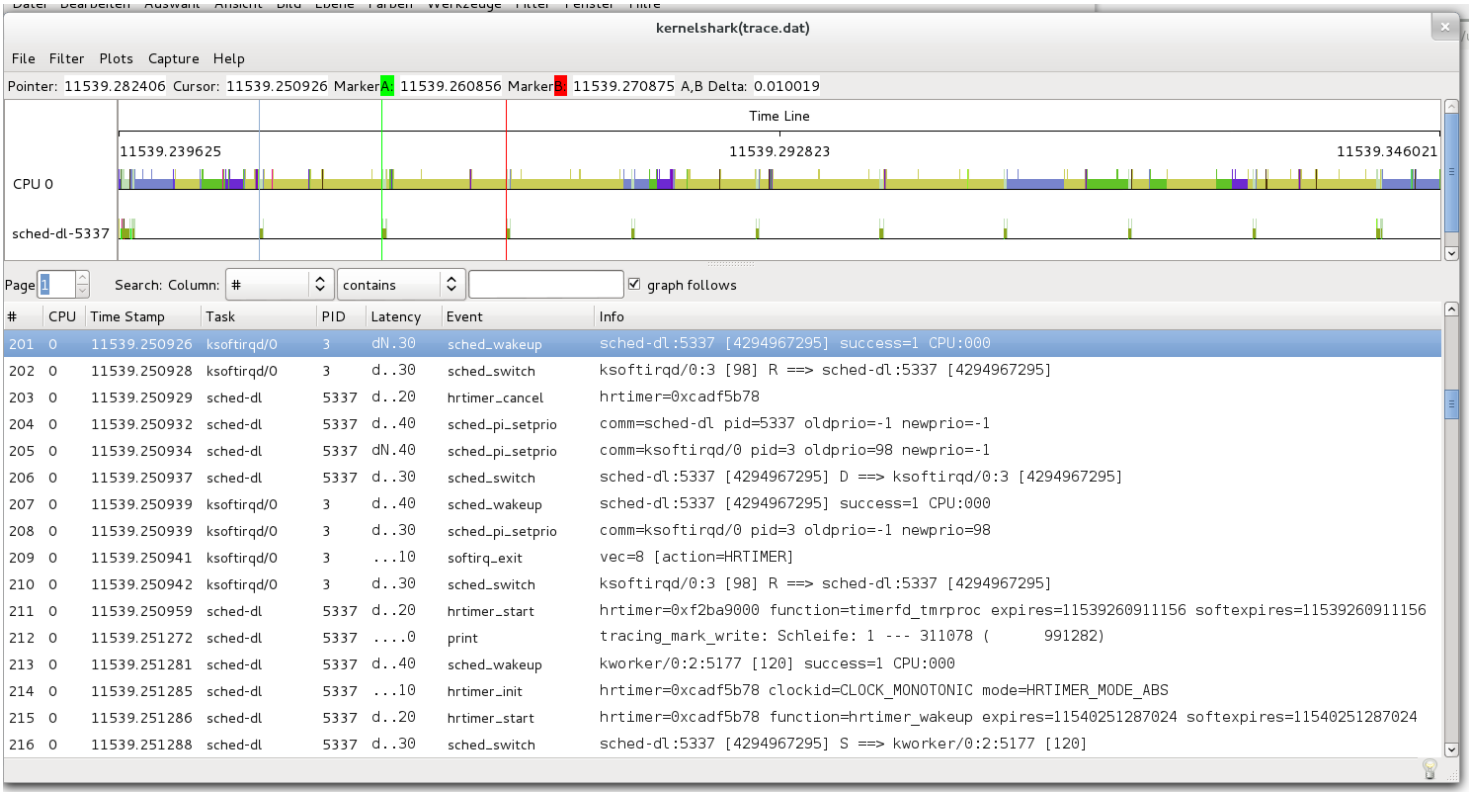
Deadline-Task-Overrun



- Deadline-Task gibt innerhalb der spezifizierten Runtime nicht freiwillig Rechenzeit ab
- Scheduler unterbricht ihn und läßt alle anderen Tasks entsprechend ihrem Scheduling-Verfahren rechnen
- erst nachdem die Periodendauer abgelaufen ist, bekommt der Deadline-Task wieder ein Zeitbudget zugewiesen und kann erneut rechnen

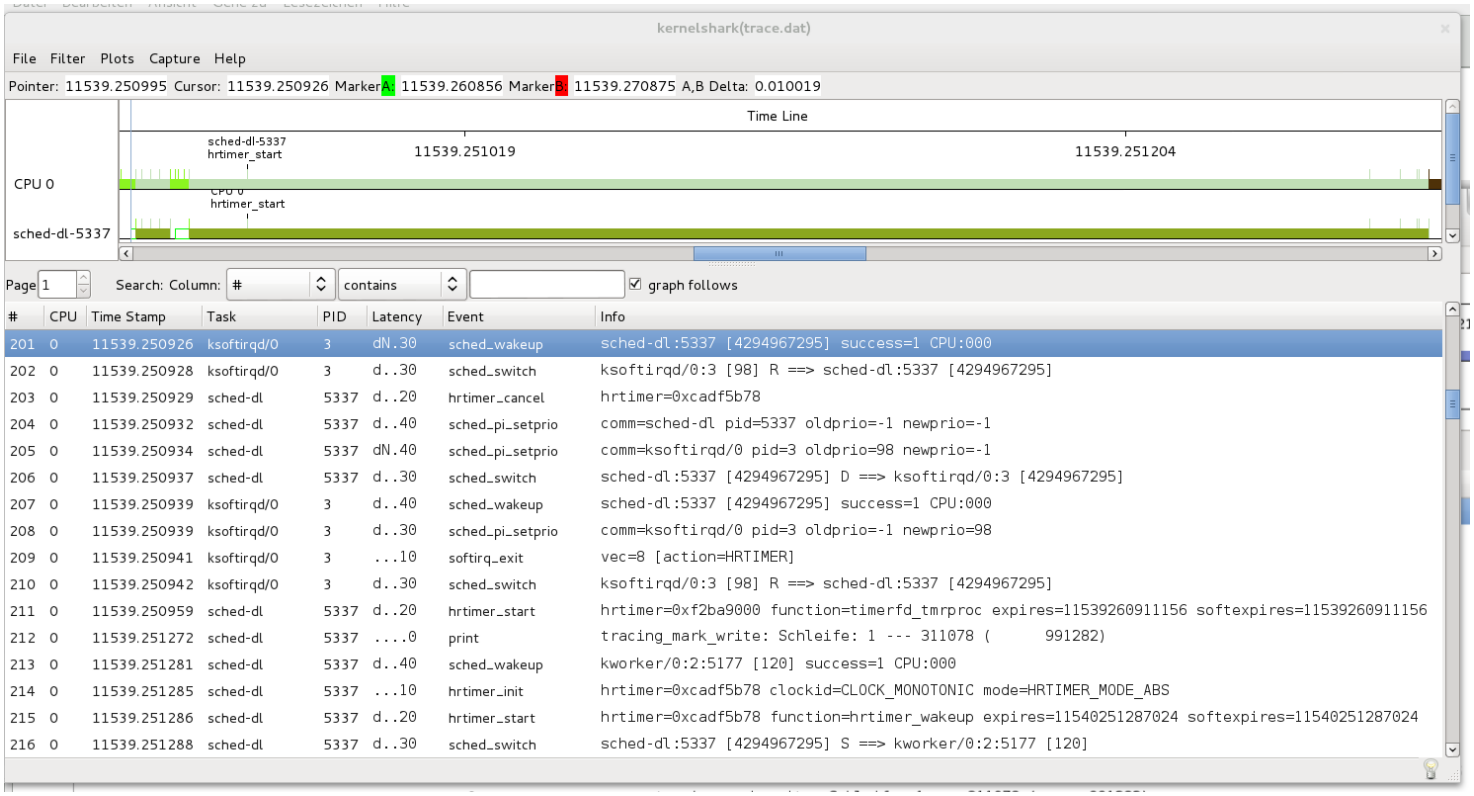
  $\Rightarrow$ für den Deadline-Task spezifizierte Runtime muß mindestens so groß sein wie die Worst-Case-Execution-Time (WCET) pro Periodendauer

zyklischer Deadline-Task (kernelshark)



- Deadline-Task (`sched-dl, pid=5337`) wird von einem Timer zyklisch alle 10*ms* aufgeweckt
- nach dem Aufwecken werden einige Operationen durchgeführt; unter anderem werden Daten auf der Console ausgegeben
- anschließend wartet der Task wieder auf das nächste getriggerte Timerereignis
- auf den ersten Blick sieht alles gut aus
- Aufzeichnung erfolgte mit `trace-cmd` und Visualisierung mit `kernelshark`

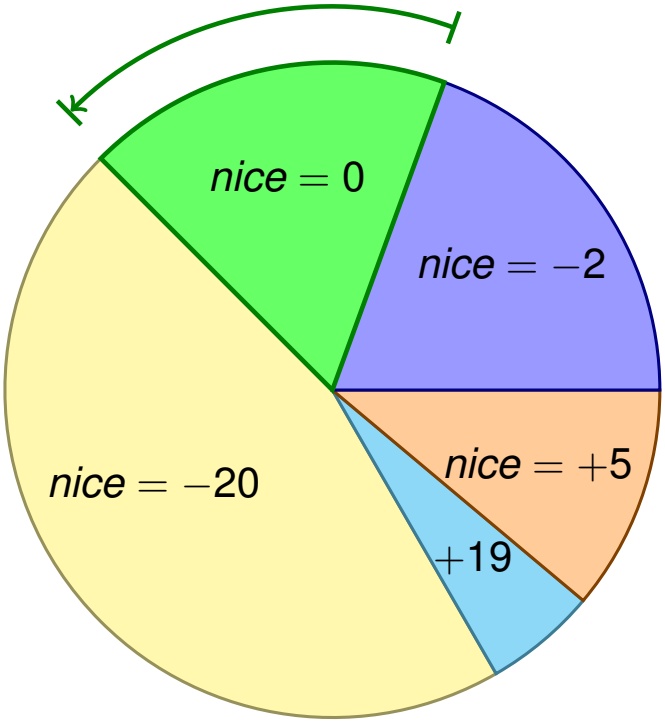
zyklischer Deadline-Task (kernelshark)



- vergrößert man einen einzelnen Aufruf des Deadline-Tasks (`sched-dl, pid=5337`), so stellt sich in diesem Beispiel heraus, dass ein Schleifendurchlauf im Programm nicht ununterbrochen gerechnet wird
- der Deadline-Task wird unterbrochen, weil er beim Aufruf einer Kernelfunktion auf eine Resource warten muß und daher schlafen gelegt wird
- die Tracing-Ausgabe zeigt, daß eine Prioritätsvererbung zu dem blockierenden Task (`ksoftirqd/0, pid=3`) erfolgt
- dies ist am Tracing-Event `sched_pi_setprio` erkennbar

  $\Rightarrow$ $t_{deadline}$ entsprechend korrigieren

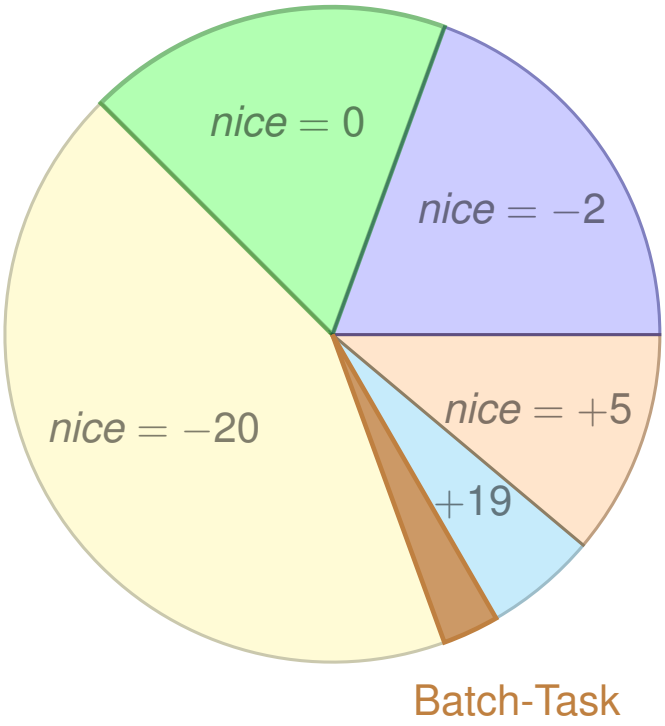
normale Task — Round-Robin-Verfahren



linuxhotel

- Scheduling im Round-Robin-Verfahren mit Zeitscheiben
  ⇒ jeder Task bekommt Rechenzeit
- nice-Wert entscheidet über die Größe der Zeitscheibe
  Zahlenbereich -20 ... $+19$
- pro nice-Wert-Abstufung: ca. 10 % mehr oder weniger Rechenzeit
- Default bei Prozessgenerierung:
  nice-Wert = 0 (wird von Vaterprozeß geerbt)
- „Größe der Torte“ = Summe der Zeitscheiben
  ⇒ je größer die Summe der Zeitscheiben, umso länger dauert es bis ein Task wieder Rechenzeit bekommt

Batch-Task



linuxhotel

- normale Tasks mit der Kennzeichnung „rechenzeitintensiv“
- winzig kleine Zeitscheibe im Round-Robin-Verfahren
  ⇒ bei CPU-Last im Hintergrund
  ⇒ ohne CPU-Last alleinig rechnend
- Batch-Tasks für Hintergrundaufgaben ohne Benutzerinteraktion
- gut geeignet für Prozesse, welche viel CPU-Zeit benötigen und nicht zeitkritisch sind

  Beispiel:
  Kompillierung eines größeren Anwendungssystems

Idle-Task

- Idle-Tasks für Aufgaben, welche komplett im Hintergrund ausgeführt werden können
- eigene Preemption-Ebene: rechnen nur dann, wenn nicht einmal normale Tasks Rechenzeit benötigen
- gut geeignet für Aufträge ohne zeitliche Bindung und zur Entlastung des laufenden Betriebes

  Beispiel:
  Übertragung von Archivdateien an zentralen Server

linuxhotel

Task einstellen I

Shell

- Einstellung der Scheduling-Policy mit `chrt` im Paket `util-linux`
- oder mit `schedtool`: https://github.com/scheduler-tools/schedtool-dl.git
- Abfrage der Scheduling-Policy, Priorität/Nice-Wert aller Tasks:

```
ps -eLo pid,tid,class,rtprio,ni,pcpu,stat,cmd
```

linuxhotel

Task einstellen II

Linux-API

`sched_setattr()`

`sched_getattr()`

`struct sched_attr`

→ wenn Funktion in C-Library nicht vorhanden:
Struktur und Funktion definieren und mittels `syscall()` aufrufen

siehe:
`man 2 sched_setattr`

`sched_setscheduler()` kann nur nicht-Deadline-Tasks einstellen

linuxhotel

CPU-Affinitäten

- Tasks können CPU-Affinitäten zugewiesen bekommen
- Maske von CPU's auf denen der Task rechnen darf
- Affinitätsmaske wird an Kindprozesse weitervererbt
- CPU-Affinität einer Task auf der Shell einstellen:

  `taskset`
- im C-Programm:

  `sched_setaffinity()`
- Kernel-Kommandozeile:

  `isolcpus=`

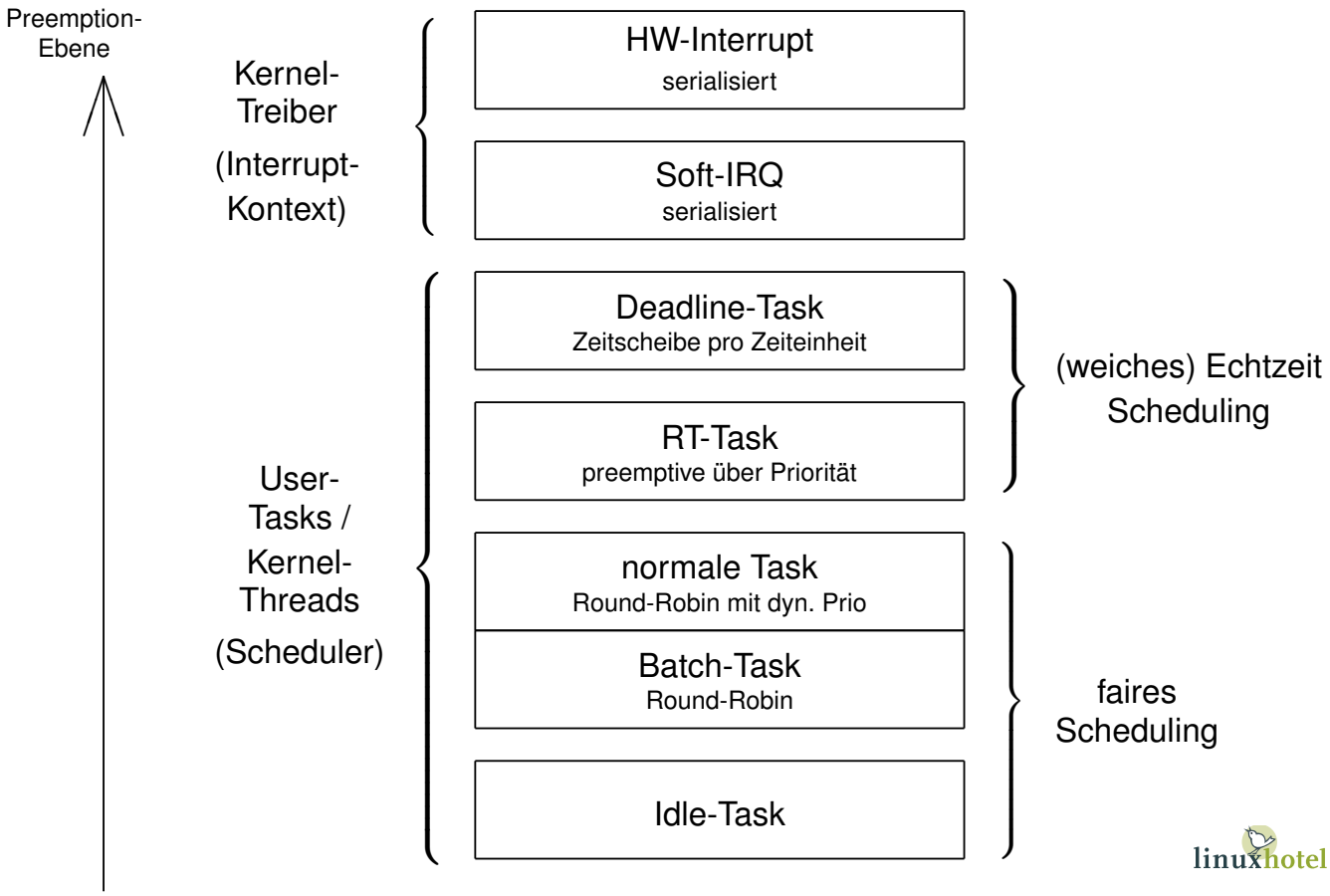
linuxhotel

Preemption-Ebenen im Standard-Kernel

- Hardware-Interrupts
- SoftIRQ's in sequentieller Reihenfolge
- Deadline-Task, gesteuert über Zeitslot
- RT-Task, preemptive über Priorität
- normale Task im Round-Robin-Verfahren
- Batch-Task für rechenintensive Aufgaben
- Idle-Task für Hintergrundaufgaben

linuxhotel

Wer unterbricht wen?



linuxhotel

Beispiel: RT-Scheduling

```
#include <stdio.h>
#include <sched.h>

int main (int argn, char* argv[])
{
  int nErr;
  struct sched_param s;

  s.sched_priority = 25;
  nErr = sched_setscheduler(getpid(), SCHED_FIFO, &s);
  if (nErr == -1)
    return -3;

  sleep (10);

  return 0;
}
```

linuxhotel

Preemption-Ebenen - Übersicht

Mechanismus	Anwendung	Policy	Scheduling untereinander
HW-Interrupt	Kern		serialisiert
SoftIRQ	Kern		serialisiert
RT-Task	Kern User	SCHED_FIFO SCHED_RR Prio: 1 ... 99	prioritätsbasiert unterbrechend
normale Task	Kern User	SCHED_OTHER Prio: 0	Round-Robin Zeitscheibe = f (nice-value)
Batch-Task	Kern User	SCHED_BATCH Prio: 0	Round-Robin
Idle-Task	Kern User	SCHED_IDLE Prio: 0	nicht relevant

Task-Varianten

Task-Typ	Anwendung	Funktionen
Userspace-Prozess	Prozess mit eigenem virtuellen Adressraum	`fork()` `sched_setscheduler()`
Userspace-Thread	Thread mit gemeinsamen virtuellen Adressraum innerhalb von Prozess	`pthread_create` `pthread_setschedparam()`
Kernel-Thread	Kernel-Thread ohne virtuellen Adressraum	`kthread_create()` `sched_setscheduler()`

Kernel-Threads I

- werden als Task ausgeführt und liegen damit im „normalen“ Scheduling-Verfahren
  $\Rightarrow$ dürfen lange rechnen, schlafen und blockieren
- werden im *Prozess-Kontext* ausgeführt
- haben im Gegensatz zu Userspace-Tasks keinen virtuellen Adressbereich
- sind in der Prozess-Liste sichtbar, z. B. bei `ps aux` mit eckigen Klammern

Kernel-Threads II

```
PID   USER     COMMAND
    1 root     init
    4 root     [kworker/0:0]
    5 root     [kworker/u:0]
  664 root     [mtdblockd]
  688 root     [orion_spi]
  742 root     /usr/sbin/dropbear
  745 root     /sbin/syslogd -n -m 0
  746 root     /sbin/klogd -n
  747 root     /usr/sbin/dropbear
  ...
```

linuxhotel

Beispiel: Kernel-Threads I

```
static struct task_struct* kio_thr;


int kio_fkt (void* data)
{

  while (!kthread_should_stop())
  {
    printk (KERN_DEBUG "kio_fkt\n");
    msleep (1000);
  }

  return 0;
}
```

linuxhotel

Beispiel: Kernel-Threads II

```
int __init kio_init (void)
{
  kio_thr = kthread_run (kio_fkt, NULL, "kio-thread");

  if (IS_ERR(kio_thr))
  {
    printk (KERN_WARNING "kio_init: err: %ld()\n",
                PTR_ERR(kio_thr));
  }
  return 0;
}

void __exit kio_exit (void)
{
  kthread_stop (kio_thr);
}
```

linuxhotel

Workqueue

- spezielle Kernel-Threads mit Listen (Workqueue) von abzuarbeitenden Arbeiten (Work)
- eine *Workqueue* sind Kernel-Threads, welche auf abzuarbeitende Arbeiten warten und diese nacheinander abarbeiten
- es gibt Threads mit festgelegter CPU-Nummer oder auch ohne CPU-Bindung (unspecified)
- eine *Work* ist eine Funktion, die bis zum Ende abgearbeitet wird
- anschliessend wird die nächste Arbeit abgearbeitet

linuxhotel

Was fehlt dem Standard-Kernel zu harter Echtzeit?

- Unterteilung der Interrupts nach Echtzeitrelevanz nicht vorgesehen
  $\Rightarrow$ Abarbeitung von Interrupts und SoftIRQ's verzögern darunter angeordnete RT-Tasks
- Synchronisierungsmechanismen sind nicht unterbrechbar
  $\Rightarrow$ beliebiger Task kann echtzeitrelevanten blockieren
- bei SMP-Systemen können den Tasks CPU-Affinitäten zugeordnet werden
- keine verbindliche Aussage über Deadline-Zeiten möglich
  $\Rightarrow$ maximal Soft-RT-Aufgaben bei geschickter Wahl der Mechanismen

Interruptbearbeitung I

- Hardware-Interrupts unterbrechen die Abarbeitung des gerade laufenden Tasks
- CPU-Register werden gesichert und der Interrupt wird abgearbeitet
- man spricht vom *Interrupt-Kontext*
- weitere Verarbeitung im System wird erst fortgesetzt, wenn Interrupt beendet wurde
  $\Rightarrow$ ISR darf nicht zu lange rechnen
- wenn Interrupt zu lange rechnet:
  - weitere Interrupts gehen verloren; z. B. Inkrementalgeber
  - Interrupt-Delay steigt

Interruptbearbeitung II

- Linux unterstützt *Shared Interrupts*, also mehrere ISR's auf der gleichen Interrupt-Nummer
- Linux ruft die zu einer Interrupt-Nummer gehörenden Interrupt-Service-Routinen (ISR) nacheinander auf
- es werden immer alle ISR zu einer Interrupt-Nummer aufgerufen
- falls Architektur Interrupt-Prioritäten kennt, werden diese von Linux unterstützt (Nested Interrupts)

Interrupt-Nacharbeit

- weitere Interrupts werden aufgerufen
- SoftIRQ's werden gestartet
- Scheduler wird aufgerufen und ggf. erfolgt ein Re-Scheduling

Interrupt-Programmierung I

- Interrupts werden mit der Funktion `request_irq()` registriert
  → Interrupt-Nummer und Zeiger auf ISR-Funktion wird übergeben
- `dev_id` ist `void*`-Zeiger auf Daten des Kernel-Moduls
  → wird als Parameter an die ISR übergeben
- Linux-System benötigt `dev_id` zum Deregistrieren einer ISR bei Shared Interrupts
- ISR kann dadurch reentrant gestaltet werden:
  ein ISR-Code und mehrere registrierte Interrupts

linuxhotel

Interrupt-Programmierung II

Flag	Beschreibung
`IRQF_SHARED`	Shared-Interrupt
`IRQF_DISABLED`	Interrupts sind disabled wenn ISR aufgerufen wird
`IRQF_NODELAY`	*RT-Patch*: nicht-threaded ISR (2.6)
`IRQF_NO_THREAD`	*RT-Patch*: nicht-threaded ISR bei Cmd-Line `threadirqs` (ab 3.0)
`IRQF_TRIGGER_RISING`	auf steigende Flanke triggern
`IRQF_TRIGGER_FALLING`	auf fallende Flanke triggern
`IRQF_TRIGGER_HIGH`	Level-Trigger-High
`IRQF_TRIGGER_LOW`	Level-Trigger-Low

- Deregistrieren von Interrupts erfolgt mit `free_irq()`

linuxhotel

Interrupt-Programmierung III

- Interrupts können mit Funktion `disable_irq()` und `enable_irq()` unter Angabe der Interrupt-Nummer disabled und wieder enabled werden
- `local_irq_save()` sichert die Interrupt-Maske und disabled Interrupts auf der lokalen CPU; `local_irq_restore()` stellt die Interrupt-Maske wieder her
- reservierte Interrupts und Anzahl aufgetretener Interrupts: `cat /proc/interrupts`

linuxhotel

Beispiel: Interrupt-Registrierung I

```
static int irqnr = 7;
struct kio_dev
{
  int data;
};

static struct kio_dev *pdev;

irqreturn_t kio_interrupt (int irq, void* dev_id)
{
  struct kio_dev* dev = (struct kio_dev*) dev_id;
  printk (KERN_INFO "ISR: %d, dev_id: %p, data=%d\n",
                        irq, dev_id, dev->data);

  return IRQ_HANDLED;
}
```

linuxhotel

Beispiel: Interrupt-Registrierung II

```
int __init kio_init (void)
{
  pdev = kmalloc(sizeof(struct kio_dev), GFP_KERNEL);
  if (request_irq (irqnr, kio_interrupt,
                   IRQF_SHARED | IRQF_TRIGGER_RISING,
                   "kio", pdev))
  {
    printk (KERN_ERR "kio_init: Interrupt belegt\n");
  }
  return 0;
}

void __exit kio_exit (void)
{
  free_irq (irqnr, pdev);
  kfree (pdev);
}
```

linuxhotel

Sekundärreaktionen auf Interrupts

- Interrupt darf nur kurze Zeit rechnen
  ⇒ Sekundärreaktion notwendig
- Benachrichtigung, „Aufwecken“ von wartenden Tasks (Warteschlange)
- Funktionalität in „Verlängerung der ISR“ auslagern
  ⇒ SoftIRQ
- Schedulen der Funktionalität (Kernel-Thread, Workqueue)

linuxhotel

SoftIRQ's I

- SoftIRQ's werden bevorzugt vor dem Scheduling ausgeführt
- unterbrechen sich auf einer CPU nicht gegenseitig
  → serialisiert nacheinander abgearbeitet
- werden nur durch Hardware-Interrupt unterbrochen und anschliessend fortgesetzt
- kommen unmittelbar nach Abarbeitung der Interrupts zum laufen
- dürfen nur begrenzt lange rechnen, da keine Tasks gescheduled werden
- *Hauptvorteil:* weitere Interrupts können empfangen werden
- SoftIRQ's befinden sich im *Interrupt-Kontext*

linuxhotel

SoftIRQ's II

- sie dürfen auf keinen Fall schlafen oder blockieren
- dienen als Basismechanismus:
  * High-Priority-Tasklet
  * Kernel-Timer
  - Netzwerk-Send
  - Netzwerk-Receive
  - Blockdevices
  * Tasklet
  - Scheduling
  * hrtimer
  - RCU (Read, Copy, Update)

linuxhotel

Tasklet

- basieren auf SoftIRQ's
- zusätzlich serialisiert gegen sich selber; auch bei mehreren CPU's wird ein und dasselbe Tasklet nur einmal ausgeführt
- wiederholt gestartetes Tasklet wird nur einmal ausgeführt, solange es noch nicht begonnen wurde
- Re-Starten des eigenen Tasklets ist nicht zulässig
- Tasklet wird mit `tasklet_schedule()` und High-Priority-Tasklet mit `tasklet_hi_schedule()` gestartet

linuxhotel

Beispiel: Tasklet

```
static DECLARE_TASKLET (kio_tasklet, kio_task, 0);

void kio_task (unsigned long data)
{
  printk (KERN_INFO "kio_task; data: %d\n", data);
}

irqreturn_t kio_interrupt (int irq, void* dev_id)
{
  kio_tasklet.data = 0xBEEF;
  tasklet_schedule (&kio_tasklet);

  return IRQ_HANDLED;
}
```

linuxhotel

Kernel-Timer I

- basieren auf SoftIRQ's
- werden zu einem bestimmten Zeitpunkt in `jiffies` gestartet
- `jiffies` ist unsigned 32-Bit Zähler, inkrementiert mit Frequenz `HZ`
- `HZ` kann 100, 250 oder 1000 sein
- aktueller Zeitpunkt liegt in globaler Variable `jiffies`
- mit `add_timer()` wird Kernel-Timer aktiviert; wenn Timer schon existiert wird Fehler generiert (`BUG_ON`)
- `mod_timer()` löscht Timer, setzt Ablaufzeitpunkt und fügt Timer hinzu (keine Probleme, falls Timer schon aktiv)
- `del_timer()` bzw. `del_timer_sync()` löschen Timer

linuxhotel

Beispiel: Kernel-Timer I

```
static struct timer_list kio_timer;

void kio_tmr_fkt (unsigned long data)
{
  printk (KERN_INFO "kio_tmr_fkt; data: %d\n", data);
}

irqreturn_t kio_interrupt (int irq, void* dev_id)
{
  kio_timer.function = kio_tmr_fkt;
  kio_timer.data = 0xAFFE;
  kio_timer.expires = jiffies + 3 * HZ;
  add_timer (&kio_timer);

  return IRQ_HANDLED;
}
```

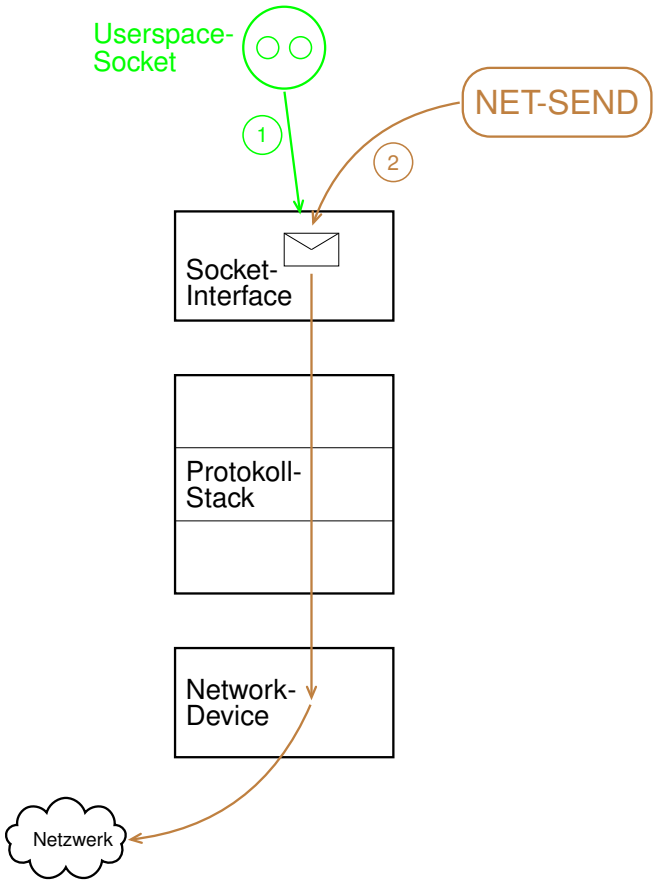
linuxhotel

Beispiel: Kernel-Timer II

```
int kio_init (void)
{
  init_timer (&kio_timer);

  return 0;
}
```

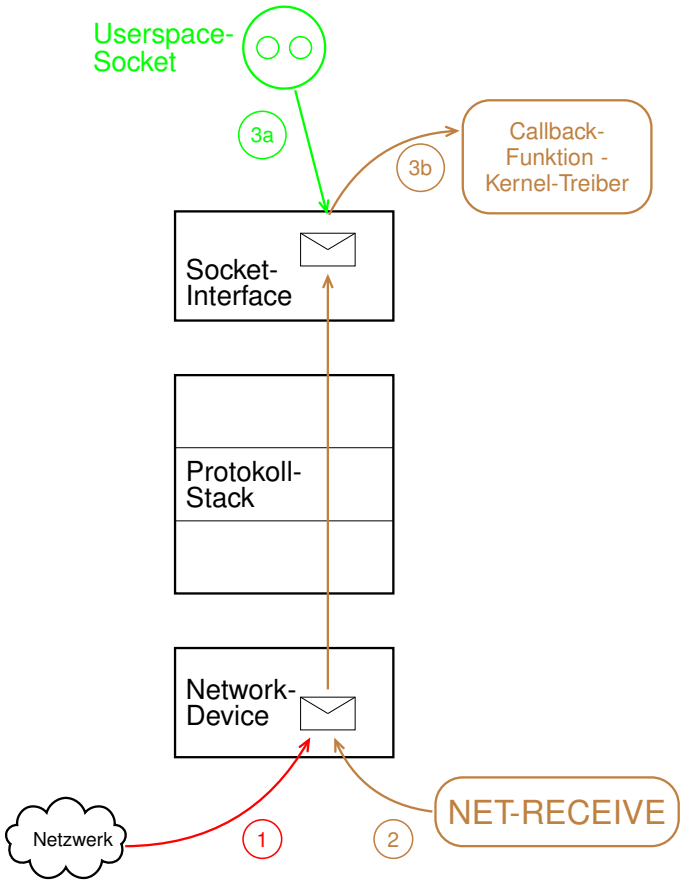
linuxhotel

Protokoll-Stack - Net-Send



linuxhotel

Protokoll-Stack - Net-Receive



linuxhotel

8 Memory-Management
- physikaler Speicher
- GFP-Flags
- Buddy-System
- Speicher-Migrationstyp
- Slab-Allocator
- Kernel-Malloc
- Userspace-Speicher
- Datenaustausch — Userspace $\leftrightarrow$ Kernel

physikalischer Speicher I

- für jede physikalisch vorhandene Speicherseite gibt es eine Verwaltungsstruktur `struct page`
- Allokierung von Speicher erfolgt in der kleinsten Granularität von einer Page
- jede physikalisch vorhandene Page wird als `struct page` verwaltet
- von 4 GiB Addressbereich bei 32 Bit-Systemen gehören 3 GiB zum virtuellen Addressbereich des Userspace
- restliche 1 GiB gliedert sich in den direkt gemappten Bereich (896 MiB bei x86) und die *High-Memory-Zone*
- direkt gemappte Bereich untergliedert sich in die Zonen *DMA* und *Normal*

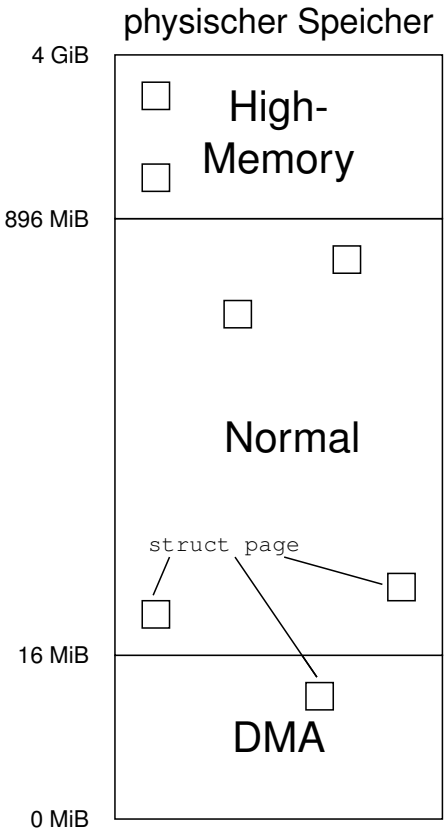
physikalischer Speicher II

- High-Memory-Pages können nicht direkt gemappt werden und erfordern daher einen Eintrag in der Page-Table
  $\Rightarrow$ vorwiegend für Userspace-Prozesse
- physikalische Speicher-Allozierung erfolgt grundsätzlich in Einheiten von 2^{order} Pages mit `order = 0 ... (MAX_ORDER - 1)`
- bei 32-Bit-Systemen hat `MAX_ORDER` typisch den Wert `11`; also $2^0..2^{10}(1024)$ Speicherseiten
- Verhalten der Speicher-Allozierung muß mittels *GFP-Flags* definiert werden:
  - Zone(n)
  - Verhaltensmimik (blockierend, Notfall-Reserve, ...)

physikalischer Speicher III

- *Funktionen:*
  `alloc_pages()`: liefert `struct page*`
  `__get_free_pages()`: liefert Adresse
  `free_pages()`: benutzt Adresse
  `__free_pages()`: benutzt `struct page*`
- Zonen-Information:
  `/proc/meminfo`
  `/proc/zoneinfo`

Memory-Layout



nicht direkt gemappt
$\Longrightarrow$ Page-Table-Entries
(Userspace, `vmalloc()`)

direkt gemappt
$\Longrightarrow$ bevorzugt für Kernel-Treiber

direkt gemappt
$\Longrightarrow$ Direct Memory Access
$\longrightarrow$ Gerät schreibt in RAM ohne MMU

linuxhotel

Speicher-Allozierung: GFP-Flags I

- GFP-Flags steuern in welcher Zone Speicher angefordert wird:
  - Zone DMA (`__GFP_DMA`) bedeutet, es muß DMA-fähiger Speicher angefordert werden
  - Zone High-Mem (`__GFP_HIGHMEM`) läßt die Zonen in der Reihenfolge High-Memory, Normal und DMA zu
  - keine Zonenangabe heißt bevorzugt Zone Normal und alternativ Zone DMA

linuxhotel

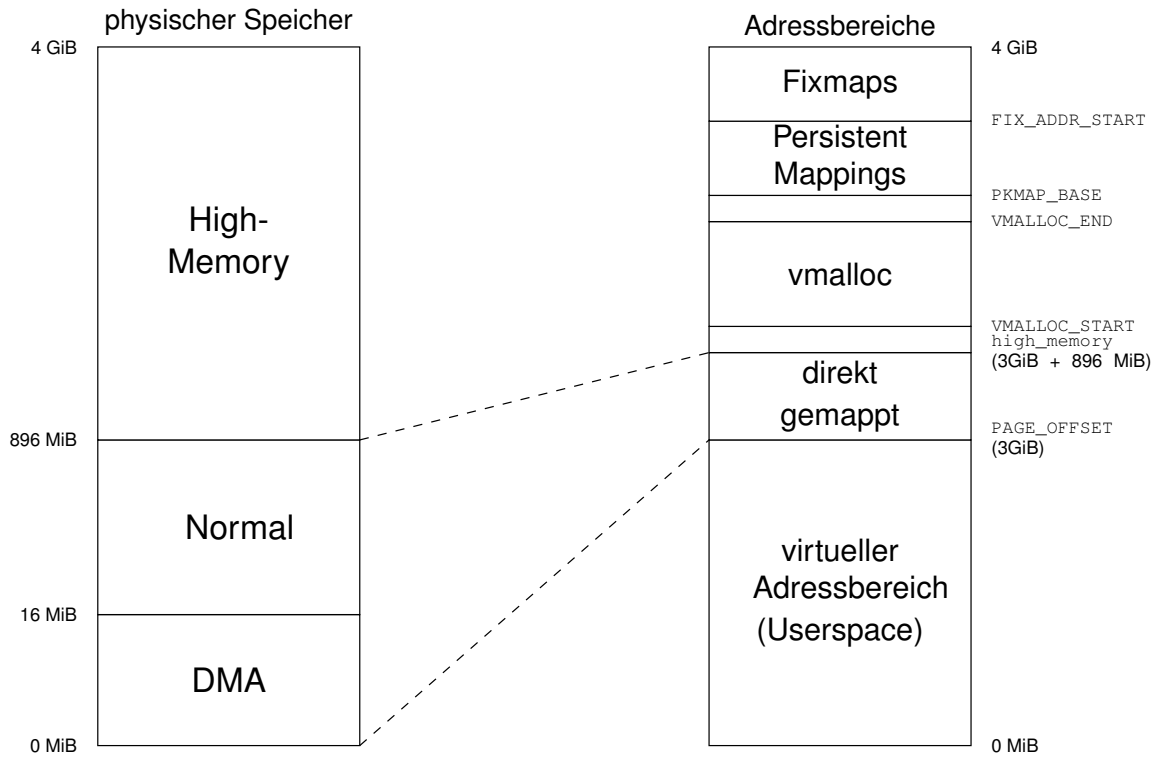
Speicher-Allozierung: GFP-Flags II

- Aktion-Flags bestimmen die Verhaltensmimik:
  - mit `__GFP_WAIT` wird mitgeteilt, dass die Allokation warten darf bis beispielsweise genügend zusammenhängender Speicher verfügbar ist
  - `__GFP_REPEAT` wiederholt fehlgeschlagene Allokationen einige mal und `__GFP_NOFAIL` wiederholt für immer, bis die Allokation funktioniert
  - `__GFP_HIGH` (High Priority) gibt den Zugriff auf Notfall-Reserven frei
  - `__GFP_ZERO` führt zur Rückgabe von genullten Speicherseiten

Speicher-Allozierung: GFP-Flags III

- vordefinierte, bereits zweckmässig veroderte Flag-Kombinationen:
  - `GFP_KERNEL` kann warten, greift nicht auf Notfall-Reserven zu und sucht in den Zonen Normal und DMA; häufigste Speicher-Allokation im Kernel
    $\Rightarrow$ Prozess-Kontext (Dateioperationen, Kernel-Thread)
  - `GFP_ATOMIC` blockiert garantiert nicht, greift ggf. auf Notfall-Reserven zu und sucht in den Zonen Normal und DMA; sollte nur dann verwendet werden, wenn unbedingt notwendig
    $\Rightarrow$ Interrupt-Kontext (ISR, SoftIRQ)
  - `GFP_DMA` wird mit `GFP_KERNEL` oder `GFP_ATOMIC` verodert und schränkt die Suche auf die Zone DMA ein
    $\Rightarrow$ DMA-Geräte

Kernel-Adress-Raum



linuxhotel

Buddy-System I

- Grundlage der Speicherallozierung ist das *Buddy-System*
- physikalisch aufeinander folgende und nicht genutzte Speicherseiten werden als zusammenhängender freier Speicherbereich verwaltet
- ein Buddy ist die Verwaltung von 2^{order} zusammenhängenden und freien Pages
- für jeden Wert von `order` im Bereich `0 .. (MAX_ORDER - 1)` gibt es eine Liste mit den zugehörigen Buddies:
  1 *Liste mit* $2^0 = 1$ *Page*
  1 *Liste mit* $2^1 = 2$ *Pages*
  1 *Liste mit* $2^2 = 4$ *Pages*
  1 *Liste mit* $2^3 = 8$ *Pages*
  ...

linuxhotel

Buddy-System II

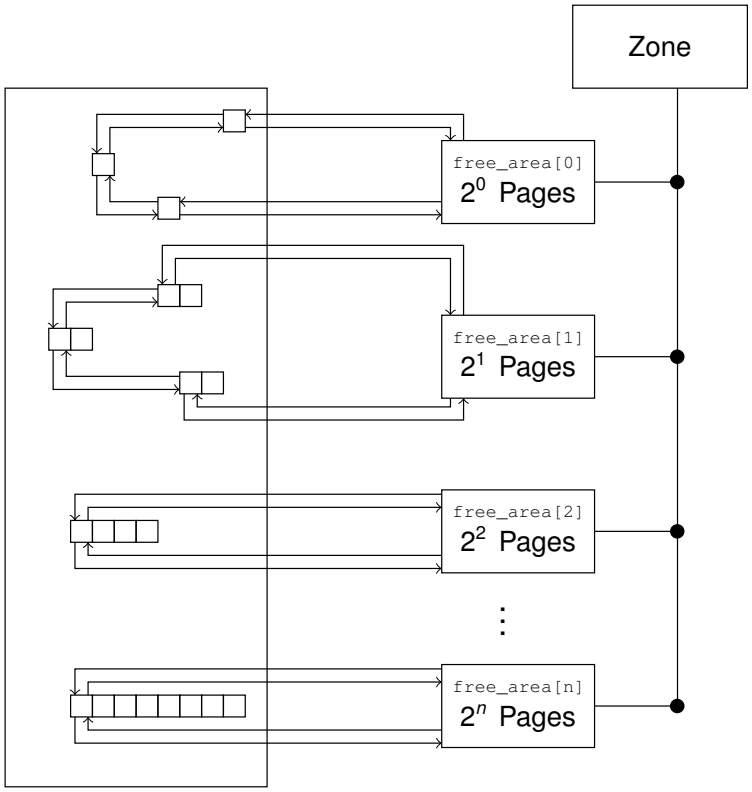
- beim Allozieren von Speicher wird versucht, einen möglichst passenden Buddy zu verwenden; wenn Speicherseiten übrig bleiben, werden diese in andere Buddy-Listen eingehängt
- beim Freigeben von Speicherseiten wird versucht, zusammen mit den vorhandenen Buddies einen neuen möglichst großen Buddy zu erzeugen
- Buddy-Listen existieren jeweils einmal für eine Speicher-Zone; sprich ein Satz von Buddy-Listen für Zone DMA, ein Satz für Zone Normal und ein Satz für Zone High-Memory
- diese Zonen-Buddy-Listen gibt es wiederum einmal pro NUMA-Node bei Multiprozessor-Systemen

Buddy-System III

- Beispiel:
  Anzeige der Anzahl an Buddys in den vorhandenen Zonen mit jeweils 2^0 2^1 2^2 ... 2^{10} Pages

```
kdev:~ # cat /proc/buddyinfo

Node 0, zone      DMA   3   3   2   1   1   1   1   0   1   1   3
Node 0, zone   Normal   3   2   3   2   0   3   2   2   0   0 199
Node 0, zone HighMem 71 56 43 29 15   5   5   1   3   1 230
```

Buddy-Listen



linuxhotel

Speicher-Migrationstyp I

- Vermeidung von Speicher-Fragmentierung durch Einteilung des Speichers in Migrationsbereiche
- *Strategie: Vermeidung der Allokation von einzelnen „verschiebbaren“ Speicherseiten innerhalb eines „nicht-verschiebbaren“*
- virtueller Userspace-Speicher kann beliebig verschoben werden durch Änderung der Page-Table-Einträge
- direkt gemappter Kernel-Speicher kann nicht verschoben werden

linuxhotel

Speicher-Migrationstyp II

Migrationstypen

- `MIGRATE_UNMOVABLE`: nicht verschiebbar (Bsp.: direkt gemappter Kernel-Speicher)
- `MIGRATE_RECLAIMABLE`: wiederherstellbare Speicherseiten (Bsp.: gemappte Dateien)
- `MIGRATE_MOVABLE`: verschiebbare Speicherseiten (ändern der Page-Table-Einträge)
- `MIGRATE_RESERVE`: Notfall-Reserve bei Speicherverknappung
- `MIGRATE_ISOLATE`: Verschiebe-Reserve bei SMP-Systemen

linuxhotel

Speicher-Migrationstyp III

Beispiel

Anzeige der freien Buddies bezogen auf Migrationstyp, Zone und Node für jeweils 2^0, 2^1, 2^2, ..., 2^{10} Pages

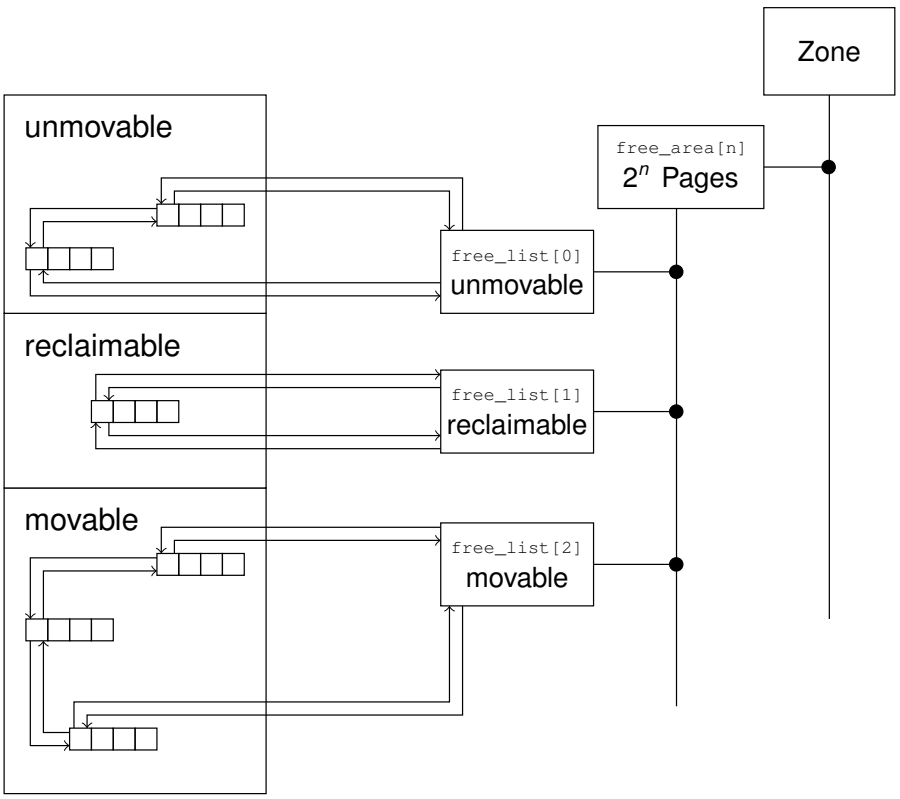
linuxhotel

Speicher-Migrationstyp IV

```
kdev:~ # cat /proc/pagetypeinfo

Page block order: 10
Pages per block:  1024

Free pages count per migrate type at ord 0    1    2    3    4    5    6    7    8    9  10
Node 0, zone      DMA, type    Unmovable    0    0    0    0    0    0    0    0    0    0    1
Node 0, zone      DMA, type  Reclaimable    0    0    0    0    0    0    0    0    0    0    0
Node 0, zone      DMA, type      Movable    3    3    2    1    1    1    1    0    1    1    1
Node 0, zone      DMA, type      Reserve    0    0    0    0    0    0    0    0    0    0    1
Node 0, zone      DMA, type      Isolate    0    0    0    0    0    0    0    0    0    0    0
Node 0, zone   Normal, type    Unmovable    1    0    1    1    0    1    1    0    0    0    0
Node 0, zone   Normal, type  Reclaimable    1    0    0    0    0    1    0    1    0    0    0
Node 0, zone   Normal, type      Movable    1    1    1    1    0    1    1    1    0    0  198
Node 0, zone   Normal, type      Reserve    0    0    0    0    0    0    0    0    0    0    1
Node 0, zone   Normal, type      Isolate    0    0    0    0    0    0    0    0    0    0    0
Node 0, zone HighMem, type    Unmovable    0    1    0    0    0    0    0    0    0    1    0
Node 0, zone HighMem, type  Reclaimable    0    0    0    0    0    0    0    0    0    0    0
Node 0, zone HighMem, type      Movable   81   55   42   28   15    5    5    1    3    0  230
Node 0, zone HighMem, type      Reserve    0    1    0    0    0    0    0    0    0    0    0
Node 0, zone HighMem, type      Isolate    0    0    0    0    0    0    0    0    0    0    0

Number of blocks type     Unmovable  Reclaimable     Movable     Reserve     Isolate
Node 0, zone      DMA             1            0           2           1           0
Node 0, zone   Normal             6            3         208           1           0
Node 0, zone  HighMem             1            0         288           1           0
```

linuxhotel

Buddy-Migrationstyp



linuxhotel

Slab-Allocator I

- Slab-Allocator ist ein Objekt-Cache; Objekte innerhalb von Pages und über mehrere Pages werden verwaltet
- ein Slab (engl.: Platte, Fliese) besteht aus 1 ... n Page(s); je nach Objektgröße
- für häufig benutzte Objekte wird nicht jedesmal Speicher alloziert und freigegeben, sondern einmalig und allozierter Speicher wiederverwendet
  → höhere Performance durch weniger Allokationen
  → geringere Speicher-Fragmentierung
- Objekt-Größen kleiner als eine Page werden effizient verwaltet
  → weniger Speicherverlust

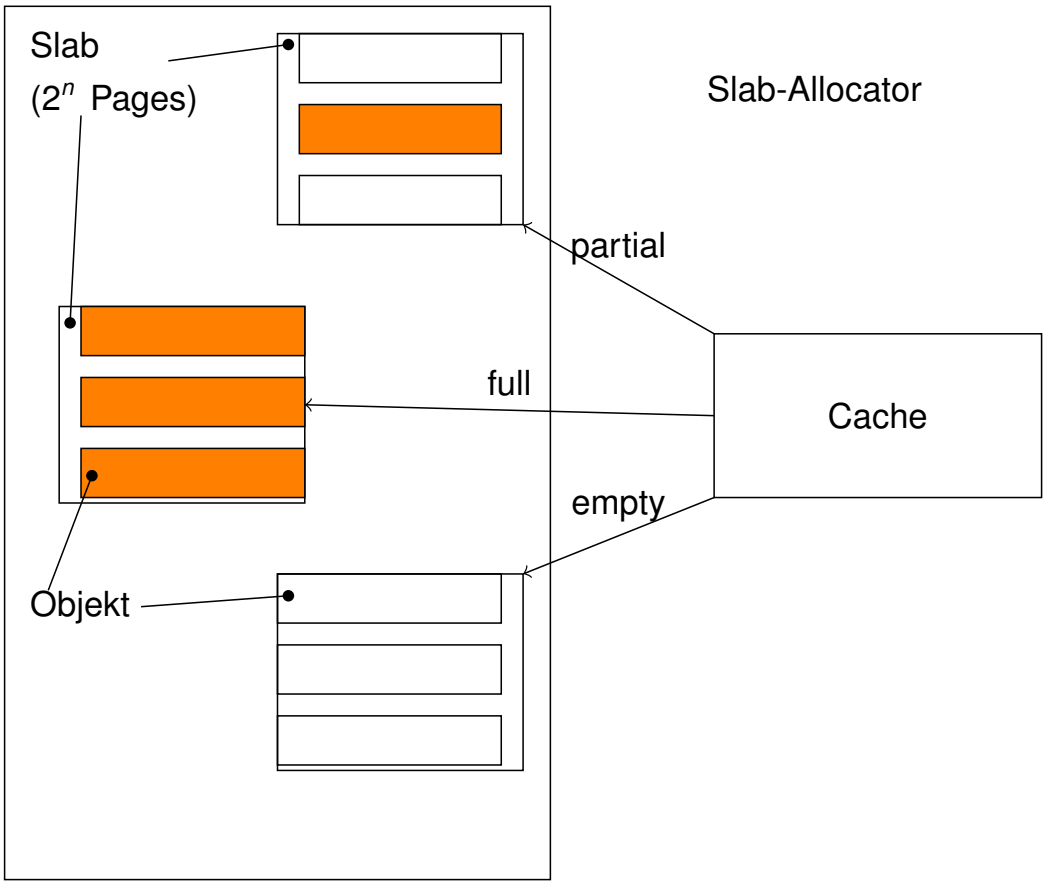
linuxhotel

Slab-Allocator II

- eigentliche Cache verwaltet die Slabs sowie die darin angelegten oder freien Objekte
- Slabs können unterschiedlich „gefärbt“ sein: auf unterschiedlichen Cache-Lines der CPU liegen
- Slabinformationen:
  `/proc/slabinfo`

linuxhotel

Slab-Allocator



linuxhotel

Beispiel: Slab-Allocator I

```
#include <linux/slab.h>

static struct kmem_cache* kio_cache;

struct kio_struct
{
  int i;
  char c[20];
};

// Slab-Allocator anlegen
kio_cache = kmem_cache_create ("kio-cache",
            sizeof(struct kio_struct),
            0, 0, NULL);
```

linuxhotel

Beispiel: Slab-Allocator II

```
// Slab-Allocator verwenden
struct kio_struct* kio1, kio2;

kio1 = (struct kio_struct*) kmem_cache_alloc (
                        kio_cache, GFP_KERNEL);
kio2 = (struct kio_struct*) kmem_cache_alloc (
                        kio_cache, GFP_KERNEL);

kmem_cache_free (kio_cache, kio1);
kmem_cache_free (kio_cache, kio2);


// Slab-Allocator freigeben
kmem_cache_destroy (kio_cache);
```

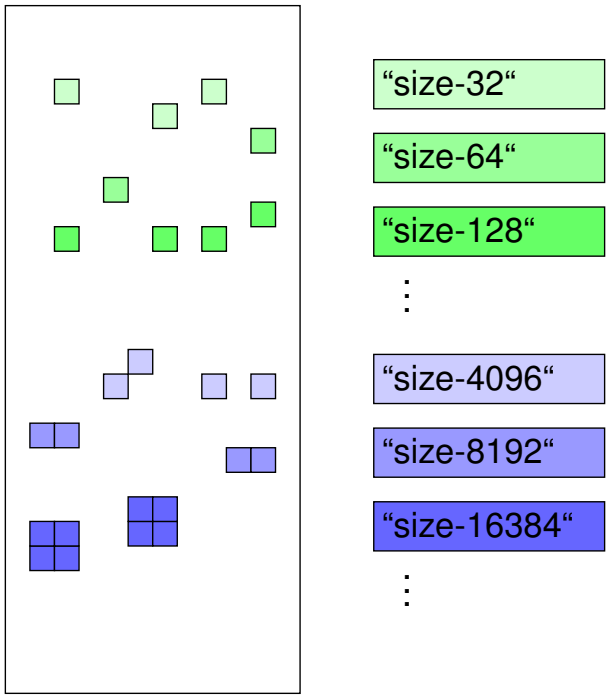
linuxhotel

Kernel-Malloc

- die Funktion `kmalloc()` dient als „Standard“-Allokationsfunktion in Kernel-Treibern
- Funktion greift auf bereits vordefinierte Slab-Allokatoren in der Größe 32 Byte, 64 Byte, ... zu; es findet also nicht unmittelbar eine Allokation statt, sondern es wird versucht, ein Objekt von einem Slab-Allocator zu holen
- Speicherplatz zwischen der Objektgröße und dem gewählten Slab-Allocator-Objekt bleibt ungenutzt
- übergebenen GFP-Flags werden über den Slab-Allocator bis hin zur eigentlichen Speicher-Allokation durchgereicht wenn kein Objekt mehr frei ist

linuxhotel

Slab-Allocator und `kmalloc()`




```
void* kmalloc(size,flags);
```

- Objekt von passendsten Slab-Allocator anfordern
- flags beziehen sich auf eventuelle Cache-Erweiterung

linuxhotel

Kernel-Virtual-Malloc

- `vmalloc()` liefert innerhalb des Kernels virtuell zusammenhängenden Speicher
- Speicherseiten werden mittels der Page-Table als zusammenhängender Speicherbereich virtuell gemapped
- Mapping erfolgt innerhalb des Bereiches `VMALLOC_START` und `VMALLOC_END` im 1 GiB großen Kernel-Adressraum
- Anzeige der vmalloc-Bereiche: `/proc/vmallocinfo`

linuxhotel

virtueller Adressraum des Userspace I

- Userspace-Adressen sind rein virtuell in einem 3 GiB großen Adressbereich abgebildet
- die `task_struct` eines Prozesses zeigt auf eine `mm_struct`, welche die Speicherbereiche des Prozesses enthält; unterschiedliche `task_struct`-Objekte unterschiedlicher Threads eines Prozesses zeigen auf die gleiche `mm_struct`
- `mm_struct` zeigt auf das *Page Global Directory*, dieses wiederum auf das *Page Upper Directory*, dann auf das *Page Middle Directory* und schließlich zeigt das *Page Table Entry* auf die eigentliche Speicher-Page
- vierstufiges Page-Directory-System; bei dreistufigen Architekturen entfällt das Page-Upper-Directory

linuxhotel

virtueller Adressraum des Userspace II

- 32 Bit einer virtuellen Adresse teilen sich in die vorhandenen Directory-Stufen sowie dem Offset innerhalb der Page auf
- Vorteil der mehrstufigen Page-Directories liegt im geringeren Speicherverbrauch, da vom Prozess nicht benutzte Directories nicht angelegt werden
- ein zusammenhängender Speicherbereich mit gleichen Rechten ist in einer `vm_area_struct` (VMA) dargestellt
- Anzeige der VMA's eines Prozesses: `/proc/<pid>/maps`
- Mapping der virtuellen Adressen auf phys. Pages (binär, siehe `linux/Documentation/vm/pagemap.txt`):

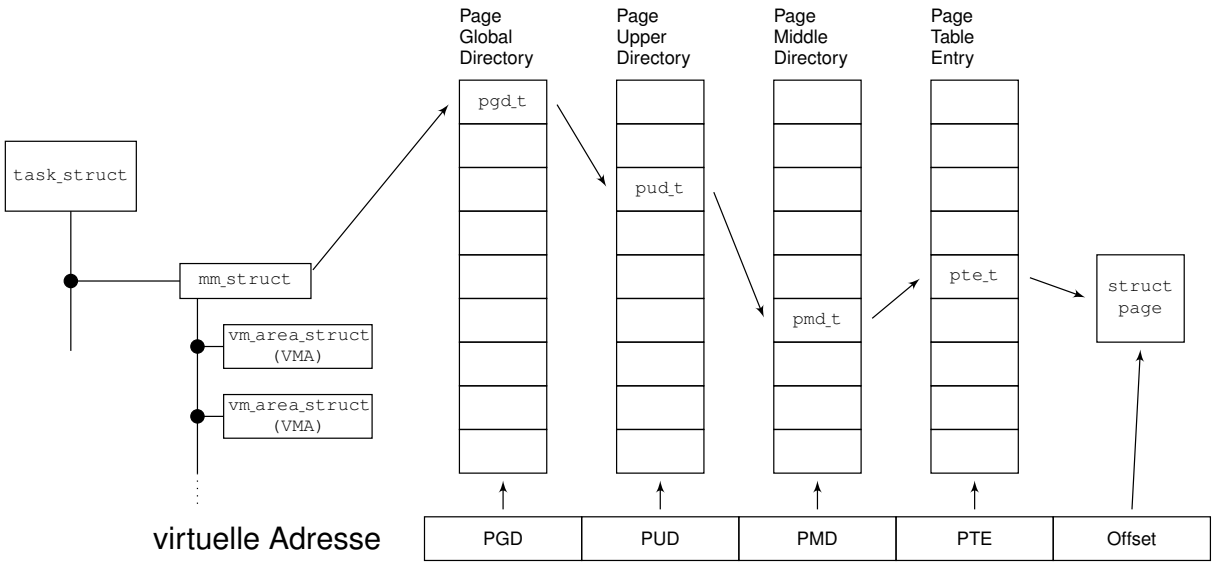
  `/proc/<pid>/pagemap`

linuxhotel

virtueller Adressraum des Userspace III

- Page-Flags:
  `/proc/kpageflags`

Userspace-Adressen



Page-Map des Userspace anzeigen

```
kdev:~ # cd linux/Documentation/vm
kdev: # make page-types
kdev: # ./page-types -h
kdev: # ./page-types

# Beschreibung des Tools
kdev: # less pagemap.txt
```

linuxhotel

`read()` - u. `write()` - Operationen

- in den Kernel übergebene Userspace-Adressen können nicht direkt verwendet werden; die Daten müssen in den Kernel mit speziellen Funktionen kopiert werden
- Funktionen `copy_to_user()` und `copy_from_user()` verifizieren die Userspace-Adresse des aufrufenden Prozesses, lösen die virtuelle Adresse auf und kopieren den Speicher in den Kernel-Adressbereich hinein oder heraus
- dabei findet jedesmal ein Kopieren von Speicher statt
  $\Rightarrow$ kostet Rechenzeit

linuxhotel

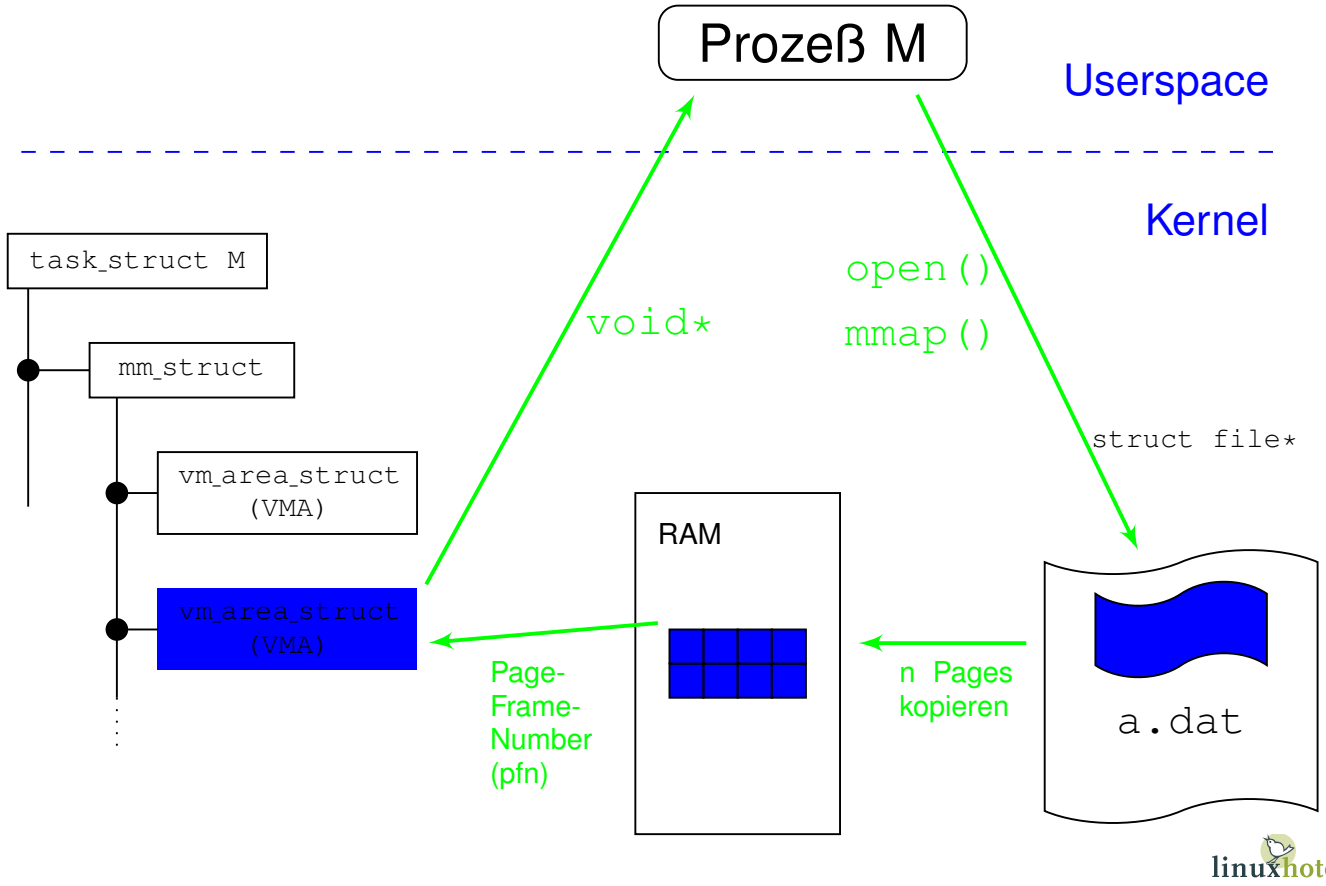
Memory-Mapping I

- beim Memory-Mapping wird der Adressbereich eines Prozesses um eine VMA erweitert bzw. eine bestehende VMA vergrössert
- Speicher-Pages werden in den VMA eingeblendet und ein Zeiger auf den Beginn des Bereiches zurückgegeben
- ein und dieselbe Speicher-Page kann auch mehrfach gemapped werden; die Page existiert dann einmalig, jedoch für jedes Mapping ein VMA
- Userspace-Funktion `mmap()` oder `mmap2()` mapped Dateien (reguläre oder Character-Device)
- beim Mappen von regulären Dateien wird der Dateiinhalt ins RAM kopiert und dort verwendet
  → schnellerer Zugriff bei vielen Schreib-/Lese-Operationen auf der Hardware

Memory-Mapping II

- spezielles Character-Device `/dev/mem` mapped physikalischen Speicher, unter anderem auch ins RAM gemappte Hardware-Register
  → Hardwarezugriff direkt aus dem Userspace heraus
- eigene Character-Devices können auch im Kernel-Treiber allokierten Speicher durch den Userspace mappen lassen
  → schneller Datenaustausch zwischen Userspace und Kernel

Datei Memory-Mappen



mmap() im Character-Device I

```
#define NPAGES 8
#define NPAGES_ORDER 3

static char* kio_mem;

static int __init kio_init (void)
{
  if ((kio_mem = (char*)__get_free_pages
                    (GFP_KERNEL, NPAGES_ORDER)) == NULL)
  {
    return -ENOMEM;
  }

  return 0;
}

void __exit kio_exit(void)
{
  free_pages ((long)kio_mem, NPAGES_ORDER);
}
```

linuxhotel

mmap() im Character-Device II

```
int kio_mmap(struct file *f, struct vm_area_struct *vma)
{
  int ret;
  long length = vma->vm_end - vma->vm_start;

  // Pruefe Laenge
  if (length > NPAGES * PAGE_SIZE)
    return -EIO;

  if ((ret = remap_pfn_range(vma,
           vma->vm_start,
           virt_to_phys((void *)kio_mem) >> PAGE_SHIFT,
           length,
           vma->vm_page_prot)) < 0) {
    return ret;
  }

  return 0;
}


// physikalische Adresse in Page-Frame-Number wandeln:
// virt_to_phys((void *)kio_mem) >> PAGE_SHIFT
```

linuxhotel

mmap() im Character-Device III

```
#define NPAGES 8

int main(int argn, char* argv[])
{
  int fd;
  unsigned int *kadr;
  int len = NPAGES * getpagesize();

  if ((fd = open("/dev/kio", O_RDWR)) < 0)
    return -1;

  kadr = mmap(0, len, PROT_READ|PROT_WRITE,
                                MAP_SHARED, fd, 0);
  if (kadr == MAP_FAILED)
    return -2;

  munmap (kadr, len);
  close(fd);
  return 0;
}
```

linuxhotel

9 Flattened-Device-Tree

linuxhotel

FDT - Flattened Device Tree I

- definiert welche Komponenten — vor allem Treiber — des laufenden Kernels aktiviert werden
  ⇒ *Beschreibung der Hardware*
- enthält keine „erkundbare“ Hardware
  → PCI- und USB-Geräte sind nicht enthalten
- definiert wie angesprochene Komponenten parametisiert werden
  ⇒ *Kernel muß Treiber einkompiliert haben*
- wird beim Booten dem Kernel übergeben
  ⇒ *Bootloader muß Device-Tree-Binary übergeben können* oder
  ⇒ *Device-Tree-Binary wird an Kernel angehängt*
  „Übergangslösung“

linuxhotel

FDT - Flattened Device Tree II

- möglich ist:
  ein Kernel-Kompilat pro (Unter-)Architektur sowie
  ein Device-Tree-Source-File (*.dts) pro Board
  ⇒ Kernel enthält Vereinigungsmenge aller Treiber der verwendeten Boards

- Device-Tree-Sourcen sind modular aufgebaut; prinzipiell:
  `myBoard.dts` inkludiert
  `mySOC.dtsi` inkludiert
  `skeleton.dtsi`

- Verzeichnis in Kernel-Sourcen:
  `arch/<ARCH>/boot/dts`

- Kernel-Konfiguration:
  `CONFIG_USE_OF`
  `CONFIG_ARM_APPENDED_DTB` (angehängtes DTB)

linuxhotel

FDT - Flattened Device Tree III

- Device-Tree-Binary (`*.dtb`) wird mittels Device-Tree-Compiler `dtc` aus Device-Tree-Source `*.dts` generiert

- bei Erstellung des Kernels erfolgt dies automatisch bei `make` oder auch explizit angestoßen durch:
  `make dtbs`

- Rückwandlung von Device-Tree-Binary in Sourcen, wobei Aliases ersetzt sind (siehe `phandle`):

```
dtc -I dtb -O dts myBoard.dtb
```

  ⇒ Was ist aus *.dts-File mit seinen *.dtsi-Includes am Ende geworden?

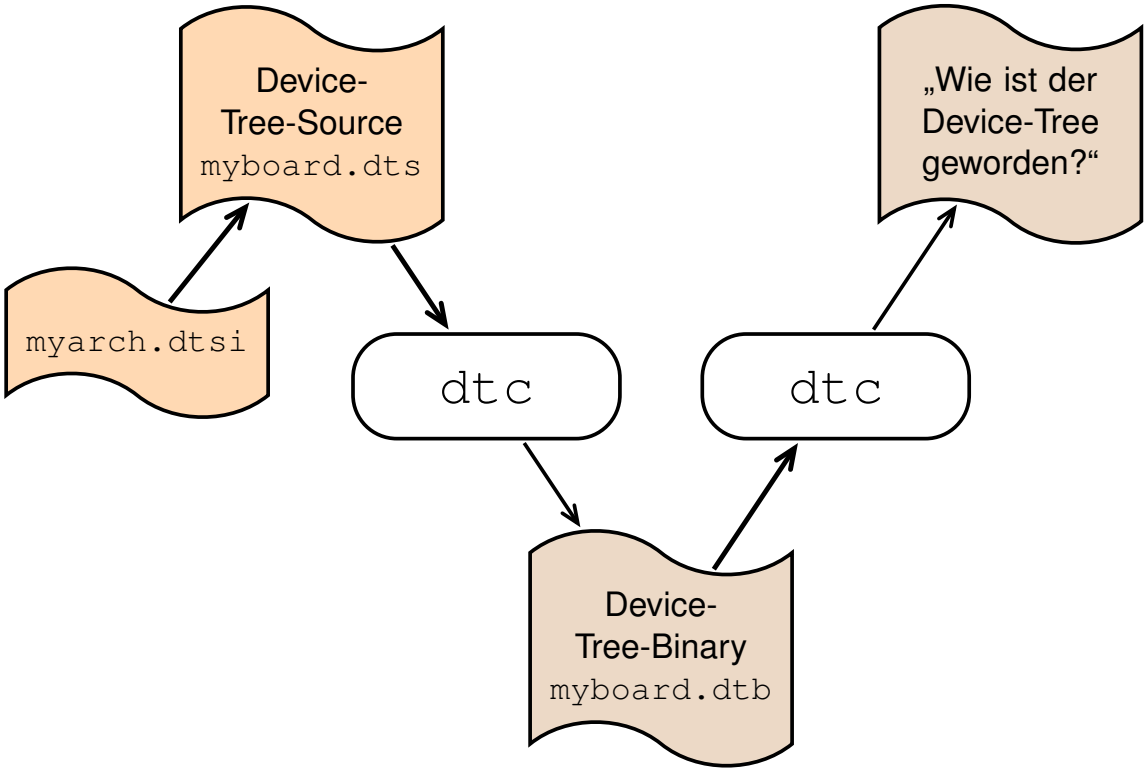
linuxhotel

FDT - Flattened Device Tree IV

- im laufenden Linux-System befindet sich der vom Kernel erkannte Device-Tree als Dateisystemstruktur mit Binärdateien unter `/sys/firmware/devicetree/base`
- daraus Generierung von ASCII-Device-Tree mit `dtc -I fs -O dts /sys/firmware/devicetree/base`

linuxhotel

Device-Tree-Binary generieren



linuxhotel

Device-Tree-Source — Beispiel in Auszügen I

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "globalscale,sheevaplug",
              "marvell,kirkwood-88f6180", "marvell,kirkwood";

    chosen {
        bootargs = "console=ttyS1,115200n8 earlyprintk";
    };
    aliases {
        gpio0 = &gpio0;
        gpio1 = &gpio1;
    };

...
```

linuxhotel

Device-Tree-Source — Beispiel in Auszügen II

```
    ocp@f1000000 {
        compatible = "simple-bus";
...
        gpio1: gpio@10140 {
            compatible = "marvell,orion-gpio";

            #gpio-cells = <2>;
            gpio-controller;
            reg = <0x10140 0x40>;
            ngpios = <18>;

            interrupt-controller;
            #interrupt-cells = <2>;
            interrupts = <39>, <40>, <41>;
            clocks = <&gate_clk 7>;
        };

...
```

linuxhotel

Device-Tree-Source — Beispiel in Auszügen III

```
        pinctl: pinctrl@10000 {
            compatible = "marvell,88f6180-pinctrl";
            reg = <0x10000 0x20>;
            pmx_led_green: pmx-led-green {
                marvell,pins = "mpp42";
                marvell,function = "gpio";
            };
        };
    };
    gpio-leds {
        compatible = "gpio-leds";
        pinctrl-0 = <&pmx_led_green>;
        pinctrl-names = "default";
        gruen {
            label = "sheevaplug:green";
            gpios = <&gpio1 10 0>;
        };
    };    };
```

linuxhotel

FDT — U-Boot I

- *.dts-Datei für Board anpassen / erstellen
- Kernel mit FDT-Support erstellen; Boardsupport mit auswählen
- Root-Filesystem erstellen
- Bootloader-Image-Definition (*.its-Datei) erstellen
  siehe: `u-boot/doc/uImage.FIT/howto.txt`
- Bootloader-Image erstellen mit aktuellem mkimage-Tool
  `mkimage -f myfit.its myfit.itb`
- runterladen und testen

```
u-boot> tftp myfit.itb
u-boot> bootm
```

linuxhotel

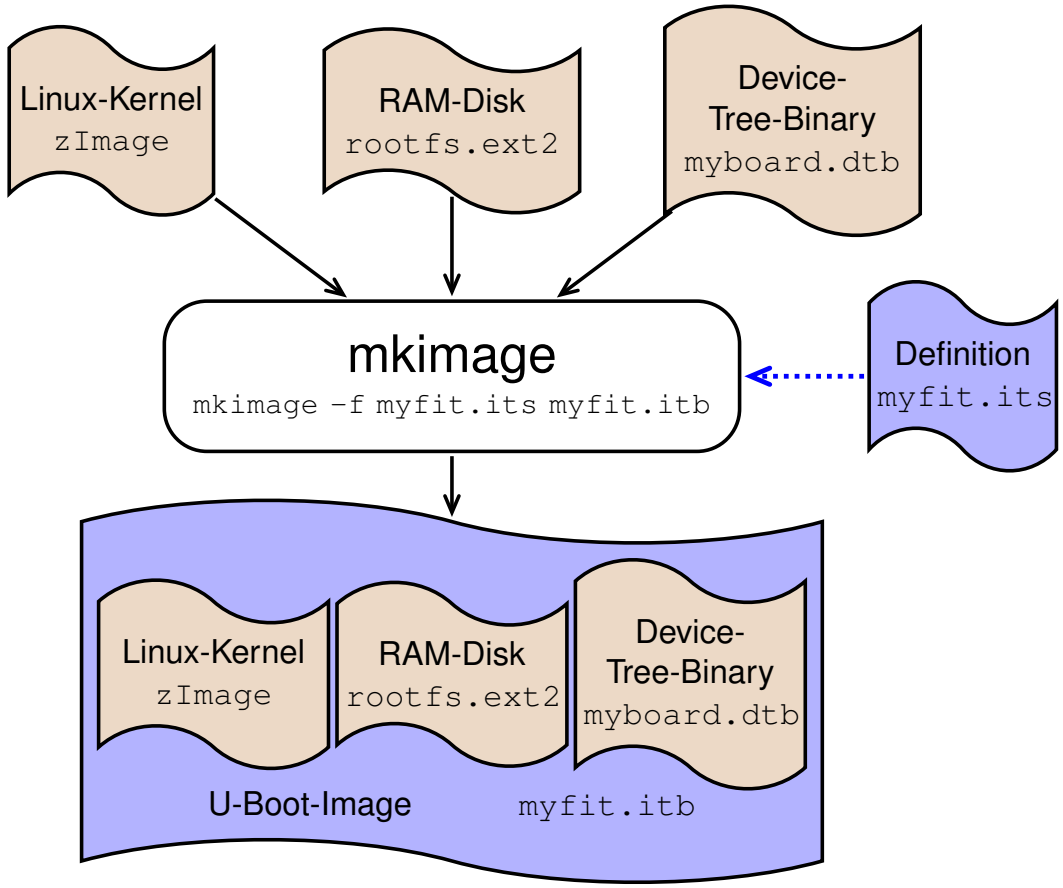
FDT — U-Boot II

- bei „altem“ Bootloader ohne FDT-Support:
  `cat zImage myboard.dtb > zImage_dtb`

  `zImage_dtb` wie „normales“ Kernel-Image verwenden

linuxhotel

U-Boot-Image mit FDT generieren



linuxhotel

Bootloader-Image-Definition — Beispiel I

```
/dts-v1/;
/ {
  description = "kernel, ramdisk und FDT";
  #address-cells = <1>;

  images {

    kernel@1 {
      description = "linux-3.11-wut";
      data = /incbin/("./zImage");
      type = "kernel";
      arch = "arm";
      os = "linux";
      compression = "none";
      load = <0x00008000>;
      entry = <0x00008000>;
    };
```


Bootloader-Image-Definition — Beispiel II

```
    ramdisk@1 {
      description = "buildroot-ramdisk";
      data = /incbin/("./rootfs.ext2");
      type = "ramdisk";
      arch = "arm";
      os = "linux";
      compression = "none";
      load = <00000000>;
      entry = <00000000>;
    };
    fdt@1 {
      description = "sheevaplug-fdt";
      data = /incbin/("./kirkwood-sheevaplug.dtb");
      type = "flat_dt";
      arch = "arm";
      compression = "none";
    };
  };
```

Bootloader-Image-Definition — Beispiel III

```
  configurations {
    default = "config@1";
    config@1 {
      description = "sheevaplug-fdt";
      kernel = "kernel@1";
      ramdisk = "ramdisk@1";
      fdt = "fdt@1";
    };
  };
};
```

linuxhotel

FDT — barebox mit SD-Card I

- *.dtb-Datei wird in `/boot`-Partition kopiert
- interne Device-Tree muß entladen und eigener geladen werden
  → später eigenen Device-Tree als internen einbauen
- „normal“ Booten

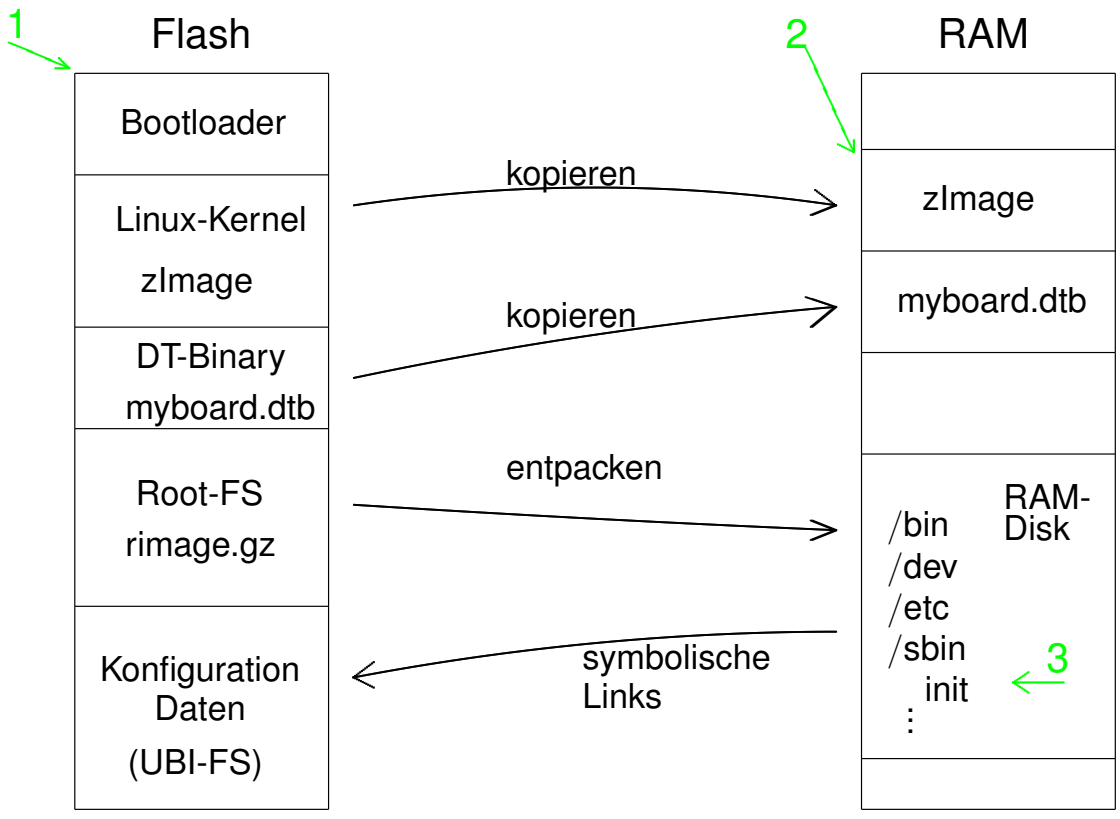
linuxhotel

/env/bin/init-Skript für barebox

```
#!/bin/sh

global linux.bootargs.base="console=ttyS0,115200"
global linux.bootargs.myroot="root=/dev/mmcblk0p2 rw rootwait"

echo -n "Hit any key to stop autoboot: "
timeout -a 3
if [ $? != 0 ]; then
    exit
fi

oftree -f
oftree -l /boot/am335x-wega-rdk.dtb

bootm /boot/uImage
```

linuxhotel

Bootvorgang eines Embedded-Linux mit FDT



linuxhotel

Teil III

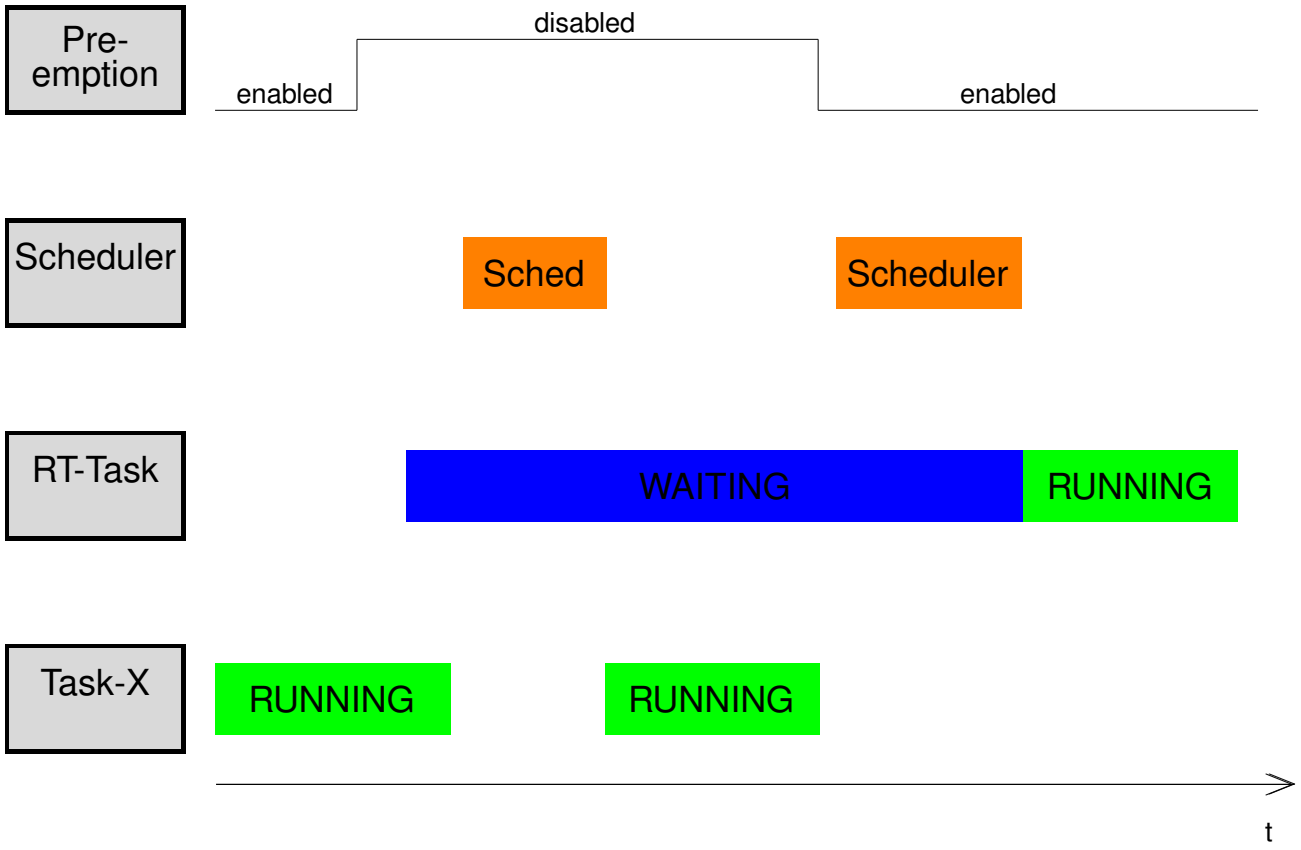
Synchronisation

Synchronisation

10 Synchronisierung - Konzepte

- Preemption-Sperre
- Bottom-Half-Sperre
- Interrupt-Sperre
- Memory-Barrier
- lockdep
- Beispiel: Treiber mit verketteten Listen

Preemption-Sperren



- Preemption-Sperre
  gerade rechnender Task hat Preemption disabled
  → Abschalten des Schedulers auf der lokalen CPU
  ⇒ er kann nicht unterbrochen werden

- auch hochpriorisierte RT- oder Deadline-Tasks bekommen keine Rechenzeit

- Aufsetzen des virtuellen Adressbereiches kritisch bei Swapping
  → Page-Fault und Laden des Speichers von Massenspeicher
  → Angabe der zeitlichen Dauer nicht möglich
  ⇒ bei echtzeitrelevanten Systemen Swapping unterbinden

Preemption-Sperre I

- Prozeß-Kontext kann im Systemraum jederzeit unterbrochen werden (`CONFIG_PREEMPT`)
- Preemption-Sperre bewirkt, daß der gerade laufende Kontext nicht unterbrochen werden kann
- Task ruft bei abgeschalteter Preemption blockierende Funktion auf:
  - Task kommt auf Liste der schlafenden Tasks
  - Scheduler wird aufgerufen
  - Scheduler unternimmt nichts, da Preemption-Sperre gesetzt
    $\Rightarrow$ Deadlock
- keine blockierenden Funktionsaufrufe bei gesetzter Preemption-Sperre
  $\rightarrow$ *atomarer Kontext*

linuxhotel

Preemption-Sperre II

- Einsatz: Absicherung im Prozeß-Kontext vor gegenseitigem Zugriff
- wenn Scheduler abgeschaltet ist, hat laufende Task höchst mögliche Priorität
  $\rightarrow$ entspricht *Priority-Ceiling*
- im Interrupt-Kontext zwecklos
- `CONFIG_DEBUG_ATOMIC_SLEEP` liefert Log-Ausgaben, wenn wartende / blockierende Funktion im atomaren Kontext (z. B. Preemption-Sperre) aufgerufen wird
- `CONFIG_PREEMPT_TRACER` misst jede Zeitspanne, über welche hinweg Preemption abgeschaltet ist und trägt diese als Histogramm auf

linuxhotel

Preemption-Sperre III

- kann direkt verwendet werden mit
  `preempt_disable()`
  `preempt_enable()`
  `preempt_count()`
  `...`
- Preemption-Sperre eingebaut in viele Synchronisierungsmechanismen, wie Spin-Lock, RW-Lock, Seq-Lock, RCU, ...

linuxhotel

Bottom-Half-Sperre I

- Begriff „Bottom-Half“ kommt vom gleichnamigen Mechanismus, welcher aber nicht mehr existiert
- ab Kernel 2.6 übernehmen SoftIRQ's die Rolle der Bottom-Halves als Verlängerung des Hardware-Interruptes in Software
- Einsatz: Datenaustausch zwischen Prozeß- und Interrupt-Kontext
- keine blockierende oder wartende Funktionsaufrufe bei gesetzter Bottom-Half-Sperre
  → *atomarer Kontext*
  → Preemption-Sperre implizit gesetzt
- SoftIRQ's werden mit `local_bh_disable()` und `local_bh_enable()` auf der lokalen CPU gesperrt

linuxhotel

Bottom-Half-Sperre II

- Bottom-Half-Sperre eingebaut in Synchronisierungsmechanismen mit dem Suffix `_bh`, z. B.:
  `spin_lock_bh()`
  `rcu_read_lock_bh()`
  `...`

linuxhotel

Interrupt-Sperre I

- Sperren der Interrupts auf der lokalen CPU
- globale Interrupt-Sperren (alle CPU's) wurden aus dem 2.6-er-Kernel entfernt
- dienen dem Datenaustausch zwischen Prozeß- und Interrupt-Kontext
- keine blockierende / wartende Funktionsaufrufe bei gesetzter Interrupt-Sperre
  → *atomarer Kontext*
  → Preemption-Sperre und Bottom-Half-Sperre implizit gesetzt
- `CONFIG_IRQSOFF_TRACER` misst jede Zeitspanne, über welche hinweg Interrupts abgeschaltet sind und trägt diese als Histogramm auf mit `CONFIG_INTERRUPT_OFF_HIST`

linuxhotel

Interrupt-Sperre II

- Interrupts werden mit `local_irq_disable()` und `local_irq_enable()` auf der lokalen CPU gesperrt
- Vorsicht: Wenn Interrupts in der aufrufenden Funktion bereits gesperrt waren werden diese mit `local_irq_enable()` wieder freigegeben!
- besser: Sichern der Interrupt-Maske und Abschalten der Interrupts am Beginn des kritischen Abschnitts;
  Wiederherstellen der Interrupt-Maske am Ende des kritischen Abschnitts:
  `local_irqsave()`
  `local_irqrestore()`

linuxhotel

Interrupt-Sperre III

- Interrupt-Sperre eingebaut in Synchronisierungsmechanismen mit `_irqsave`-Extension, z. B.:
  `spin_lock_irqsave()`
  `...`

linuxhotel

Testabfragen

- `preempt_count()` liefert Preemption-Tiefe
  `0`: Preemption ist aktiviert
  `> 0`: Preemption-Tiefe (deaktiviert)
- `in_softirq()`: Code im SoftIRQ
- `in_irq()`: HW-Interrupt
- `in_nmi()`: in einem NMI

linuxhotel

Memory-Barrier I

- Annahme: Compilierte Code wird auf einer CPU ohne Nebenläufigkeiten korrekt ausgeführt
- Nebenläufigkeiten können entstehen durch weitere CPU's (SMP-System) oder auch durch Hardware-IO
- Beispiele:
  - Compiler ordnet Code anders an (Optimizer)
  - geänderte Daten sind auf anderer CPU im Cache
  - CPU optimiert Ausführung zur Laufzeit
- *Memory Barrier* verhindert, daß Operationen davor und danach nicht über die Barrier hinweg vertauscht werden
  $\Rightarrow$ Barrier ist rote Linie

linuxhotel

Memory-Barrier II

- Barrier kann sich auf Lese- und / oder Schreiboperationen beziehen
  → *Read-Memory-Barrier*
  → *Write-Memory-Barrier*
  → *General-Memory-Barrier*

linuxhotel

Barrier — einfaches Beispiel I

```
d[0] = 0; d[1] = -1; i = 0;

====================          ====================
CPU-1                         CPU-2
====================          ====================

d[1] = 2;
i = 1;
                              x = i;
                              y = d[x];
-----------------------------------------------------
Ergebnis:
y = 2;
```

linuxhotel

Barrier — einfaches Beispiel II

```
d[0] = 0; d[1] = -1; i = 0;
====================          ====================
CPU-1                         CPU-2
====================          ====================

i = 1;
                                  x = i;
                                  y = d[x];
d[1] = 2;
-----------------------------------------------------
Ergebnis:
y = -1;

mögliche Ursachen:
- Reordering (Compiler, Runtime)
- CPU-Caches
```

linuxhotel

Barrier — einfaches Beispiel III

```
d[0] = 0; d[1] = 1; i = 0;
====================          ====================
CPU-1                         CPU-2
====================          ====================

d[1] = 2;
// write barrier
i = 1;
                                  x = i;
                                  // read barrier
                                  y = d[x];
-----------------------------------------------------
<d[1]> wird vor <i>           <i> wird vor <d[x]>
geschrieben                   gelesen
```

linuxhotel

Memory-Barrier I

- *Write-Barrier* bewirkt, daß Daten vor der Barrier weggeschrieben sind bevor Daten nach der Barrier geschrieben werden
- *Read-Barrier* führt zum Lesen der Daten vor der Barrier bevor Daten nach der Barrier gelesen werden
- Barriers wirken sich immer nur auf die ausführende CPU aus
- Schreiben in der richtigen Reihenfolge heißt nicht, daß auch in der richtigen Reihenfolge gelesen wird!
  $\Rightarrow$ Write- und Read-Barriers paarweise verwenden
- weniger Overhead als klassisches Locking
- komplexer in der Verwendung wenn viele voneinander abhängige Daten gesichert werden sollen

linuxhotel

Memory-Barrier II

- Linux-Locking-Mechanismen haben „One-Way-Barriers“ eingebaut:
  $\rightarrow$ keine Anweisung nach einem Lock wird vor dem Lock ausgeführt
  $\rightarrow$ keine Anweisung vor einem Unlock wird nach dem Unlock ausgeführt
  $\Rightarrow$ Anweisungen vor dem Lock oder nach dem Unlock können in den kritischen Abschnitt „reinwandern“
- `barrier()` ist eine reine Compiler-Barrier
- mit `smp_wmb()`, `smp_rmb()` und `smp_mb()` können Barriers zwischen CPU's gebildet werden
- `mmiowb()` fügt eine Memory-Mapped-IO-Write-Barrier ein
- Siehe auch:
  `Documentation/memory-barriers.txt`

linuxhotel

lockdep — Locking Correctness I

- in Kernel integrierter Mechanismus zur Überprüfung auf richtiges Locking
- *Lock-Klasse*: Gruppe von logisch zusammengehörigen Locking-Objekten
  Beispiel:
  Spin-Lock in einer Struktur ist eine Lock-Klasse auch wenn es viele Instanzen dieser Struktur gibt
- statisch initialisierter Lock (Bsp.: `SPIN_LOCK_UNLOCKED()`) beansprucht jeweils eine Lock-Klasse
  → Initialisierung zur Laufzeit bevorzugen
- Lock-Klasse ist Vereinfachung, um Datenmenge handelbar und Performanceverlust erträglich zu gestalten

linuxhotel

lockdep — Locking Correctness II

- übwacht wird, in welchem Kontext (Hard-IRQ, SoftIRQ, Prozeß) eine Lock-Klasse verwendet wird und ob diese Lock-Klasse immer mit entsprechendem Schutz verwendet wird

Example

Spin-Lock wird einmalig im Hard-IRQ verwendet und ein andermal im Prozeß-Kontext ohne abgeschaltete Interrupts
beide Fälle werden aufgezeichnet und davon ausgegangen, daß sich beide Abschnitte auch überlappen können
⇒ potentieller Deadlock

linuxhotel

lockdep — Locking Correctness III

- Berücksichtigung von Read-Only- und Read/Write-Abschnitten mit ihrer Semantik

Example

Read-Lock im Hard-IRQ erfordert keine Interrupt-Sperre im Prozeß-Kontext-Read-Lock
jedoch: Write-Lock im Prozeß-Kontext muß Interrupt-Sperre haben

- Prüfung auf immer gleiche Reihenfolge bei der Verwendung mehrerer Locks
  ⇒ Deadlock-Szenario möglich
- Ausgaben im Systemlog beachten:
  `dmesg` oder
  `tail -f /var/log/messages`

linuxhotel

lockdep — Locking Correctness IV

Lock-Statistik in `/proc/lock_stat`

- Lock-Contention (Wartezuständen am Lock)
- „Auf wen wird gewartet?“(Top-4 Blockierer)
- „Wer wartet auf mich?“(Top-4 Blockierte)
- Wartezeit (min/max) in [μs] und Anzahl der Wartefälle
- Haltezeit (min/max) in [μs] und wie oft Lock gehalten wurde

linuxhotel

lockdep — Locking Correctness V

Tracing-Events im Debug-FS

- *lock:lock_acquire*: Anforderung des Locks
- *lock:lock_acquired*: Locks erhalten
- *lock:lock_contended*: Tasks blockiert an Lock-Operation
- *lock:lock_release*: Lock freigegeben

→ Tracing der Lock-Aktivität zusammen mit Scheduling- und Interrupt-Events liefert detailliertes Bild der Abläufe

linuxhotel

lockdep — Locking Correctness VI

Kernel-Konfiguration

- `CONFIG_LOCKDEP` schaltet lockdep-Mechanismus ein
- `CONFIG_PROVE_LOCKING` prüft semantische Korrektheit des Locking
- `CONFIG_DEBUG_LOCK_ALLOC`: prüft (Re-)Inititialisierung, Freigabe, Task-Exit mit gehaltenem Lock
- `CONFIG_LOCK_STAT`: liefert Lock-Statistik und Contended-/Acquired-Tracing

linuxhotel

lockdep — Fehlerausgabe im Systemlog

```
tail -f /var/log/messages
```

```
 ================================
 [ INFO: inconsistent lock state ]
 3.6.1 #1 Tainted: G           O
 --------------------------------
 inconsistent {HARDIRQ-ON-W} -> {IN-HARDIRQ-W} usage.
 lese.sh/28218 [HC1[1]:SC0[0]:HE0:SE1] takes:
  (&(&rcul_list_lock)->rlock){?.+...},
          at: [<bf010080>] rcul_add_messwert+0x28/0xa8 [list_rcu]
 {HARDIRQ-ON-W} state was registered at:
   [<c006c620>] mark_lock+0x140/0x630
 other info that might help us debug this:
  Possible unsafe locking scenario:

        CPU0
        ----
   lock(&(&rcul_list_lock)->rlock);
   <Interrupt>
     lock(&(&rcul_list_lock)->rlock);

  *** DEADLOCK ***

 1 lock held by lese.sh/28218:
  #0:  (&mm->mmap_sem){++++++}, at: [<c00150d0>] do_page_fault+0x88/0x3cc
```

linuxhotel

lockdep — Lockstatistik-Beispiel

```
cat /proc/lock_stat > /tmp/lock_stat.txt
cat /tmp/lock_stat.txt | cut -c -125
```

```
lock_stat version 0.3
-----------------------------------------------------------------------------------------------
       class name    con-bounces    contentions   waittime-min   waittime-max waittime-total
-----------------------------------------------------------------------------------------------

  &rcul_list_lock:             0             76         115.90       22672.65      336591.26
  ---------------
  &rcul_list_lock             67          [<bf004144>] messwert_show+0x28/0xf4 [list_mutex]
  &rcul_list_lock              9          [<bf004298>] delete_store+0x38/0xd4 [list_mutex]
  ---------------
  &rcul_list_lock             76          [<bf004144>] messwert_show+0x28/0xf4 [list_mutex]

.......................................................................................

       slab_mutex:             0              0           0.00           0.00           0.00
  rcu_read_lock-R:             0              0           0.00           0.00           0.00
         sb_lock:              0              0           0.00           0.00           0.00

usw.
```

linuxhotel

lockdep — Lockstatistik-Beispiel

```
cat /proc/lock_stat > /tmp/lock_stat.txt
cat /tmp/lock_stat.txt | cut -c -40,140-
```

```
lock_stat version 0.3
-------------------------------------------------------------------------------
       class name    acquisitions   holdtime-min   holdtime-max holdtime-total
-------------------------------------------------------------------------------

  &rcul_list_lock           6393           6.86       28159.71     2194666.33
  ---------------
  &rcul_list_lock
  &rcul_list_lock
  ---------------
  &rcul_list_lock

.....................................................................

       slab_mutex             61         152.53         240.55       11262.85
         ntp_lock          36513           1.57           3.64       70477.34
           l3_key             92           1.72          31.12        1267.87
   tty_ldisc_lock           4117           1.69           4.23        9179.72
      logbuf_lock              2           1.73          13.70          15.43
       xtime_lock          12202           7.89          35.17      329086.58

usw.
```

linuxhotel

lockdep — Ausgaben im ftrace

```
mount -t debugfs nodev /debug
cd /debug/tracing

echo lock:* > set_event
```

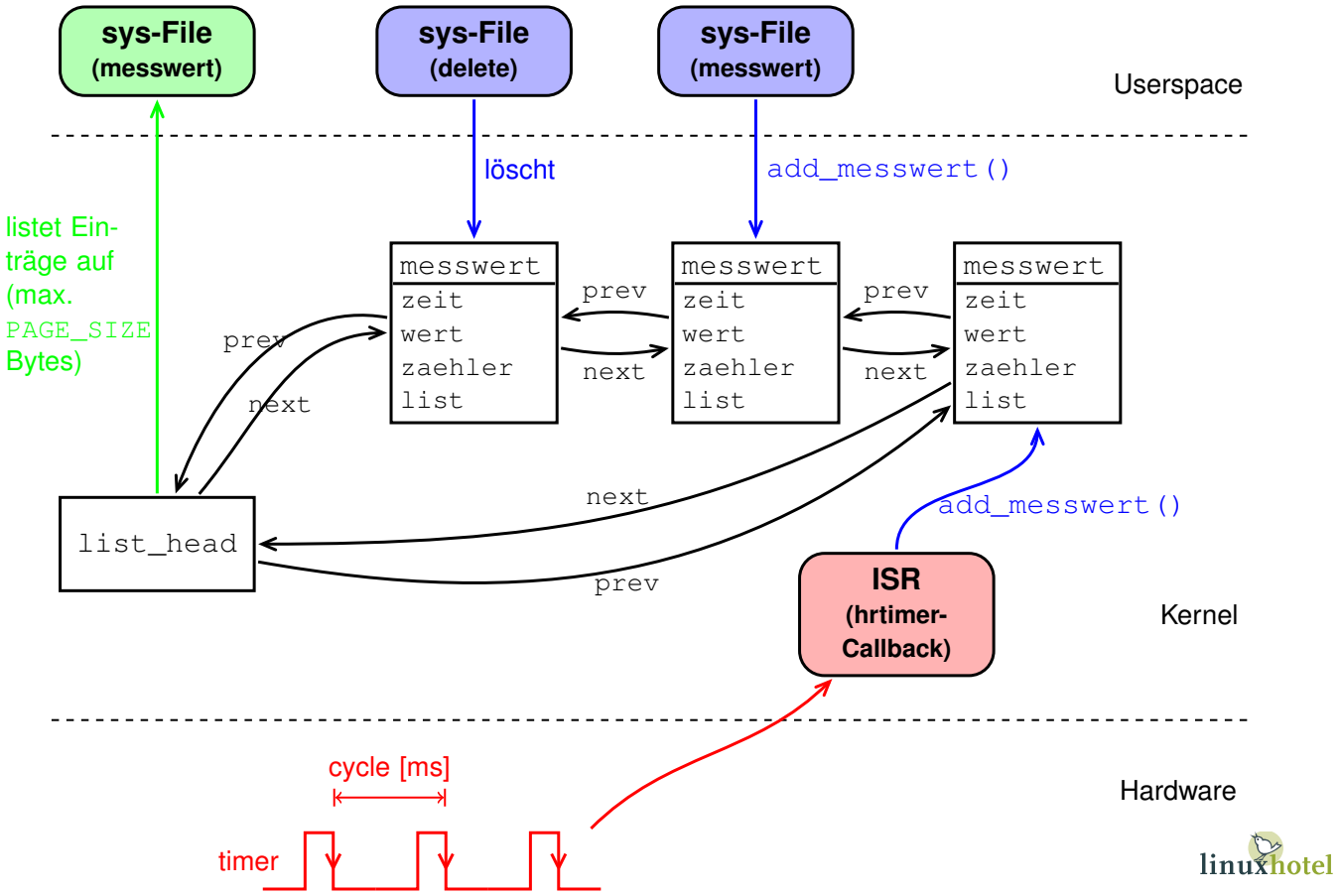
Modul laden und testen

```
cat trace | grep rcul > /tmp/t1.log
less /tmp/t1.log
```

```
cat-31356 [000] d..2  2195.588214: lock_acquire: bf0048b8 &(&rcul_list_lock)->rlock
cat-31356 [000] d..2  2195.588219: lock_acquired: bf0048b8 &(&rcul_list_lock)->rlock
cat-31356 [000] d..2  2195.588241: lock_release: bf0048b8 &(&rcul_list_lock)->rlock
cat-31357 [000] d..2  2195.627073: lock_acquire: bf0048b8 &(&rcul_list_lock)->rlock
cat-31357 [000] d..2  2195.627078: lock_acquired: bf0048b8 &(&rcul_list_lock)->rlock
cat-31357 [000] d..2  2195.627101: lock_release: bf0048b8 &(&rcul_list_lock)->rlock
```

linuxhotel

Beispiel: Listen-Synchronisierung



- Kernel-Treiber enthält chronologische Liste der letzten gemessenen Meßwerte
- Interrupt benachrichtigt über neuen Meßwert und trägt diesen in die Liste ein (Funktion: `add_messwert()`)
- Zykluszeit des periodischen hrtimer-Interrupts kann mit der Datei `cycle_ms` in [*ms*] eingestellt werden (nicht abgebildet)
  Bsp.: Zykluszeit auf 10 ms einstellen
  `echo 10 > /sys/class/rcul/rcul1/cycle`
- Treiber-Interface zum Userspace komplett im sysfs abgebildet (wegen Übersichtlichkeit)
- sysfs-Datei `delete` (im Pfad „`/sys/class/rcul/rcul1`“) dient zum Löschen der $< n >$ ältesten Einträge, wobei $< n >$ der in die Datei geschriebene Zahlenwert bedeutet
  Bsp.: Löschen der drei ältesten Einträge
  `echo 3 > /sys/class/rcul/rcul1/delete`
- Lesen der Datei `messwert` fragt Meßwerte tabelarisch ab; es werden maximal `PAGE_SIZE` Bytes zurückgegeben
  Bsp.: Messwerte abfragen
  `cat /sys/class/rcul/rcul1/messwert`
- Schreiben in die Datei `messwert` fügt neuen Meßwert ein
  Bsp.: Messwert 127 eintragen
  `echo 127 > /sys/class/rcul/rcul1/messwert`

Checkliste - Synchronisierung I

- Synchronisierung beginnt im Design des Treibers
- Anwendungsfälle abstrahieren
  → Welche Akteure gibt es (Interrupt, Prozeß, Kernel-Thread, ...)?
  → Wer kann wen unterbrechen?
- Was sind die gleichzeitig genutzten Daten?
- Welche Datenänderungen erfolgen atomar?
- In welchem Kontext erfolgen die Zugriffe (Prozeß- oder Interrupt-Kontext)?
- Wie viele gleichzeitige Schreiber und Leser gibt es?
- Werden Daten häufig gelesen oder geschrieben?

linuxhotel

Checkliste - Synchronisierung II

- Echtzeit-Prioritäten der Akteure?
- können Akteure entkoppelt werden (FIFO, Ringspeicher)?
- Design optimieren:
  → Anwendungsfälle vereinfachen
  → zu schützende Datenbereiche reduzieren

linuxhotel

11 blockierende Synchronisation
- Wait-Queue
- Semaphore
- Mutex
- Prioritätsinversion
- RT Mutex
- Completion

blockierende Synchronisation

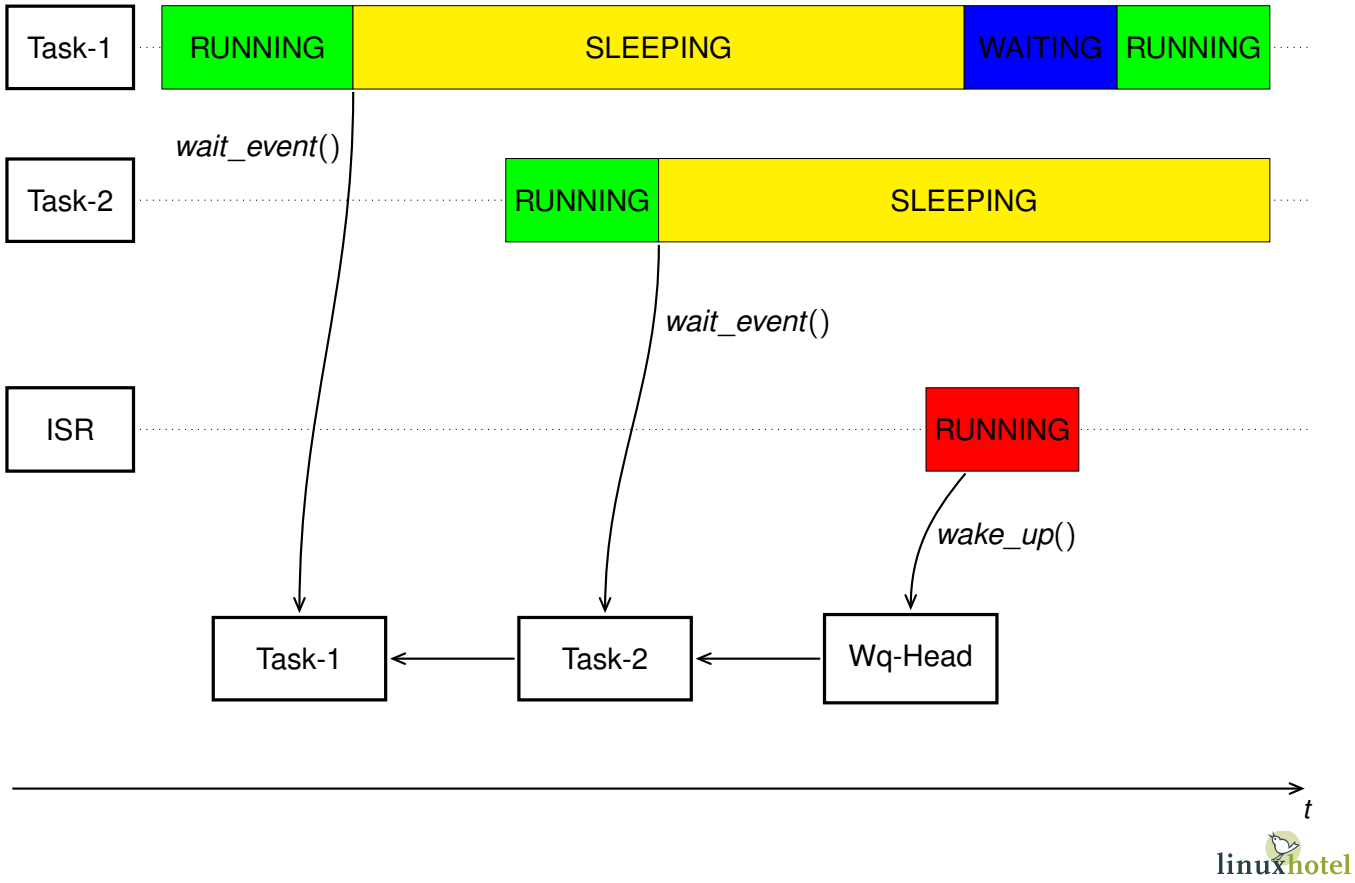
- Tasks werden blockiert indem ihnen Rechenzeit entzogen wird
- sie werden auf eine Warteschlange aufgereiht
- Warteschlange dient dem selektiven Aufwecken (zuweisen von Rechenzeit)
- ausschließlich für den Prozeß-Kontext geeignet
  ⇒ Userspace-Prozeß, -Thread
  ⇒ Kernel-Thread

Warteschlangen

- Task legt sich auf einer Warteschlange (Wait-Queue) schlafen
- Warteschlange für (mehrere) Tasks wird durch Objekt `wait_queue_head_t` dargestellt
- der Task wird in den Status sleeping versetzt
- Re-Scheduling läßt andere Tasks rechnen
- dazu dient die Funktionsgruppe `wait_event()`
- Aufwecken aller Tasks die auf einer Warteschlange warten mit `wake_up()`

linuxhotel

Waitqueue — einen Task wecken

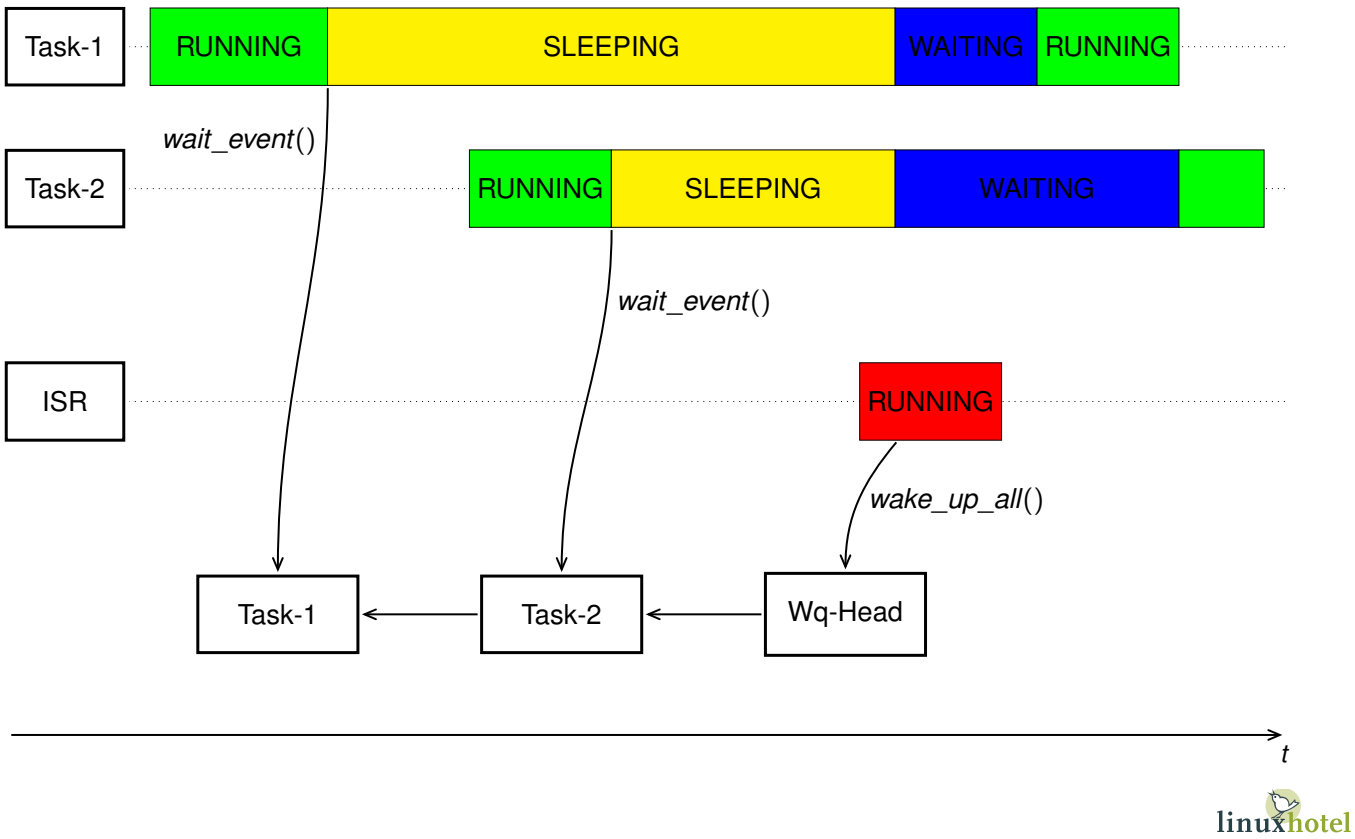


- Task-1 wartet auf einen Event durch Aufruf von `wait_event()`
- Task-1 wird in die Warteschlange (Wq-Head) eingehängt und Rechenzeit entzogen (Sleeping)
- Task-2 wartet ebenfalls auf den gleichen Event und wird genauso in die Warteschlange eingehängt
- Im Beispiel weckt die Interrupt-Service-Routine die Warteschlange Wq-Head mit `wake_up()` auf

  → ein beliebiger Task wird rechenbereit (Waiting) und bekommt später eine Zeitscheibe zugewiesen (Running)

  es ist nicht garantiert, daß der aufgeweckte Task sofort nach der ISR rechnet; wann er rechnet hängt von seiner Scheduling-Policy und -Priorität ab

Waitqueue — alle Tasks wecken



linuxhotel

- ISR weckt alle Tasks der Warteschlange mit `wake_up_all()` auf
- alle Tasks der Warteschlange Wq-Head werden rechenbereit (Waiting) und später auch Rechenzeit zugewiesen (Running)

Beispiel: Warteschlangen I

```
static DECLARE_WAIT_QUEUE_HEAD (kio_read_wq);

static int kio_wq_flag = 0;
static int kio_data_read = 0;

irqreturn_t kio_interrupt
               (int irq, void* dev_id)
{
  // Daten von Hardware empfangen ...
  kio_data_read = 1;

  kio_wq_flag = 1;
  wake_up_interruptible (&kio_read_wq);

  return IRQ_HANDLED;
}
```

linuxhotel

Beispiel: Warteschlangen II

```
ssize_t kio_read (struct file* f,
       char* buf, int cnt, loff_t *off)
{
  int ret;
  while (!kio_data_read)
  {
    if ((ret = wait_event_interruptible
          (kio_read_wq, kio_wq_flag != 0)) < 0)
      return -EINTR;
    kio_wq_flag = 0;
  }

  // Daten in buf kopieren ...
  ret = kio_data_read;
  kio_data_read = 0;
  return ret;
}
```

linuxhotel

Beispiel: Warteschlangen III

```
static unsigned int kio_poll (struct file *f,
                              poll_table* wait)
{
  unsigned int mask = 0;

  poll_wait (f, &kio_read_wq, wait);

  if (kio_data_read)
    mask |= POLLIN;

  return mask;
}
```

linuxhotel

blockierende Kernel-Funktionen I

- blockierende Kernel-Funktionen mit dem Suffix `_interruptible()` werden abgebrochen wenn der aufrufende Prozeß ein (beliebiges) Signal zugestellt bekommt
  Rückgabewert: `-ERESTARTSYS`, `-EINTR`

- entsprechende Varianten ohne das Suffix werden im Falle eines Signales nicht abgebrochen
  → ununterbrechbares Warten
  ⇒ Task kann im blockierenden Zustand nicht terminiert werden

- `_killable()` bedeutet, daß Funktion nur bei „killenden“ Signalen abbricht (z. B. SIGKILL)

linuxhotel

blockierende Kernel-Funktionen II

- `_trylock()` versucht den Lock zu erhalten; wenn es gelingt kann der kritische Abschnitt betreten werden
  Rückgabewert 1: es wurde erfolgreich gelockt
  Ausnahme: Semaphore liefert 0 im Erfolgsfall

- `_timeout()` ermöglicht die Angabe einer maximalen Wartezeit (Timeout) in *jiffies*
  (periodische Clock-Ticks mit Frequenz 100 - 1000 Hz)
  Rückgabewert `-ETIME`: Timeout

Semaphore I

- Implementierung nach `<linux/semaphore.h>`
- Semaphoren besitzen internen Zähler:
  Zähler > 0: Semaphore frei
  Zähler = 0: Semaphore gelockt, potentiell existieren wartende Tasks
- meist wird binäre Semaphore (Zahlenwert 0 oder 1) als Locking-Mechanismus für kritischen Abschnitt verwendet:
  beim Betreten wird Semaphore gelockt mit
  `down(), down_trylock()`
  beim Verlassen wird Semaphore freigegeben mit
  `up()`
- ist die Semaphore gelockt, werden weitere Tasks, welche die Semaphore locken möchten blockiert

linuxhotel

Semaphore II

- blockieren bedeutet, daß wartender Task auf einer Warteschlange aufgereiht und schlafen gelegt wird
- gibt der lockende Task die Semaphore wieder frei, weckt dieser einen der wartenden Tasks von der Warteschlange auf
- es können viele Tasks gleichzeitig auf der gleichen Semaphore warten
- Verwaltungsaufwand für Semaphoren ist relativ hoch
  $\rightarrow$ für lange Haltezeiten gut geeignet
- im Interrupt-Kontext kann Semaphore mit `up()` freigegeben oder auch das Locken mit `down_trylock()` versucht werden

linuxhotel

Semaphore III

- wenn der Interrupt-Kontext garantiert weiß, daß die Semaphore frei ist, kann diese sogar mit der eigentlich blockierenden Funktion `down()` gelockt werden
- Sperren und Freigeben kann in unterschiedlichen Ausführungspfaden erfolgen

Example

Prozeß-Kontext wartet bis Interrupt kommt
`down()` im Prozeß-Kontext und
`up()` in der ISR

- Rückgabewert von `down_trylock()` umgekehrt wie bei RW-Semaphore, Spin-Lock oder Mutex:
  0 : Semaphore gelockt
  1 : Semaphore konnte nicht gelockt werden

linuxhotel

Semaphore IV

- spartanische Unterstützung der lockdep-Mechanismen
  $\Rightarrow$ begrenzte Debugging- und Tracing-Möglichkeiten

linuxhotel

Read-Write-Semaphore I

- Implementierung nach `<linux/rwsem.h>`
- Read-Write-Semaphoren (RW-Semaphore) erlauben die Unterscheidung nach Lesern und Schreibern
- beliebig viele Leser können gleichzeitig Daten lesen; Schreiber werden blockiert
- Schreiber kann kritischen Abschnitt erst betreten, wenn vorhandene Leser den kritischen Abschnitt verlassen haben
- nur ein Schreiber kann gleichzeitig den kritischen Abschnitt betreten
- Schreiber kann „Downgrade“ auf Leser durchführen: `downgrade_write()`

linuxhotel

Read-Write-Semaphore II

- wenn ein Schreiber auf das Locken der RW-Semaphore wartet, werden zusätzliche Leser blockiert:
  ⇒ Schreiber werden bevorzugt behandelt
  ⇒ Schreiber können Leser „aussperren“ wenn häufig Daten geändert werden
- keine unterbrechbaren Varianten (`_interruptible()`) vorhanden

linuxhotel

Mutex I

- Implementierung nach `<linux/mutex.h>`
- sehr effiziente Implementierung basierend auf atomaren Compare-Exchange-Operationen
  → Fastpath- und Slowpath-Unterscheidung
- mit `CONFIG_DEBUG_MUTEXES` werden ausführliche Informationen zu Regelverstößen sowie Deadlocks gelockt

linuxhotel

Mutex II

strenge Regelprüfung

- Owner (der Lockende) muß Mutex wieder freigeben
- kein rekursives Locking erlaubt
- kein mehrfaches Freigeben
- Initialisierungsfunktionen müssen verwendet werden; keine Re-Initialisierung bei gelocktem Mutex
- Task-Exit mit gelocktem Mutex
- keine Verwendung im Interrupt-Kontext

linuxhotel

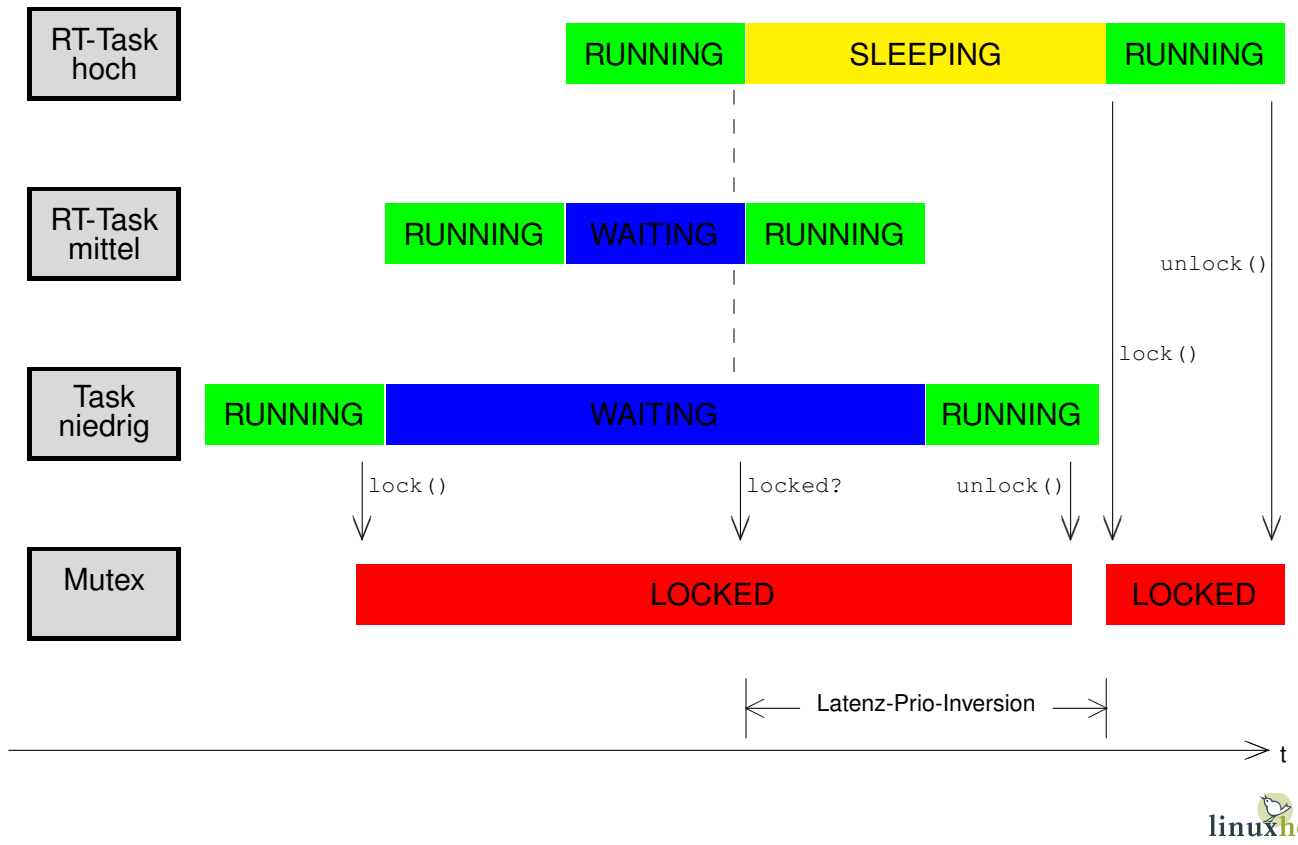
Mutex III

siehe auch

`Documentation/mutex-design.txt`
`<linux/mutex.h>`

linuxhotel

ungebundene Prioritätsinversion

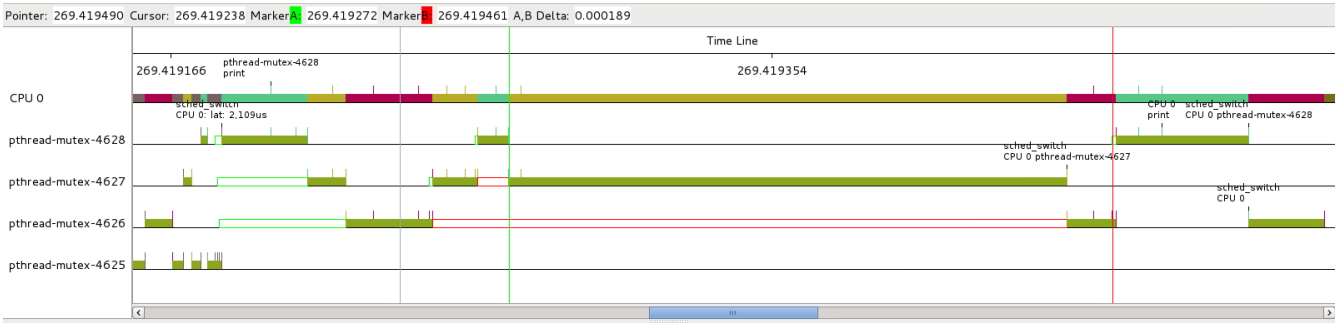


linuxhotel

1. Task mit niedriger Priorität bekommt Rechenzeit und lockt einen Mutex
2. Task mittlerer Priorität unterbricht den niedrigen; Mutex verbleibt im gelockten Zustand
3. Task hoher Priorität unterbricht mittleren und versucht den Mutex zu locken. Dies ist aber nicht möglich und so wird ihm die Rechenzeit wieder entzogen.
4. mittlerer Task darf weiterlaufen
5. Task niedrigerer Priorität läuft, gibt den Mutex wieder frei und die Rechenzeit zurück
6. Erst jetzt mit erheblicher zeitlicher Latenz kommt der Task mit der eigentlich hohen Priorität zum laufen. Seine Ausführungspriorität entspricht faktisch derjenigen eines Tasks mit niedriger Priorität. Man spricht hier von einer ungebundenen Prioritätsinversion.

   Es kann auch sein, daß es nicht nur einen sondern mehrere Tasks mit mittlerer Priorität gibt, diese Rechenzeit benötigen und damit die Latenzzeit nicht mehr vorhersehbar ist.

Prioritätsinversion (kernelshark)



Page 1 Search: Column: # contains graph follows

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	269.413072	pthread-mutex	4625	...1	print	tracing_mark_write: Priority-None
1	0	269.419197	pthread-mutex	4628	...1	print	tracing_mark_write: 171740 90 BARRIER-B -
2	0	269.419216	pthread-mutex	4627	...1	print	tracing_mark_write: 198350 75 BARRIER-A -
3	0	269.419229	pthread-mutex	4626	...1	print	tracing_mark_write: 211689 60 MUTEX-LOCK -
4	0	269.419238	pthread-mutex	4626	...1	print	tracing_mark_write: 223004 60 MUTEX-LOCKED 11314
5	0	269.419243	pthread-mutex	4626	...1	print	tracing_mark_write: 229219 60 BARRIER-A -
6	0	269.419252	pthread-mutex	4627	...1	print	tracing_mark_write: 237670 75 START-OHNE-SYNC -
7	0	269.419258	pthread-mutex	4627	...1	print	tracing_mark_write: 243886 75 BARRIER-B -
8	0	269.419268	pthread-mutex	4628	...1	print	tracing_mark_write: 253105 90 MUTEX-LOCK -
9	0	269.419276	pthread-mutex	4627	...1	print	tracing_mark_write: 261207 75 ENDE-OHNE-SYNC -
10	0	269.419455	pthread-mutex	4626	...1	print	tracing_mark_write: 439023 60 MUTEX-UNLOCK 215950
11	0	269.419469	pthread-mutex	4628	...1	print	tracing_mark_write: 453200 90 MUTEX-LOCKED 200095
12	0	269.419476	pthread-mutex	4628	...1	print	tracing_mark_write: 462000 90 MUTEX-UNLOCK 8870
13	0	269.419613	pthread-mutex	4625	...1	print	tracing_mark_write: 594420 0 ENDE-MAIN -

el

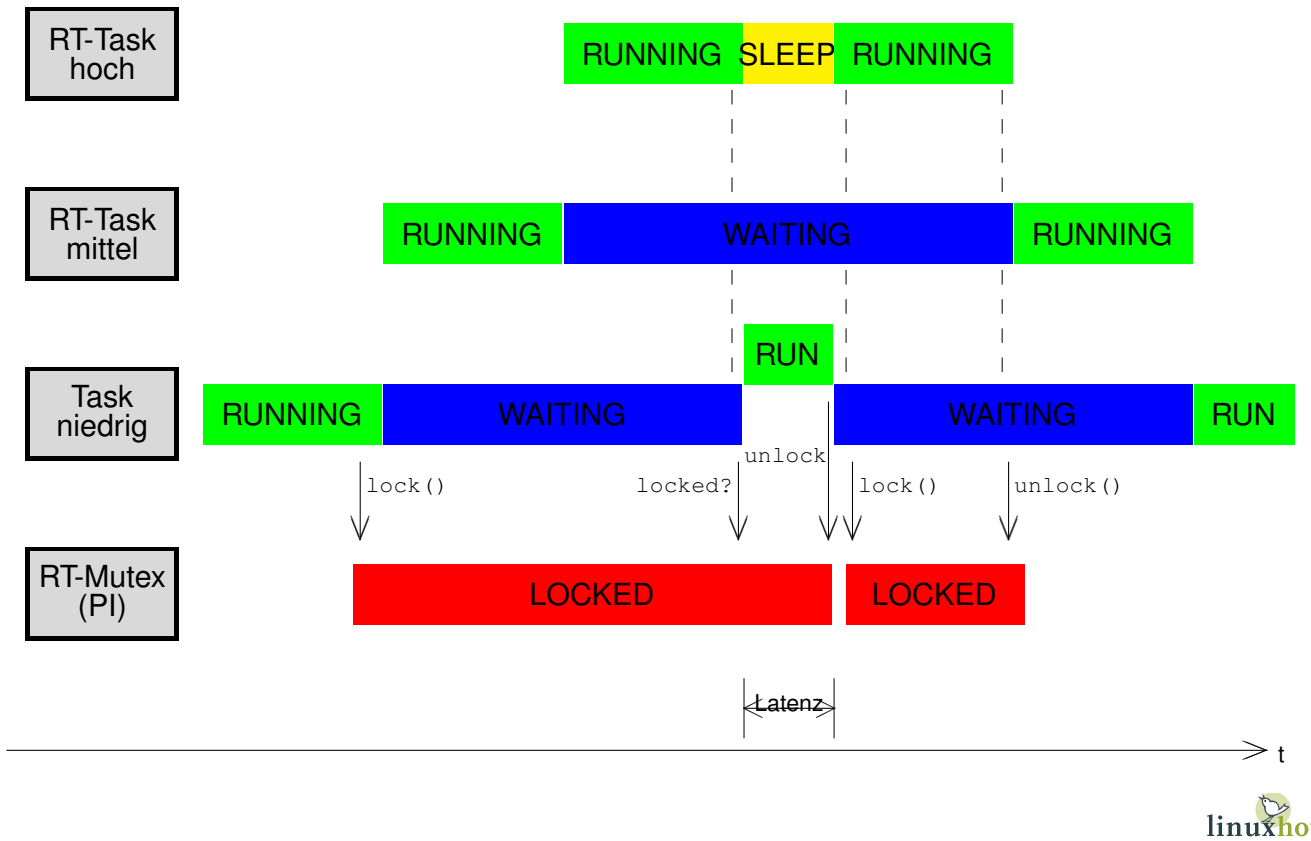
Szenario: `./pthread-mutex -n`

- Hauptprogramm (`main()`-Funktion) startet drei RT-Threads unterschiedlicher Priorität (Task-90: `SCHED_FIFO/90`; Task-75: `SCHED_FIFO/75`; Task-60: `SCHED_FIFO/60`)
- Start der drei RT-Tasks wird mittels einer Barrier synchronisiert, so daß aufgrund ihrer jeweiligen Prioritäten zuerst die Task-90, dann Task-75 und zuletzt Task-60 loslaufen
- mithilfe zweier weiterer Barriers (`BARRIER-A` und `BARRIER-B`) werden dann die RT-Tasks in umgekehrter Prioritätenreihenfolge gestartet
- zuerst läuft Task-60 und lockt dabei einen Mutex
- nachdem Mutex gelockt wurde wird Task-75 ablaufbereit; dieser Task verwendet keinen Mutex
- nachdem dieser Task losgelaufen ist, wird Task-90 ablaufbereit; dieser Task versucht den gleichen Mutex wie Task-60 zu locken
- im Beispiel: PThread-Mutex ohne zusätzliche Attribute

Ergebnis:

- Task mit der höchsten Priorität wird ausgescheduled
- er muß bis zur Beendigung der Task-75 sowie bis zur Freigabe des Mutex durch Task-60 warten
- Latenzzeit beträgt im Beispiel 189μs !

Prioritätsvererbung



linuxhotel

1 Task mit hoher Priorität versucht nun den Mutex zu locken.

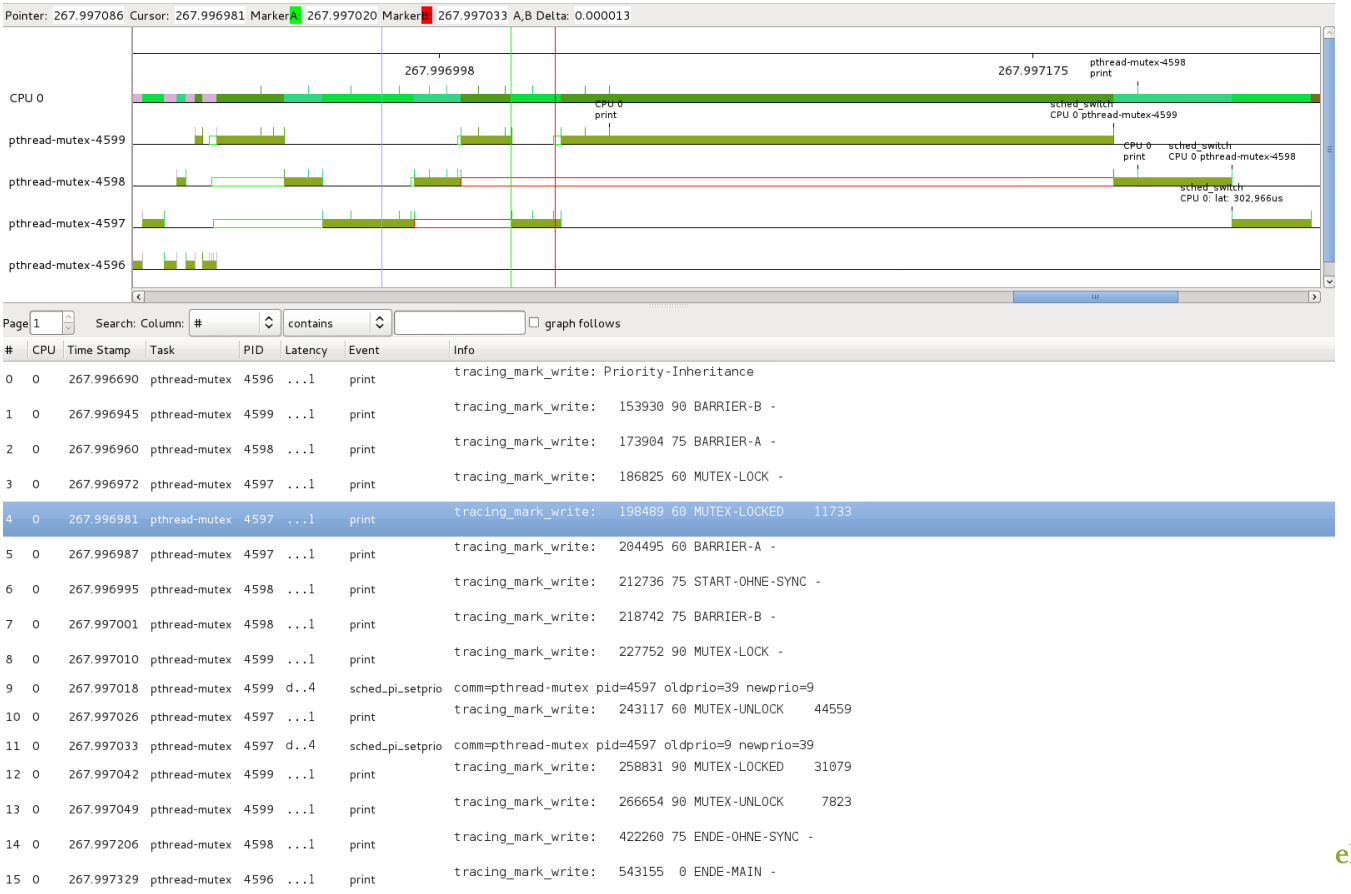
2 RT-Mutex erkennt, daß der Mutex von einer Task mit niedrigerer Priorität gelockt ist. Der lockende Task wird nun in seiner Priorität soweit erhöht, daß er das Niveau des hohen Tasks erreicht. Dadurch erhält er nun Rechenzeit und kann seine Arbeiten zügig weitermachen.

3 niedrig priore Task erbt die Priorität so lange, bis er den Mutex wieder freigibt. Anschließend fällt er wieder auf seine ursprüngliche Priorität zurück.

4 Task mit hoher Priorität kann nun den Mutex locken und rechnen. Die dabei auftretende Latenzzeit konnte dadurch erheblich reduziert werden.

Vorraussetzung für diesen Mechanismus ist, daß der RT-Mutex einen Besitzer hat, es also ein Attribut mit der Kennung des gerade lockenden Tasks gibt, sowie eine Liste mit den auf die Freigabe des Mutex wartenden Tasks.

Priority-Inheritance (kernelshark)



Szenario: `./pthread-mutex -i`

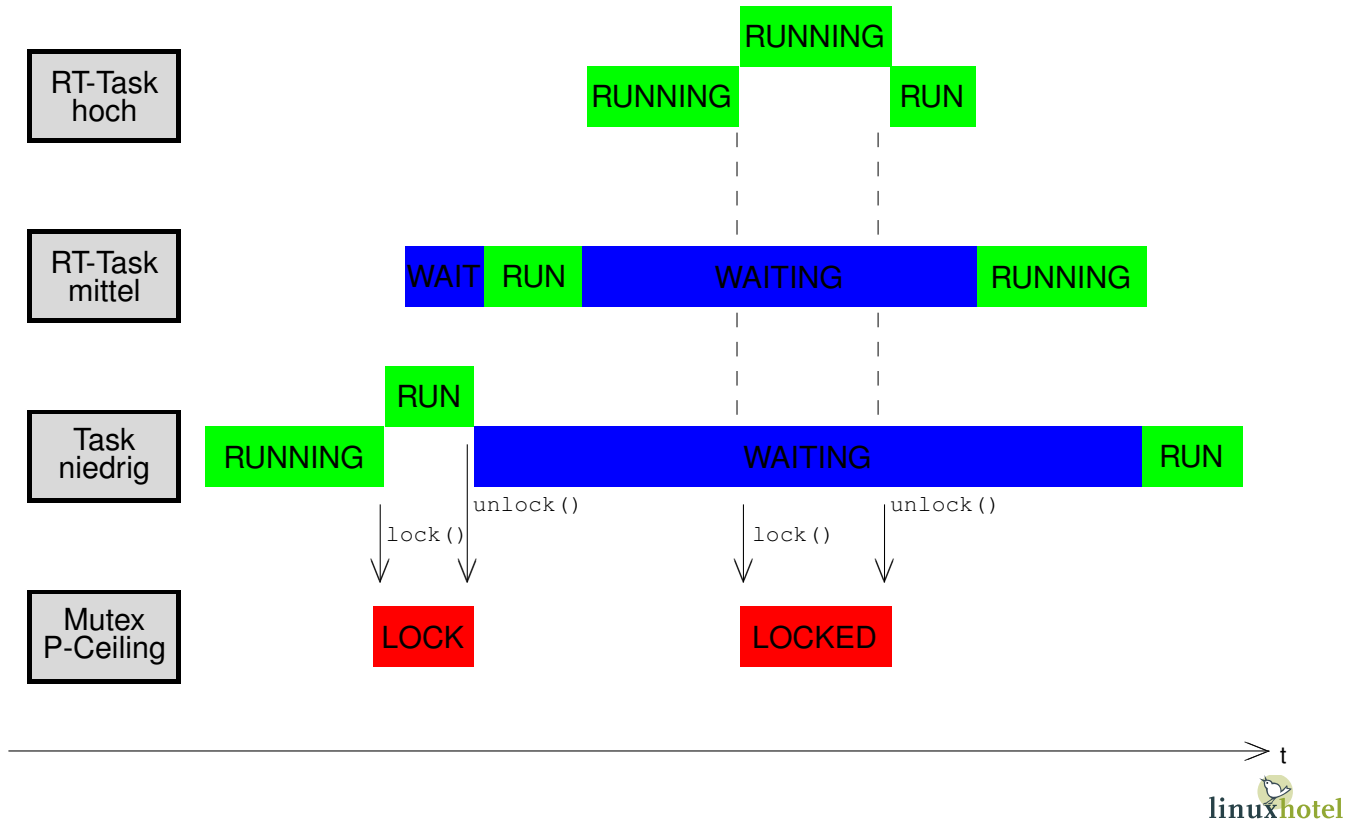
- siehe oben
- im Beispiel: PThread-Mutex mit Priority-Inheritance

Ergebnis:

- Task-60 ist der Owner des Mutex
- wenn Task-90 versucht, den Mutex zu locken, erbt der Owner die Priorität des höchstpriorisierten Wartenden (Prio 90)
- dadurch wird zwar Task-90 ausgescheduled, jedoch läuft unmittelbar danach Task-60 mit der geerbten Priorität von 60 los
- wenn Mutex wieder freigegeben wurde, fällt Task-60 wieder auf seine ursprüngliche Priorität 60 zurück
- Task-90 wird wieder ausgeweckt und kann nun rechnen
- entstandene Latenzzeit beträgt im Beispiel 13μs

  $\Rightarrow$ Locking-Konflikte werden aufgelöst

Priority-Ceiling



linuxhotel

1. Task mit niedriger Priorität lockt den Mutex und wird bereits beim Locken auf die Ceiling-Priorität gesetzt

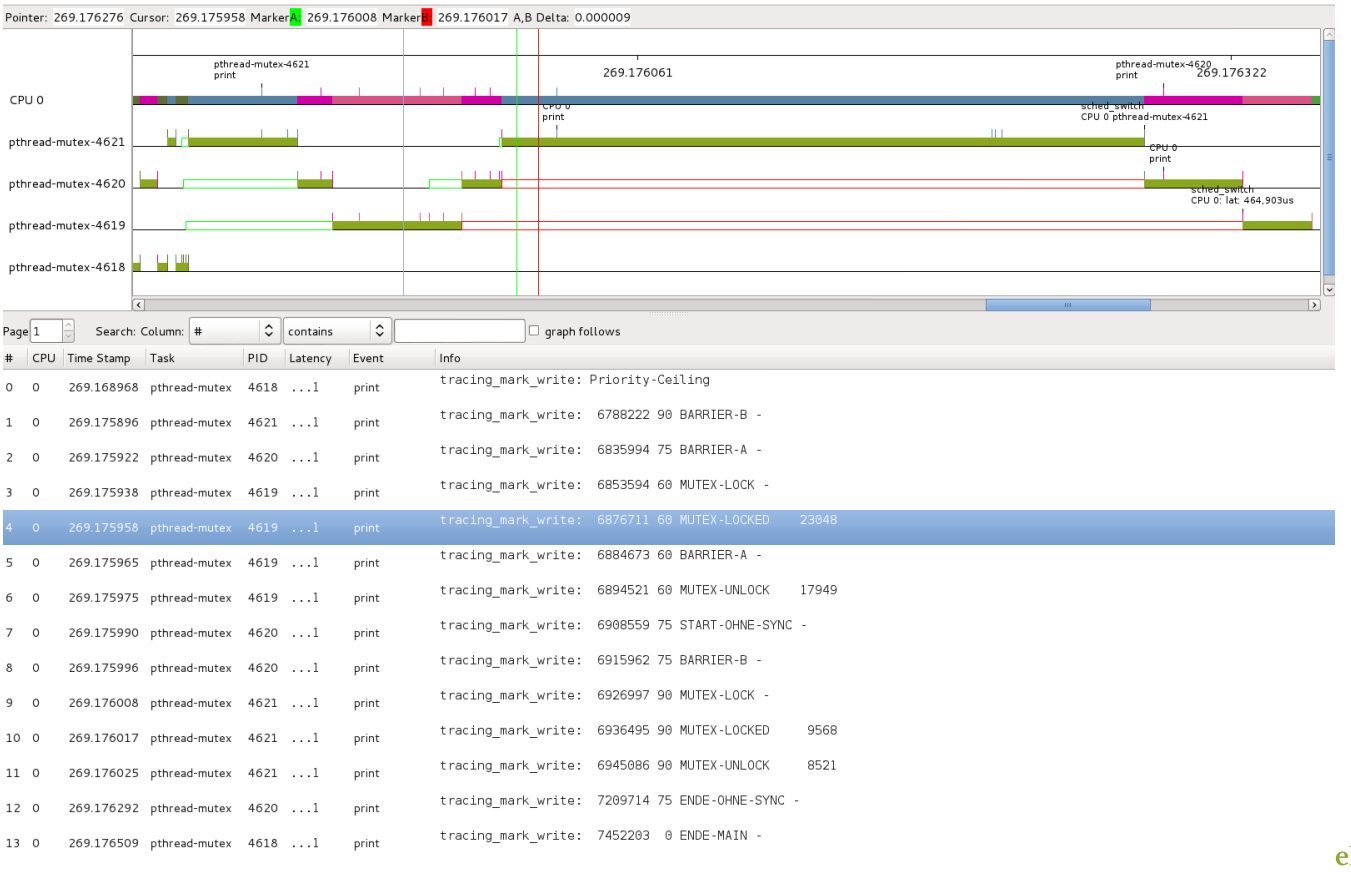
2. Obwohl Task mit mittlerer Priorität zwischenzeitlich rechenbereit wird, muß dieser warten, bis der den Mutex lockende Task diesen wieder freigegeben hat

3. Nachdem der Lock freigegeben wird, fällt der Task wieder auf seine ursprüngliche niedrige Priorität zurück und es wirken wieder die statischen Prioritäten

4. auch der Task mit der hohen Priorität wird auf die Ceiling-Priorität angehoben, falls diese höher als seine eigene Priorität ist

   Ceiling-Priorität sollte so hoch sein, wieder der Task mit der höchsten Priorität, welcher diese Resource (diesen Mutex) nutzt

Priority-Ceiling (kernelshark)



el

Szenario: `./pthread-mutex -c`

- siehe oben
- im Beispiel: PThread-Mutex mit Priority-Ceiling
- als Ceiling-Priority wurde 90 eingestellt

Ergebnis:

- Task-60 bekommt bereits beim Locken des Mutex die Ceiling-Priorität von 90 zugewiesen
- dadurch kann er ungehindert rechnen, bis er den Mutex wieder freigibt
- Task-75 wird rechenbereit, während Task-60 den Mutex hält und bekommt keine Rechenzeit zugewiesen, weil seine Priorität 75 kleiner als die Ceiling-Priorität 90 ist
- sobald der Task-60 den Mutex wieder freigibt, wird er durch den Task-75 unterbrochen und dieser wiederum durch den Task-90
- es kommt zu keiner Prioritätsinversion mehr, weil der jeweils lockende Task die höchste Priorität besitzt

  $\Rightarrow$ Locking-Konflikte werden vermieden

RT-Mutex I

- besitzenden Task (Owner)
- Liste mit wartenden Tasks
- unterstützen Prioritätsvererbung (Priority Inheritance)
  ⇒ Prioritätsinversion kann nicht auftreten
- sind unterbrechbar (Preemption enabled)
- `rt_mutex_interruptible()`-Variante

linuxhotel

RT-Mutex II

- `rt_mutex_timed_lock()` unterstützt Timeout beim Warten auf Lock;
  Aufwecken erfolgt mit hrtimer-Framework
  → kein Jiffies-Timeout
  ⇒ effizient implementiert
- höherer Overhead im Vergleich zum „normalen“ Mutex
  ⇒ für Echtzeit-Tasks verwenden
- Mutexe in der NPTL: Aufgebaut mittels Futexen
  → PI-Futexe benutzen kernelseitig RT-Mutexe für blockierendes Warten
- Einstellen der Priorität im ftrace sichtbar
  Event: `sched:sched_pi_setprio`

linuxhotel

RT-Mutex III

- `CONFIG_DEBUG_RT_MUTEXES` ermöglicht die Erkennung von Deadlocks und semantischen Fehlern (im Syslog sichtbar)

linuxhotel

RT-Mutex — Ausgaben im ftrace

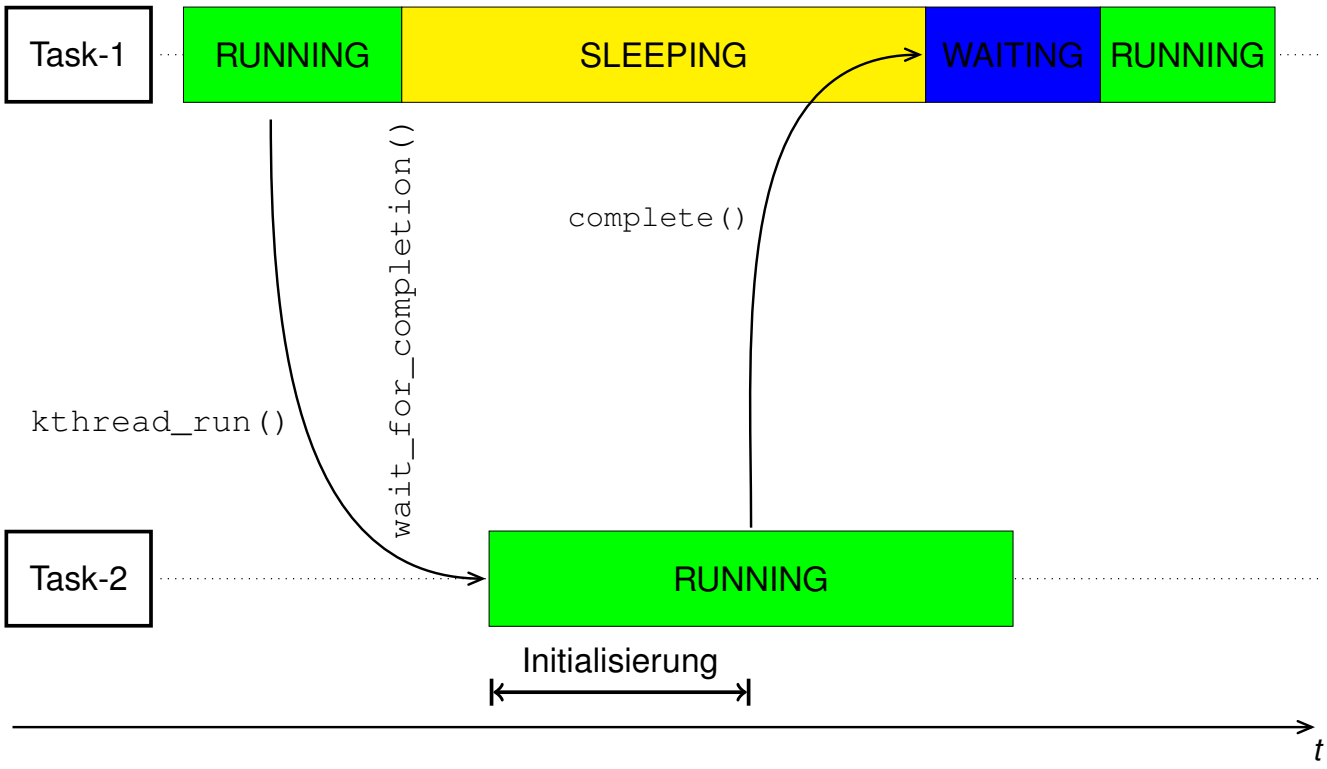
```
mount -t debugfs nodev /debug
cd /debug/tracing

echo sched:* > set_event

echo > trace
```

Modul laden, RT-Task starten

```
cat trace > /tmp/t2.log
less /tmp/t2.log
```

linuxhotel

Completion



linuxhotel

- Task-1 startet einen Kernel-Thread und wartet bis dieser gestartet und fertig initialisiert ist mit `wait_for_completion()`
- Task-2 führt Initialisierungen durch und benachrichtigt Task-1 darüber mit `complete()`
  $\Rightarrow$ Task-1 wird wieder rechenbereit (Waiting)
- Task-1 bekommt rechenzeit zugewiesen (Running)

Completion I

- Synchronisationsobjekt mit dem Zweck anderen Ablaufpfad über die Beendigung eines Vorganges zu informieren
- ein Ausführungspfad wartet mit der Funktion `wait_for_completion()` bis ein anderer mit der Funktion `complete()` die Beendigung mitteilt
- das Wechselspiel warten und benachrichtigen kann beliebig oft wiederholt werden
- falls benachrichtigt wird, bevor jemand darauf wartet, wird nicht gewartet
- intern über einen Zähler realisiert

linuxhotel

Completion II

- die Funktion `complete_all()` benachrichtigt alle wartenden Tasks; Completion-Objekt kann dann aber nicht ohne erneute Initialisierung verwendet werden
- Hauptanwendung: Benachrichtigung über Fertigstellung bzw. Erreichung eines bestimmten Zustandes

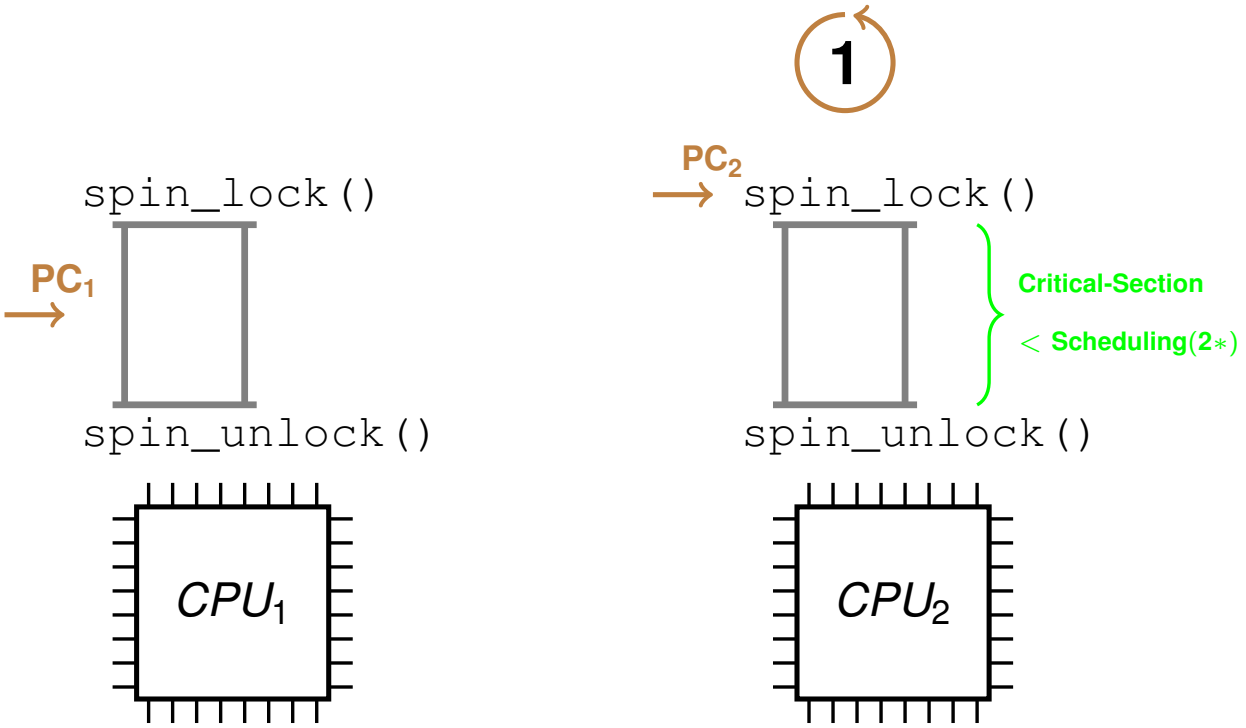
linuxhotel

12 aktiv wartende Synchronisierung

- Spinlock
- RW Lock
- Sequence-Lock

linuxhotel

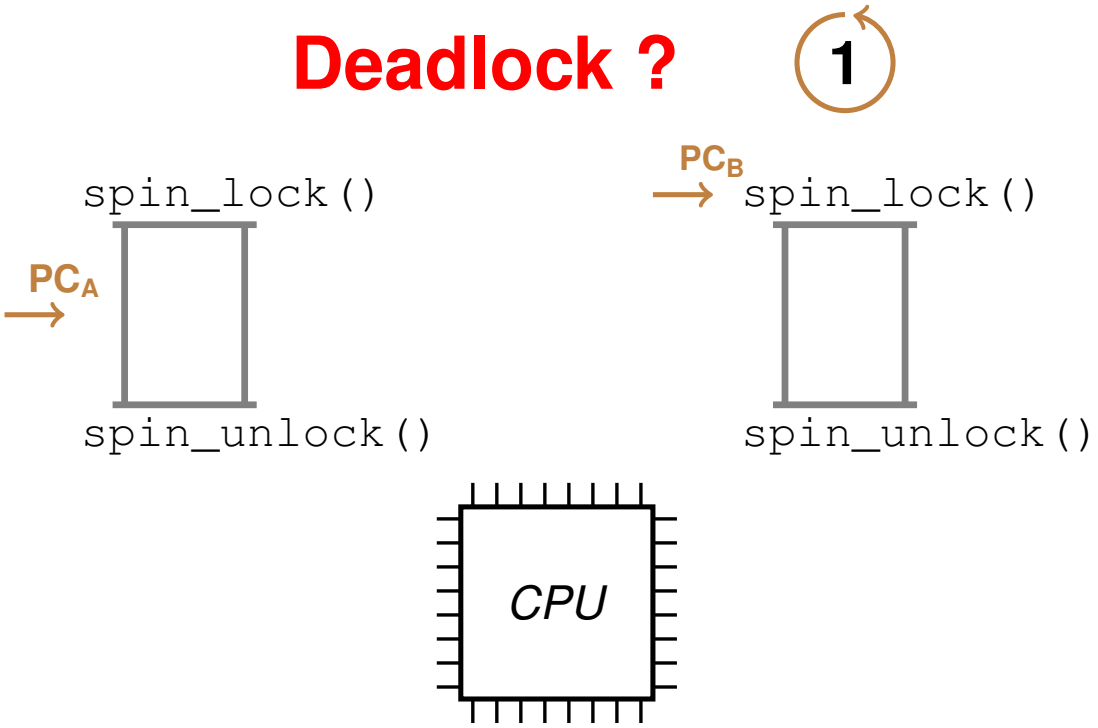
Spinlock auf SMP-System



linuxhotel

- Locken durch Busy-Waiting während Lock von anderer CPU gehalten wird
- sehr effiziente Implementierung in Assembler mit minimalsten Overhead; eigentliche Lock ist ein Bit in einem Integerwert
- gut geeignet für kurze Haltezeiten
- nicht rekursiv
- sowohl im Prozeß- als auch im Interrupt-Kontext
- spezielle Funktionen zum disablen von Interrupts oder SoftIRQ's (`_bh()`, `_irqsave()`, `_irqrestore()`)
- Variante `spin_lock_irq()` gefährlich: Interrupts disablen und enablen ohne Berücksichtigung des vorhergehenden Zustandes

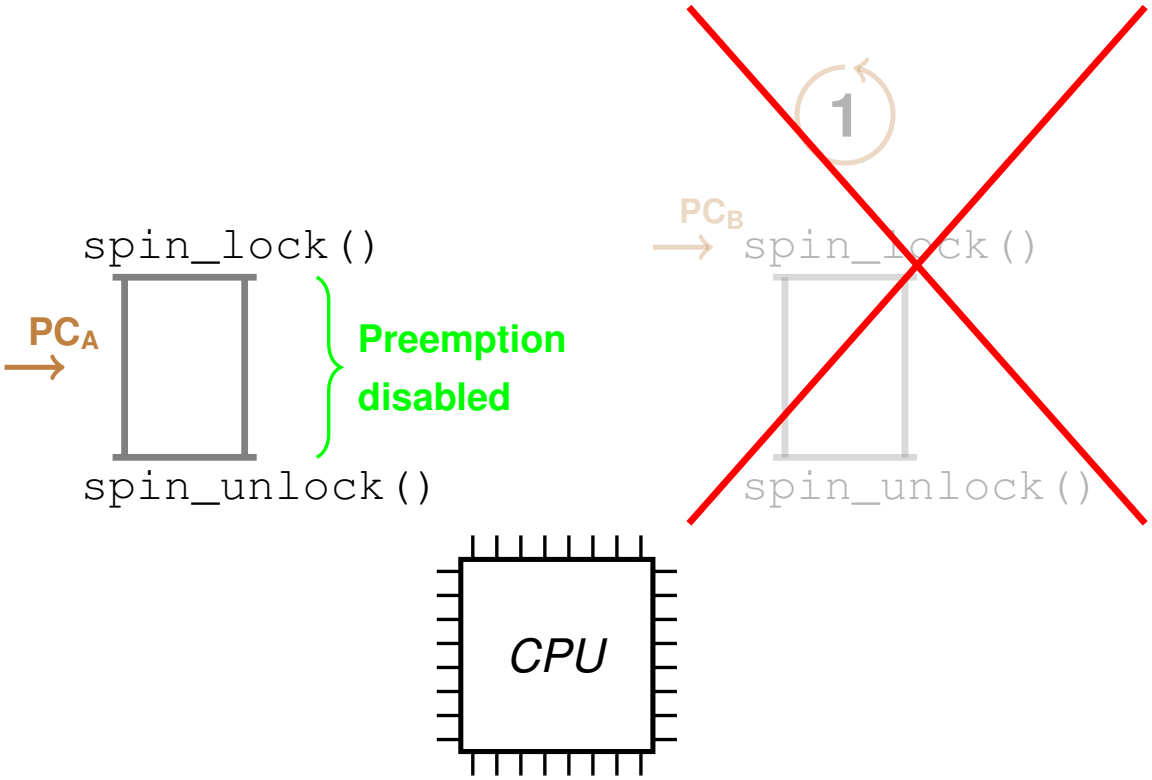
Deadlock auf UP-System?



linuxhotel

- Spin-Lock wird auf Einprozessormaschine (UP-System) eingesetzt oder
- beide Ausführungskontexte sind „zufällig“ auf der gleichen CPU
- führt dies zu einem Deadlock?

Spin-Lock auf UP-System



linuxhotel

- nur Abschalten der Preemption
  auch auf SMP-Systemen wird Preemption abgeschaltet

  ⇒ innerhalb kritischen Abschnitts darf nicht blockiert werden

- wenn Premption abgeschaltet:
  Scheduler-Funktion beendet sich sofort wieder

  ⇒ kein Blockieren oder Warten

- alle Spin-Locks schalten die Preemption ab

Beispiel: Spin-Lock

```
spinlock_t lock1;

// Initialisierung
spin_lock_init (&lock1);

// Verwendung
unsigned long flags;

spin_lock_irqsave (&lock1, flags);

// kritischer Abschnitt:
//   Preemption disabled
//   kein Warten, Blockieren, ...

spin_unlock_irqrestore (&lock1, flags);
```

linuxhotel

RW-Lock — Reader-Writer-Spin-Lock I

- Implementierung von Reader-Writer-Locks basierend auf Spin-Locks
- beliebig viele Leser können gleichzeitig Daten lesen
- nur ein Schreiber kann gleichzeitig Daten ändern
- Schreiber muß warten, bis alle Leser den kritischen Abschnitt verlassen haben
  $\Rightarrow$ Schreiber können ausgesperrt werden
- Funktionen analog zu Spin-Locks implementiert
- mehr Overhead durch Leser-/Schreiber-Logik
  $\rightarrow$ RCU's häufig effizienter einsetzbar

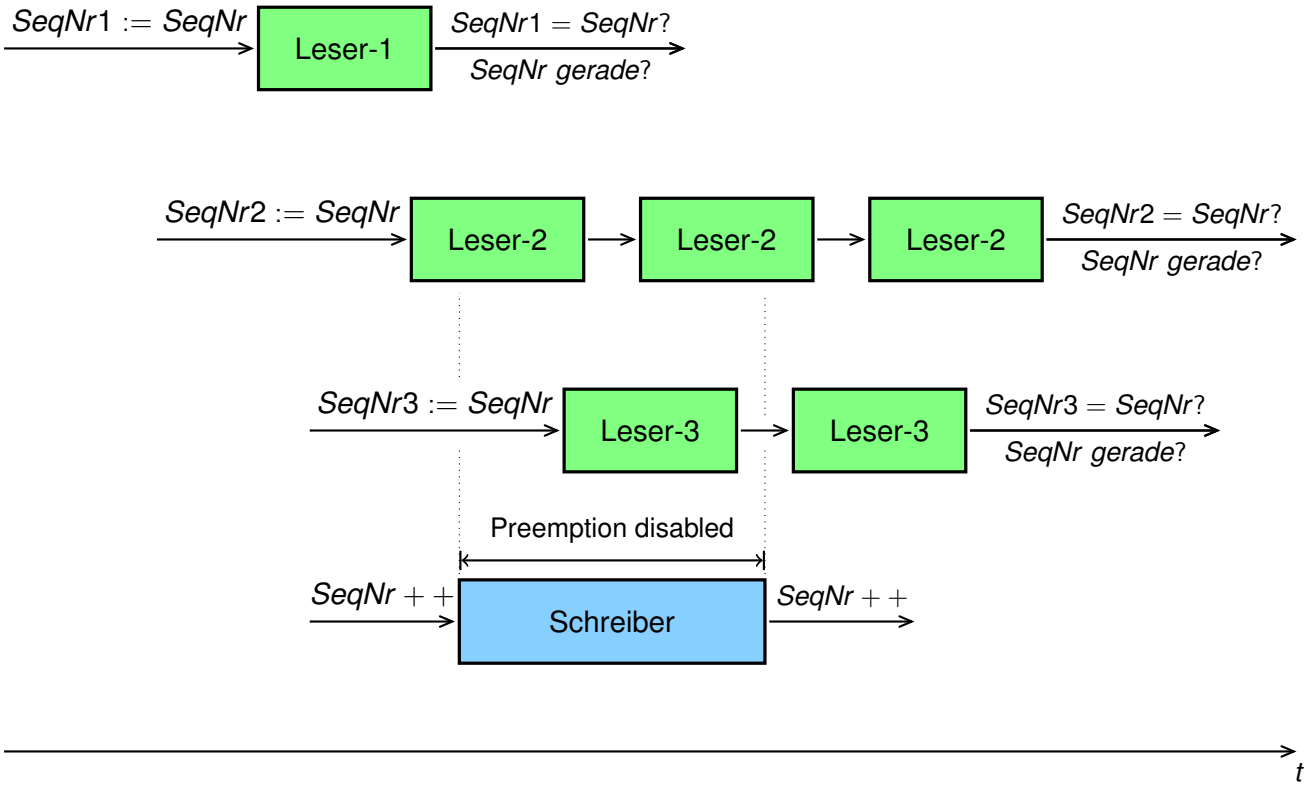
linuxhotel

RW-Lock — Reader-Writer-Spin-Lock II

- Funktionen:
  `read_lock()`
  `write_lock()`
  `...`

linuxhotel

Sequence-Lock



linuxhotel

- Leser-1 merkt sich Sequence-Number in Variablen *SeqNr*1
- nach dem Lesen (ggf. Wegkopieren) der Daten wird geprüft, ob sich:
  Sequence-Number geändert hat und
  Sequence-Number gerade ist
  → beides ist nicht der Fall
  ⇒ Lesen war erfolgreich
- Leser-2 merkt und prüft Sequence-Number ebenso:
  → Sequence-Number hat sich geändert
  ⇒ Lesen war nicht erfolgreich
- Leser-3 merkt und prüft Sequence-Number:
  → Sequence-Number ist ungerade
  ⇒ Lesen war nicht erfolgreich
- Schreiber inkrementiert Sequence-Number beim Betreten und Verlassen des kritischen Abschnitts
- zusätzlich hält Schreiber bei der `seqlock_t`-Implementierung ein Spin-Lock:
  ⇒ Schreiber gegenseitig im Prozeß-Kontext geschützt
  ⇒ Preemption ist disabled
- Interrupt-Kontext: Interrupt-sichere Varianten benutzen
  `write_seqlock_irqsave()`, ...

Sequence-Lock I

- Datentyp `seqlock_t` in `<linux/seqlock.h>`
- Sequence-Nummer wird bei jedem Schreibvorgang inkrementiert:
  Start-Schreibvorgang: ⇒ ungerader Zahlenwert
  Ende-Schreibvorgang: ⇒ gerader Zahlenwert
- Leser holt sich vor dem Lesen die Sequence-Nummer:
  ungerade: Sequence-Nummer erneut holen (Schreiber schreibt gerade)
  gerade: Lesevorgang starten
- Leser vergleicht nach dem Lesen die Sequence-Nummer:
  ungleich: Lesevorgang erneut starten (Schreiber war während des Lesevorganges zugange)
  identisch: Lesevorgang war erfolgreich und wird abgeschlossen

linuxhotel

Sequence-Lock II

- Schreiber müssen nicht auf Leser warten (vgl. RW-Lock)
- kritische Abschnitt des Schreiber enthält (normales) Spin-Lock
  → Preemption abgeschaltet
  ⇒ keine blockierenden Funktionsaufrufe
- Leser enthalten gar keine zusätzliche Synchronisation
  → Synchronisierung ausschließlich über Sequence-Nummer
- Achtung: Leser können in „Dauerschleife“ verfallen, wenn zu häufig geschrieben wird
- Schreiben im Interrupt-Kontext:
  `write_seqlock_irqsave()`,
  `write_seqlock_irqrestore()` (HW-ISR)
  `write_seqlock_bh()`, `write_seqlock_bh()` (SoftIRQ)

linuxhotel

Sequence-Lock III

- nicht für Daten geeignet, die über Zeiger referenziert werden
  Zeiger könnten während Schreibvorgang ungültig werden
- Variante `seqcount_t` in `<linux/seqlock.h>` mit entsprechenden Funktionen enthält keine Schreiber-Synchronisierungen
  $\Rightarrow$ verwendbar, wenn im Schreiber bereits geeignete Synchronisierung vorhanden

linuxhotel

Beispiel: Seq-Lock

```
// Initialisierung
seqlock_t seqlock1;
seqlock_init (&seqlock1);

// Schreiber
write_seqlock (&seqlock1);
// kritischer Schreiber-Abschnitt
write_sequnlock (&seqlock1);

// Leser
unsigned long seqnr;

do
{
  seqnr = read_seqbegin (&seqlock1);
  // kritischer Leser-Abschnitt
} while (read_seqretry (&seqlock1, seqnr));
```

linuxhotel

13 minimalste Synchronisierung
- atomare Variablen
- kfifo – Kernel-FIFO
- Ringspeicher
- Read-Copy-Update (RCU)

atomare Variablen I

- einfache Integer-Operationen (addieren, subtrahieren) sind nicht zwangsläufig atomar
  → abhängig von Architektur
- Integer-Operation kann mit kritischem Abschnitt gesichert werden
  ⇒ Synchronisations-Overhead größer als Netto-Operation
- Linux stellt Satz architekturunabhängiger atomarer Operationen zur Verfügung
- Datentyp für atomare Operationen:
  `atomic_t`
- Operationen arbeiten ausschließlich mit diesem Datentyp
  ⇒ keine Verwechslung möglich
- atomare Operation und Test auf einen bestimmten Wert möglich

atomare Variablen II

- siehe auch:
  `Documentation/atomic_ops.txt`

linuxhotel

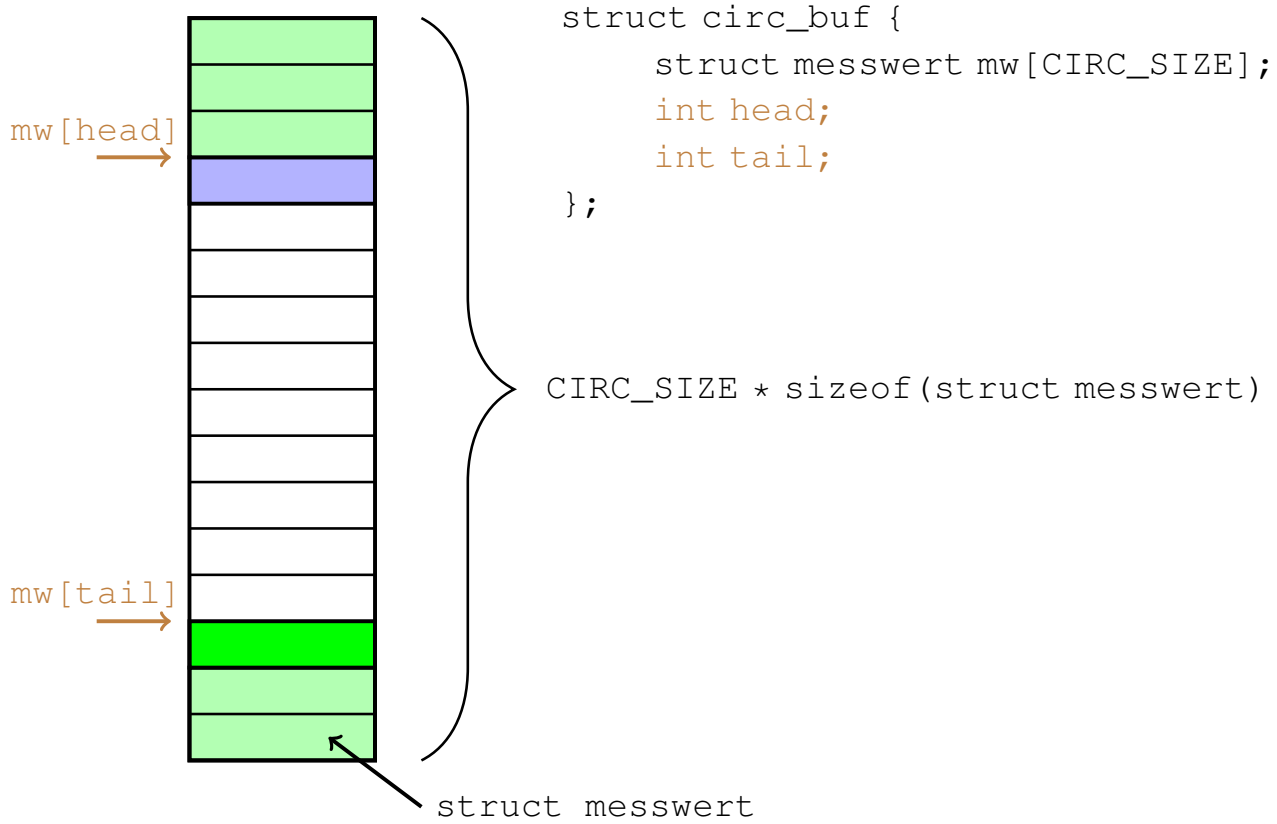
kfifo I

- Implementierung nach `<linux/kfifo.h>`
- Kopieren von Userspace-Daten im Kernel-Treiber in oder aus dem kfifo
- eigentliche Datenzugriff im kfifo implementiert und abgesichert durch Memory-Barriers
- bei mehreren Schreibern oder Lesern müssen diese sich gegenseitig absichern
  $\rightarrow$ max. ein Leser und max. ein Schreiber gleichzeitig erlaubt
- nicht geeignet für große Datenmengen, da jedesmal die Daten zweimal kopiert werden
- `CONFIG_SAMPLE_KFIFO` erstellt Beispielmodule mit kfifo's

kfifo II

- siehe Beispiele:
  `samples/kfifo`

Ringspeicher



linuxhotel

Ringspeicher I

- Implementierung nach `<linux/circ_buf.h>`
- $< n >$ Speicherbereiche werden unabhängig voneinander beschrieben und gelesen
- Anzahl der Speicherbereiche: 2^x (immer Zweierpotenz zur Vereinfachung der Berechnung)
- Verzicht auf Locking zwischen Leser und Schreiber durch Benutzung jeweils unterschiedlicher Speicherbereiche
- es darf max. einer gleichzeitig Lesen und max. einer gleichzeitig Schreiben
  $\Rightarrow$ Schreiber gegenseitig serialisieren und
  $\Rightarrow$ Leser gegenseitig serialisieren

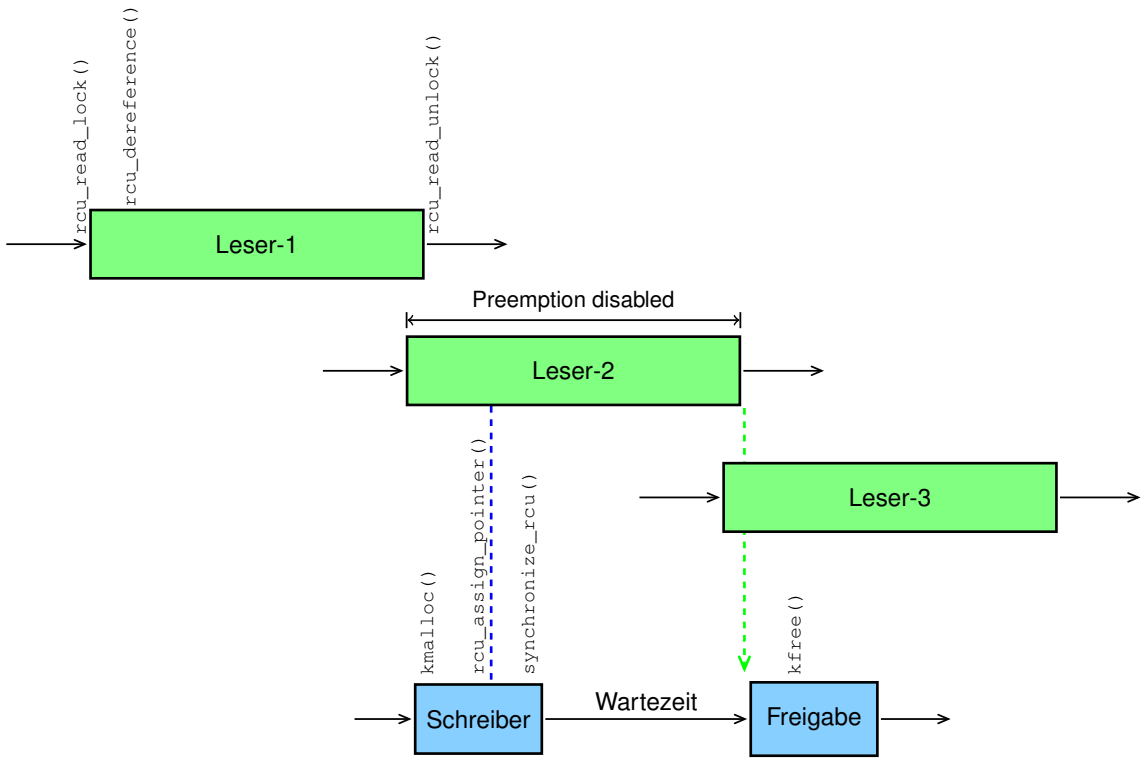
linuxhotel

Ringspeicher II

- Makro `CIRC_SPACE()` liefert minimale Anzahl freier Speicherbereiche
  → Schreiber bekommt freien Platz für neue Daten
- `CIRC_CNT()` liefert minimale Anzahl belegter Speicherbereiche
  → Leser bekommt belegten Platz mit Daten
- mittels Memory-Barriers muß die Reihenfolge des Lesens und Schreibens zwischen Index und Daten festgelegt werden
- keine zusätzliche Speicherallokation zur Laufzeit
  → Was tun bei Buffer-Overflow?
- siehe auch:
  `Documentation/circular-buffers.txt`

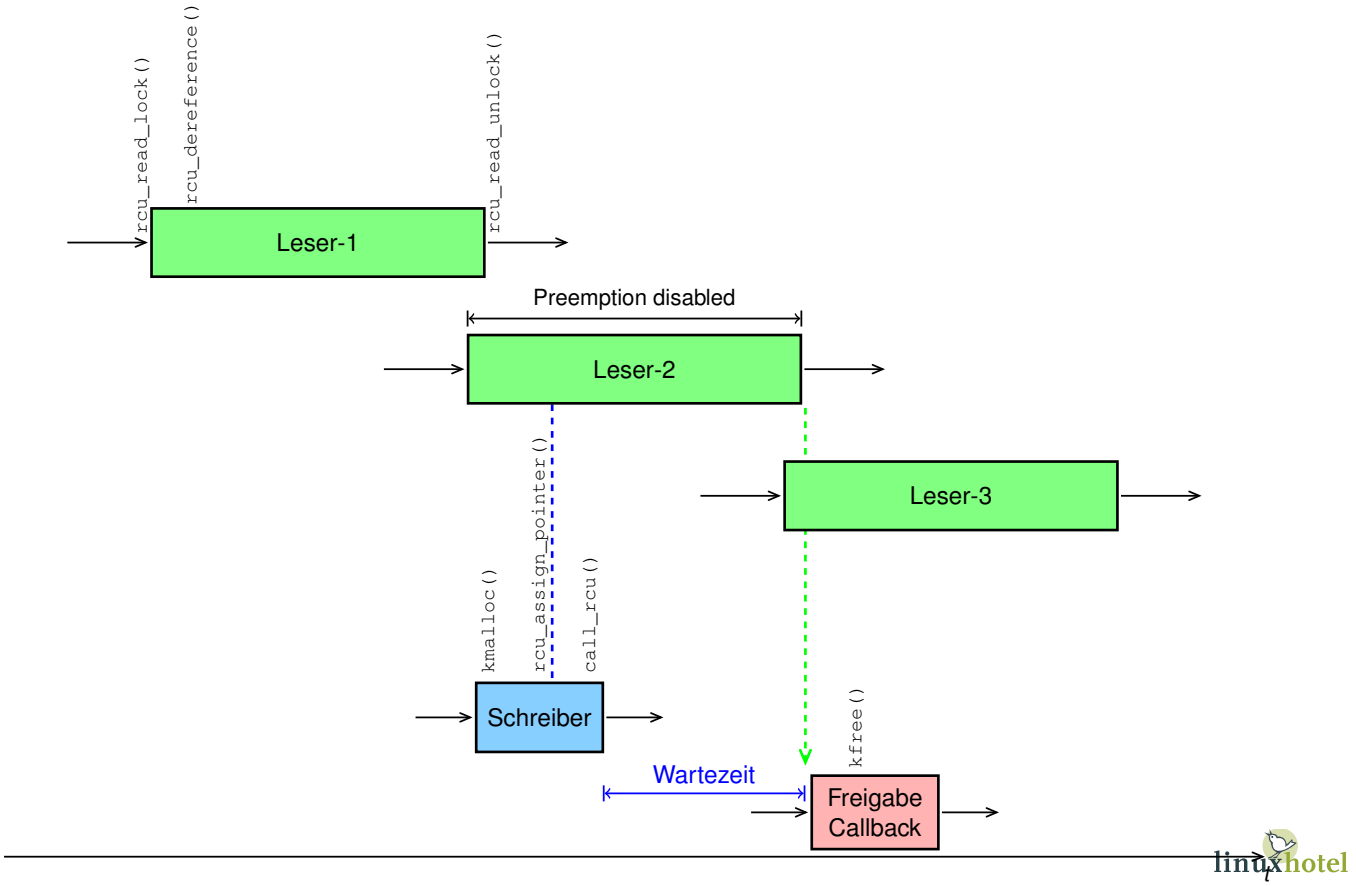
linuxhotel

RCU — Synchronize



- leseseitige kritische Abschnitt mittels `rcu_read_lock()` und `rcu_read_unlock()` gekennzeichnet
- `rcu_dereference()` holt einen geschützten Zeiger
- Schreiber allokiert Datenbereich und füllt diesen (`kmalloc()`)
- `rcu_assign_pointer()` weist dem geschützten Zeiger den neuen Datenbereich zu
- alle Leser welche ab diesem Zeitpunkt den Zeiger referenzieren bekommen neuen Zeiger
- alte Datenbereich darf erst gelöscht werden, wenn alle Leser die zum Zeitpunkt des Austausches aktiven kritischen Abschnitte verlassen haben; `synchronize_rcu()` wartet darauf →*Grace-Period*
- Nachdem alle Leser die potentiell betroffenen kritischen Abschnitte verlassen haben, kann der alte Datenbereich freigegeben werden (`kfree()`)

RCU — Callback



- Schreiber wartet in diesem Szenario nicht, sondern installiert eine Callback-Funktion mit dem Aufruf `call_rcu()`
- nach Ablauf der Grace-Period (Wartezeit), wenn also alle betroffenen Leser den kritischen Abschnitt verlassen haben, wird die Callback-Funktion zur Freigabe des alten Datenbereiches aufgerufen
- es ist der RCU-Implementierung überlassen, wie die Callback-Funktion aufgerufen wird;
  dies kann im Interrupt-Kontext (SoftIRQ) erfolgen
  $\Rightarrow$ Callback-Funktion muß atomar sein

Read-Copy-Update (RCU) I

- Synchronisation ausschließlich über Zeiger referenzierter Daten (z. B. Listen)
- Vorraussetzung: Zeigeroperationen sind atomar abgebildet
- Lesen über dereferenzierten Zugriff auf Daten ohne zu blockieren
- Schreiben mittels Kopieren der Daten, Ändern und Austauschen der Zeigerreferenzierung
- Leser greift immer auf konsistente Daten zu; gegebenenfalls veraltet

linuxhotel

Read-Copy-Update (RCU) II

- Löschen von nicht mehr benötigten Speicherbereichen durch:
  1. Warten, bis letzte Leser kritischen Abschnitt verlassen hat (`synchronize_rcu()`)
  2. asynchroner Aufruf (atomar — im Interrupt-Kontext!) einer Callback-Funktion, wenn letzter Leser Zugriff abschließt (`call_rcu()`)
     ⇒ Schreiber blockiert nicht
     → Callback-Funktion wird ggf. im Interrupt-Kontext aufgerufen
- Schreiber gleichberechtigt behandelt wie Leser
- „Locking-Mechanismus“ der nicht lockt!
- `CONFIG_PREEMPT`: Preemption beim Leser abgeschaltet
- `CONFIG_PREEMPT_RT`: Preemption aktiv

linuxhotel

Read-Copy-Update (RCU) III

- Variante SRCU: Preemption immer aktiv
  $\Rightarrow$ ausschließlich Prozeß-Kontext
- zwei Implementierungen:
  - *Tree-RCU*: aufwändige Implementierung für Systeme mit vielen CPU's
  - *Tiny-RCU*: einfache, schlanke Implementierung für kleine Systeme
- `call_rcu()` im Interrupt-Kontext einsetzbar
- jeder offene Callback verursacht Speicherverbrauch
  $\rightarrow$ kann außer Kontrolle geraten bei hoher Update-Frequenz
- `synchronize_rcu()` begrenzt Anzahl ausstehender „Aufräumvorgänge"und limitiert gleichzeitig die Update-Frequenz

Read-Copy-Update (RCU) IV

Einsatzmöglichkeiten

- Daten werden „meistens" gelesen
  Faustwert: weniger als 10 % der Zeit geschrieben
- Daten gruppiert und mittels Zeiger referenzierbar
  $\rightarrow$ Listenoperationen prädestiniert dafür
- Synchronisierung ohne Blockieren
- veraltete Daten und zusätzlicher Speicherverbrauch können akzeptiert werden

Read-Copy-Update (RCU) V

Tracing

- `CONFIG_RCU_TRACE`: eigener Tracer für RCU's im debugfs:
  ```
  mount -t debugfs nodev /debug
  cd /debug/rcu
  ```
- `CONFIG_RCU_TORTURE_TEST`: Kernel-Modul für RCU-Stresstest
- `CONFIG_SPARSE_RCU_POINTER`: Kennzeichnung von RCU-geschützten Zeigern mit `__rcu`; Ausgabe einer Warnung wenn diese Zeiger nicht RCU-geschützt verwendet werden

linuxhotel

Read-Copy-Update (RCU) VI

siehe auch

```
Documentation/RCU/:
whatisRCU.txt
checklist.txt
listRCU.txt, ...

http://lwn.net/Articles/262464/
http://lwn.net/Articles/253651/
http://lwn.net/Articles/220677/
```

linuxhotel

RCU in Listenoperationen

- eigene RCU-Listenfunktionen, wie z. B.:
  `list_for_each_entry_rcu()`
  `list_add_rcu()`
  `list_del_rcu()`
  `list_replace_rcu()`

- Funktionen enthalten Barriers, um Reordering zu unterbinden (wo notwendig)

- Datenänderung atomar darstellen:
  → zusätzliche Sicherung (Bsp.: Spin-Lock)

- gelesene Daten können „veraltet“ sein

linuxhotel

14 Kernel-Debugging

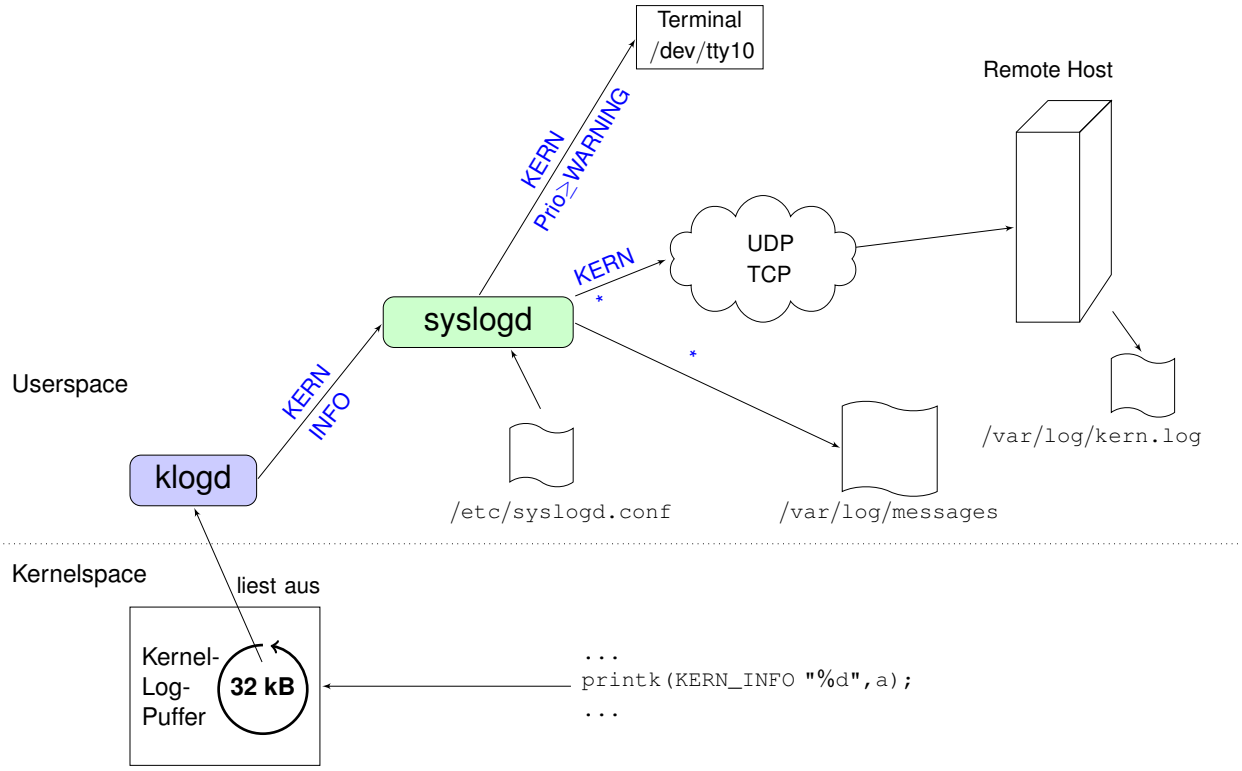
linuxhotel

printk-Debugging I

- Funktion `printk()` schreibt in Kernel-Log-Ringspeicher
- `klogd`-Dämon liest Logmeldungen aus und übergibt diese an `syslogd`-Dämon, welcher wiederum ins Systemlog (`/var/log/messages`) schreibt
- primitives Debugging; typische Verwendung mittels `#define`-Makros
- sehr stabil und auch im Interrupt-Kontext verwendbar
- Kernel-Log-Puffer abfragen:
  `dmesg`
- System-Log abfragen:
  `tail -f /var/log/messages`

linuxhotel

Syslog-Dämon



linuxhotel

procfs-Debugging I

- Treiber legt Dateien im `/proc`-FS mit Debugging-Informationen an
- bei Abfrage der Datei (z. B. `cat /proc/debug-kio`) werden Daten aktuell generiert
- belastet Laufzeitverhalten nicht, wenn keine Abfragen erfolgen
- bedarfsorientiert aktivierbare Informationen aus dem Systemkern / Kernel-Treiber
- auch im fertigen, ausgelieferten Treiber verwendbar

linuxhotel

procfs-Debugging II

Anwendungsfälle

- Status- und Fehler-Informationen von Kernel-Treibern
- Logging bei Bedarf
- Parametrierung des Treibers

linuxhotel

SysRq-Abfragen I

- in der Console können Kernelinformationen abgefragt oder das Verhalten beeinflußt werden
- Tastenkombination:
  `<Alt><Druck><Key>` (`0x54`)
- Kernel-Konfiguration `CONFIG_MAGIC_SYSRQ` notwendig
- Beispiele:
  - h Hilfe
  - g kgdb aktivieren
  - q Liste mit Timern in hrtimer-Framework
  - 0-9 Log-Level der Console einstellen
- eigene Tastenkombinationen können hinzugefügt werden

linuxhotel

SysRq-Abfragen II

- Beschreibung verfügbarer Tastenkombinationen in Kernel-Sourcen `Documentation/sysrq.txt`

Anwendungsfälle

- Abruf von Systeminformationen in Crash- / Extremsituationen
- Abruf eines System-Snapshots
- Aktivierung des Kernel-Debuggings

kgdb — Kernel-Debugging I

- `kgdb` ist eingebauter Kernel-Debugger
- Debugging erfolgt mit zwei Rechnern
  → Entwicklungsrechner: erstellt Kernel und führt `gdb` als Client aus
  → Testrechner: führt Kernel im Debugging-Modus aus; `kgdb` kommuniziert als Server mit `gdb`-Client
- beide Rechner sind über serielles Nullmodemkabel verbunden; auch Ethernetverbindung ist möglich
- Kernel wird auf dem Entwicklungsrechner erstellt und in den Kernel-Sourcen gedebugged (Debug-Image `vmlinux`)

kgdb — Kernel-Debugging II

- auf dem Testrechner befindet sich lediglich
  - (komprimierte) Kernel (`bzImage`)
  - Symboltabelle (`System.map`)
  - Kernel-Module (`/lib/modules/<Kernel-Version>`)
- in den Kernel eingebauter Debugger kommuniziert mit `gdb`-Client; analog zu `gdbserver`
- Kernel kann während Bootzeit und zur Laufzeit gedebugged werden
- Debugging von ladbaren Kernel-Modulen und von Interrupt-Handlern

linuxhotel

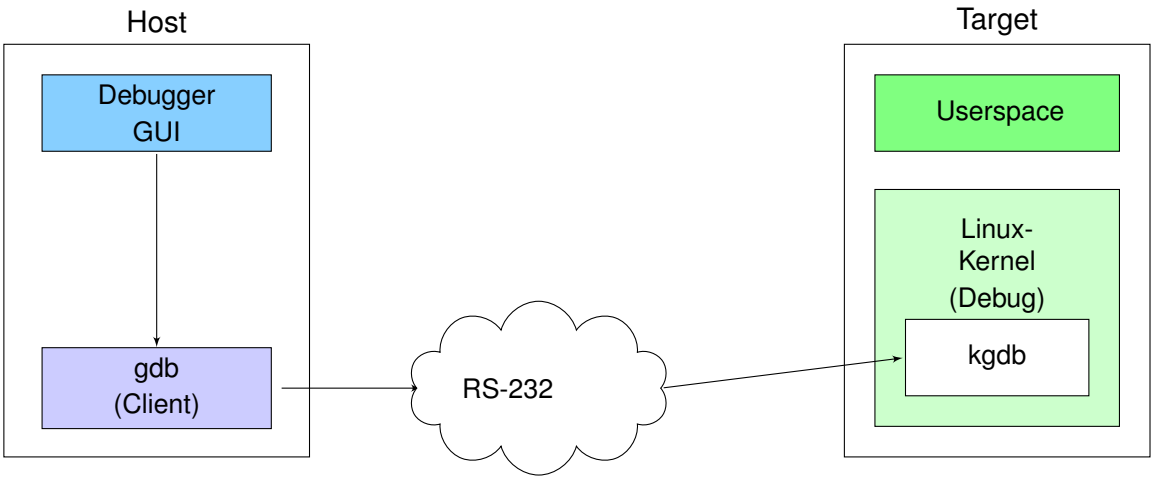
kgdb — Kernel-Debugging III

Anwendungsfälle

- Debugging des Kernels; Was macht der Kernel?
- Interrupt-Debugging
- Kernel-Treiber debuggen
- Analyse des Bootvorganges

linuxhotel

Kernel-Debugging



linuxhotel

Beispiel: Kernel-Debuggen I

Entwicklungsrechner

Kernel-Image, System.map sowie Kernel-Module werden auf das Zielsystem kopiert

```
scp bzImage root@1.2.3.4:/boot/bzImage
scp System.map root@1.2.3.4:/boot/System.map
tar -cvzf my-mod.tgz /lib/modules/<Vers.>
scp my-mod.tgz root@1.2.3.4:/lib/modules/
```

Zielsystem - Console

Kernelmodule auf dem Zielsystem entpacken

```
cd /lib/modules
tar -xvzf my-mod.tgz
```

linuxhotel

Beispiel: Kernel-Debuggen II

Zielsystem

Bootloader wird konfiguriert; Kernel-Kommandozeile erweitern

`kgdbwait`: optional; beim Booten warten auf Debugger

`kgdboc=ttyS0,115200`: Einstellen der seriellen Schnittstelle

Runlevel des init-Dämon so gering wie möglich einstellen in `/etc/inittab`

linuxhotel

Beispiel: Kernel-Debuggen III

Entwicklungsrechner

Zielsystem

serielles Nullmodemkabel zwischen beiden Rechner anschliessen und in beiden Richtungen testen:

```
stty -F /dev/ttyS0 speed 115200
cat /dev/ttyS0
echo Servus > /dev/ttyS0
```

Zielsystem neu starten

Zielsystem - Console

Debugging auf dem Zielsystem starten mittels <SysRq>

`<Alt> <Druck> <g>`

linuxhotel

Beispiel: Kernel-Debuggen IV

Entwicklungsrechner - in Kernel-Sourcen

Entwicklungsrechner verbindet sich zum wartenden Zielsystem dies ist in den Kernel-Sourcen des Zielsystems mit dem entsprechenden Cross-Debugger durchzuführen

Debugger starten

```
arm-linux-gdb ./vmlinux
```

linuxhotel

Beispiel: Kernel-Debuggen V

Entwicklungsrechner - im Debugger

Geschwindigkeit und serielle Schnittstelle einstellen

```
(gdb) set remotebaud 115200

(gdb) target remote /dev/ttyS0
```

Haltepunkt auf eine Kernel-Funktion setzen

```
(gdb) break do_nanosleep
```

Ausführung des Kernels im Zielsystems fortsetzen

```
(gdb) continue
...
```

linuxhotel

kgdb — ladbare Kernel-Module debuggen I

- Kernel-Module können am einfachsten gedebugged werden, wenn diese statisch zum Kernel dazugelinkt werden
- alternativ können sie auch zur Laufzeit als Kernel-Modul geladen werden, müssen aber dann dem Debugger mit den Adressen der vorhandenen Sektionen bekannt gegeben werden
- im Verzeichnis `/sys/module/<Modul-Name>/sections` finden sich die Sektionsadressen als Dateiinhalt (`cat .text, cat .data, cat .bss, ...`)
- im Debugger müssen diese Adressen angegeben werden:

```
(gdb) add-symbol-file drivers/char/kio.ko
0xdd618000 -s .rodata 0x...
-s .data 0x... -s .bss 0x...
```

linuxhotel

kgdb — ladbare Kernel-Module debuggen II

- nun sollten Symbole des Kernel-Moduls sichtbar sein

linuxhotel

Beispiel: Kernel-Module-Debuggen I

Zielsystem - Console

Kernelmodul laden

```
insmod drivers/char/kio.ko
```

Ermittlung der relevanten Sektionsadressen

```
cd /sys/module/kio/sections
cat .text
0xdc76d000

cat .data
0xdc76d218

cat .bss
0xdc76d3f0
```


Beispiel: Kernel-Module-Debuggen II

Entwicklungsrechner - im Debugger

Bekanntmachen der Adressen im Debugger

```
(gdb) add-symbol-file drivers/char/kio.ko
0xdc76d000 -s .data 0xdc76d218
-s .bss 0xdc76d3f0
```

kgdb — Kernel-Konfiguration I

- `CONFIG_KALLSYMS`: Laden von Debugsymbolen (in General setup → Configure standard Kernel features)
- `CONFIG_KALLSYMS_ALL`: Laden von allen Symbolen, nicht nur von Funktionen
- `CONFIG_MAGIC_SYSRQ`: SysRq-Unterbrechung aktivieren (in Kernel hacking)
- `CONFIG_DEBUG_KERNEL`: Kernel-Debugging aktivieren
- `CONFIG_DEBUG_INFO`: Debugsymbole hinzufügen
- `CONFIG_FRAME_POINTER`: Informationen für Stacktrace
- `CONFIG_KGDB`: Kerneldebugger kgdb

linuxhotel

kgdb — Kernel-Konfiguration II

- `CONFIG_KGDB_SERIAL_CONSOLE`: Serielle Console für kgdb
- `CONFIG_KGDB_TESTS`: Starten des kgdb während Bootvorgang

linuxhotel

Teil IV

Timer

Timer

15 Linux-Timer-Wheel

Klassische Timer-Behandlung in Linux

- periodischer Timer generiert mit einer Frequenz von `HZ` (100, 250 oder 1000 Hz) Timer-Interrupts
- Interrupt wird verwendet für
  - Scheduling
  - Process-Accounting
  - programmierbare Software-Timer
  - Zeitbasis und -messung (`jiffies`)

Wie sieht es mit der Echtzeit-Eignung aus?

- feinere Granularität als 1 ms nicht darstellbar
  $\Rightarrow$ Scheduling und Zeitmessung zu ungenau
- Overhead bei jedem Timer-Tick generiert Grundlast
- Leistungsfähigkeit von Hardware-Timer wird nicht genutzt
- großer Teil der Implementierung ist hardwareabhängig
- für Serversysteme gute und pragmatische Implementierung

linuxhotel

16 hrtimer-Framework
- statischer Aufbau
- dynamischer Ablauf
- Anwendung
- Konfiguration und Diagnose

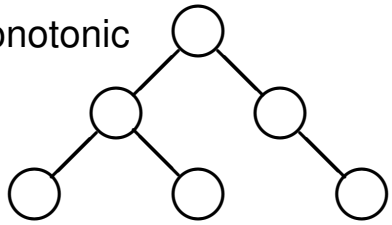
hrtimer-Subsystem

- zentrales Subsystem für Zeitbehandlung mit minimalen Hardwarezugriffen
- hohe zeitliche Auflösung (numerisch 1 Nanosekunde)
- dynamische Ticks sind nicht an periodische jiffies gebunden: Timer-Tick-Emulation inkrementiert jiffies und löst Scheduling sowie Prozess-Profiling aus
- architekturunabhängige API
- plattformabhängige Teil ist auf ein Minimum reduziert
  $\rightarrow$ Portierungsaufwand gering
- effiziente Verwaltung von Timer-Ereignissen in rb-Tree

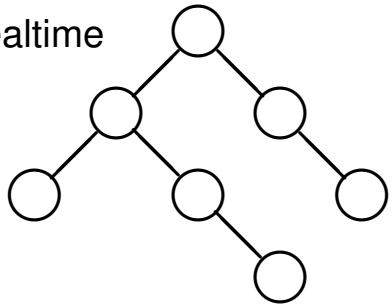
Wie werden Ereignisse verwaltet?

- pro CPU existieren zwei sortierte Red-Black-Bäume (rb-Tree) mit Ereignissen in der Zukunft
- jeder Baum bezieht sich auf eine andere Zeitbasis, gerechnet in Nanosekunden:
  1. seit System-Boot
     → `CLOCK_MONOTONIC`
  2. in Kalenderzeit seit 01.01.1970
     → `CLOCK_REALTIME`

Zeitbasis Monotonic



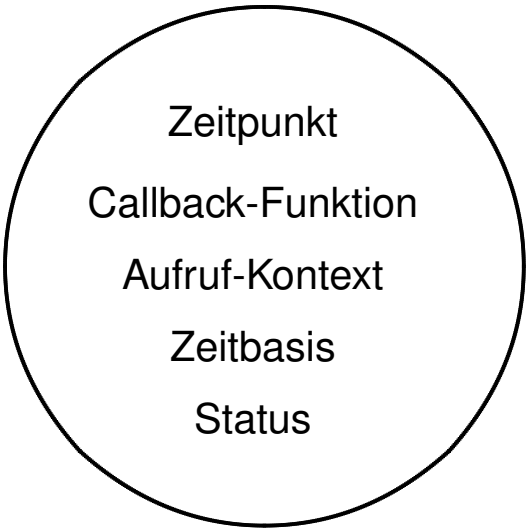
Zeitbasis Realtime



linuxhotel

Was ist ein Ereignis?

- Ereignisse enthalten die Informationen:
  - Ablaufzeitpunkt
  - Callback-Funktion des Timers
  - Aufrufkontext (direkt oder indirekt)
- in Form einer Struktur gekapselt (`struct hrtimer`)



linuxhotel

Was ist eine hrtimer-Callback-Funktion?

- Callback-Funktion ist die Reaktion auf das Eintreten des Timer-Ereignisses
  → eigentliche Ereignisbehandlung; auch benutzerdefinierter Code kann eingehängt werden
- im Kernel vorhandene Callbacks sind z. B.:
  - Scheduling-Events
  - Wartezeiten
  - Timeouts

linuxhotel

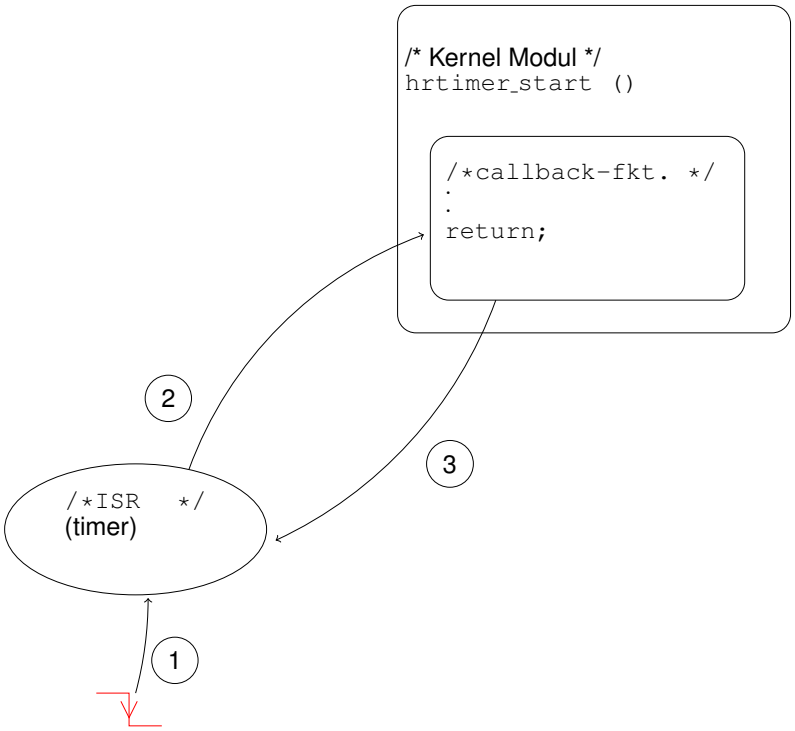
Aufruf-Kontext

durch die Angabe des Aufrufkontextes kann unterschiedliches Verhalten der Callback-Funktion erreicht werden

- direkt aufgerufen durch den HW-Interrupt mit der geringstmöglichen Latenzzeit
- indirekt aufgerufen durch den hrtimer-SoftIRQ
  → durch Software gestartet, wenn Interrupts nicht blockiert werden sollen
  → „Pending-List“ enthält ausstehende Ereignisse

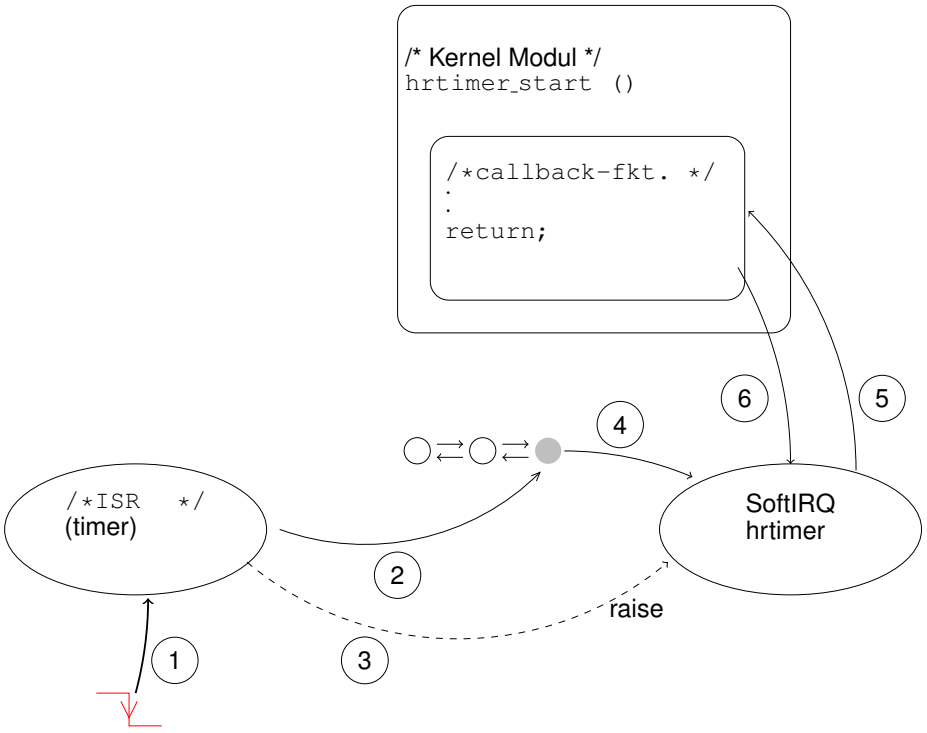
linuxhotel

direkter Aufruf-Kontext



linuxhotel

indirekter Aufruf-Kontext



linuxhotel

Wiederholung von Ereignissen

einmalige Ereignisse

„One-Shot-Timer“ zum angegebenen Zeitpunkt
→ Rückgabewert der Callback-Funktion: `HRTIMER_NORESTART`

wiederholte Ereignisse

Wiederholung durch erneutes Starten des Ereignisses unter Angabe eines beliebigen Wiederholzeitpunktes in der Callback-Funktion
→ Rückgabewert der Callback-Funktion `HRTIMER_RESTART`

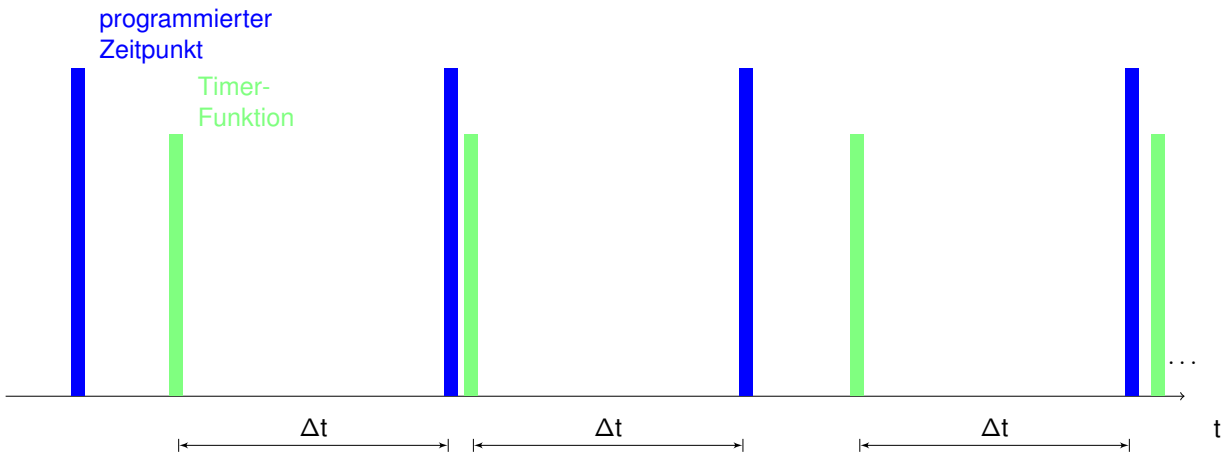
linuxhotel

Wiederholzeitpunkt

- Angabe der Wiederholzeitpunkt bezogen auf den theoretischen Ablaufzeitpunkt liefert eine konstante Anzahl an Abtastzeitpunkten pro Zeiteinheit
- Wiederholzeitpunkt bezogen auf aktuellen Zeitpunkt liefert eine konstante Wartezeit zwischen den Timer-Ereignissen (entspricht `sleep()`-Funktion)
- Einstellung des Wiederholzeitpunktes mit `hrtimer_forward()`; Rückgabewert ist die Anzahl an versäumten Zeitperioden

linuxhotel

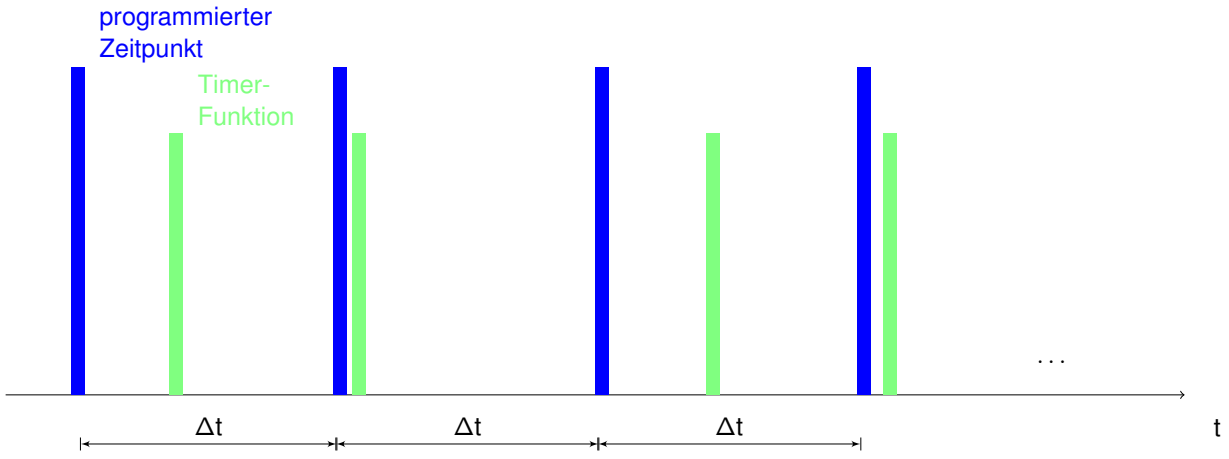
konstante Wartezeit



konstante Wartezeit nach Timerfunktionen

linuxhotel

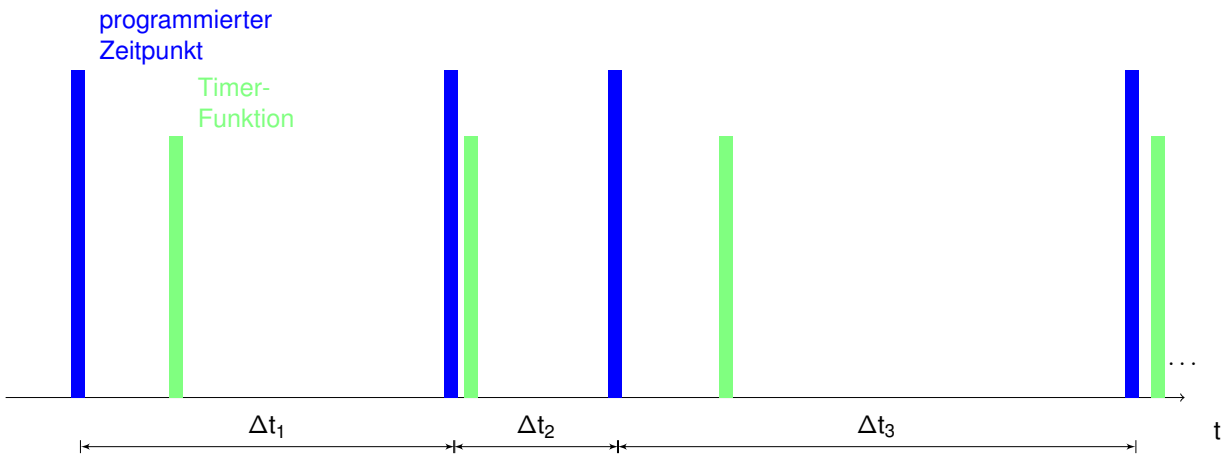
konstante Anzahl an Abtastzeitpunkten



im Mittel pro Zeiteinheit konstante Anzahl an Timerfunktionen

linuxhotel

beliebiger Wiederholzeitpunkt



linuxhotel

Welcher Hardware-Timer wird verwendet? I

- häufig sind von einer CPU aus gesehen mehrere geeignete Zeit-Quellen in der Hardware verfügbar
- jede Zeit-Quelle bewertet sich dazu nach einem Punktesystem
- Kriterien für die Bewertung sind: Timer-Features, Auflösung, ...
- die beste verfügbare Quelle wird dann als Default-Grundlage für das hrtimer-Framework verwendet
- Vorgabe des Timer-Treibers durch Kernel-Kommandozeile: `clocksource=hpet`
- Anzeige und Einstellung der Clocksource im sysfs: `/sys/devices/system/clocksource/clocksourceN`

linuxhotel

Erstellung eines Ereignisses

- ein neues Ereignis wird erstellt und dem hrtimer-Framework übergeben
- Ereignis wird in den entsprechenden rb-Tree einsortiert
- falls dieses Ereignis das nächstliegende im Baum ist, ändert sich der Ablauf

  $\rightarrow$ HW-Timer wird auf dessen Zeitpunkt umprogrammiert

linuxhotel

Timer-Interrupt tritt ein

- das zum Interrupt gehörige Ereignis wird aus dem Baum entfernt
- direkte Callbacks werden aufgerufen
- indirekte Callbacks werden in die Liste der ablaufbereiten Ereignisse (pending list) eingefügt
- der hrtimer-SoftIRQ wird als ablaufbereit gekennzeichnet, damit dieser schnellstmöglich asynchron abgearbeitet wird
- der HW-Timer wird auf das nächste anstehende Ereignis programmiert

linuxhotel

hrtimer-SoftIRQ tritt ein

- Aufgabe ist die Abarbeitung ausstehender Ereignisse, jedoch ohne Interrupts zu blockieren
- hrtimer-SoftIRQ durchsucht die Pending-Liste und ruft die dazugehörigen Callback-Funktionen auf
- ausgeführte Ereignisse werden aus der Liste entfernt

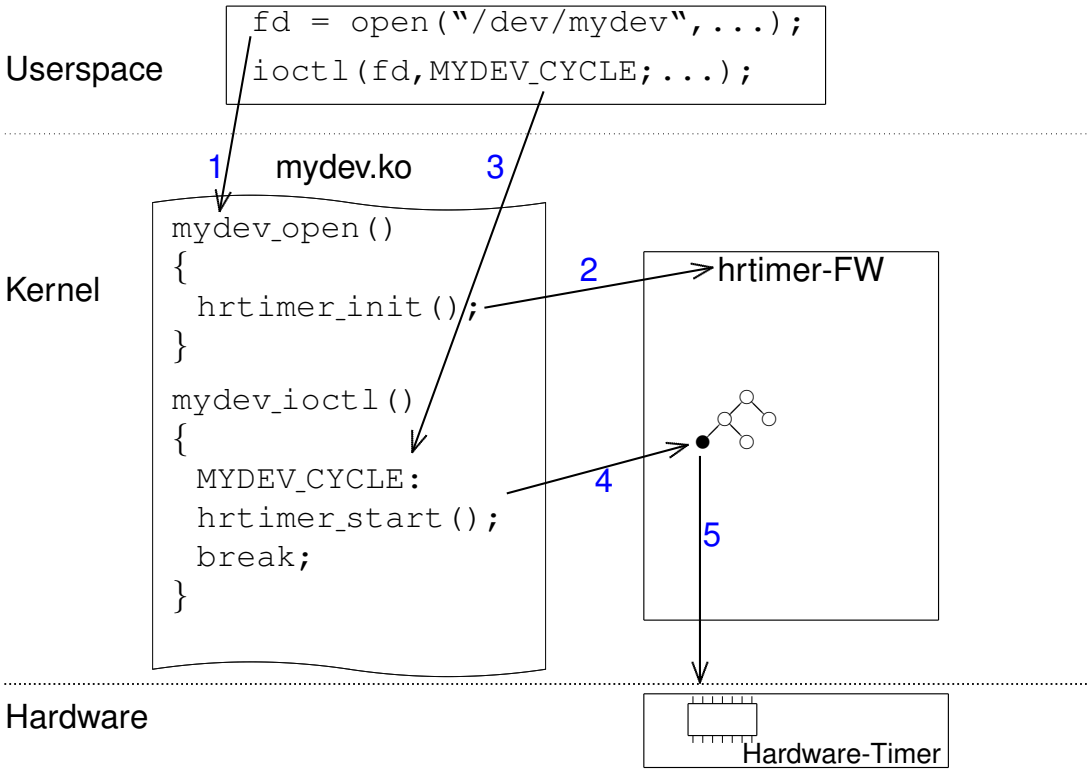
linuxhotel

Callback-Funktion

- die Callback-Funktion ist die eigentliche Reaktion auf ein Timer-Ereignis
- das dazugehörende Ereignis wird als Parameter mitgeliefert
- falls eine Wiederholung des Ereignisses notwendig ist, muß dieses in der Callback-Funktion gestartet werden und der Rückgabewert = `HRTIMER_RESTART` sein
- Interrupt-Context $\Rightarrow$ kein Blockieren, Warten, ...

linuxhotel

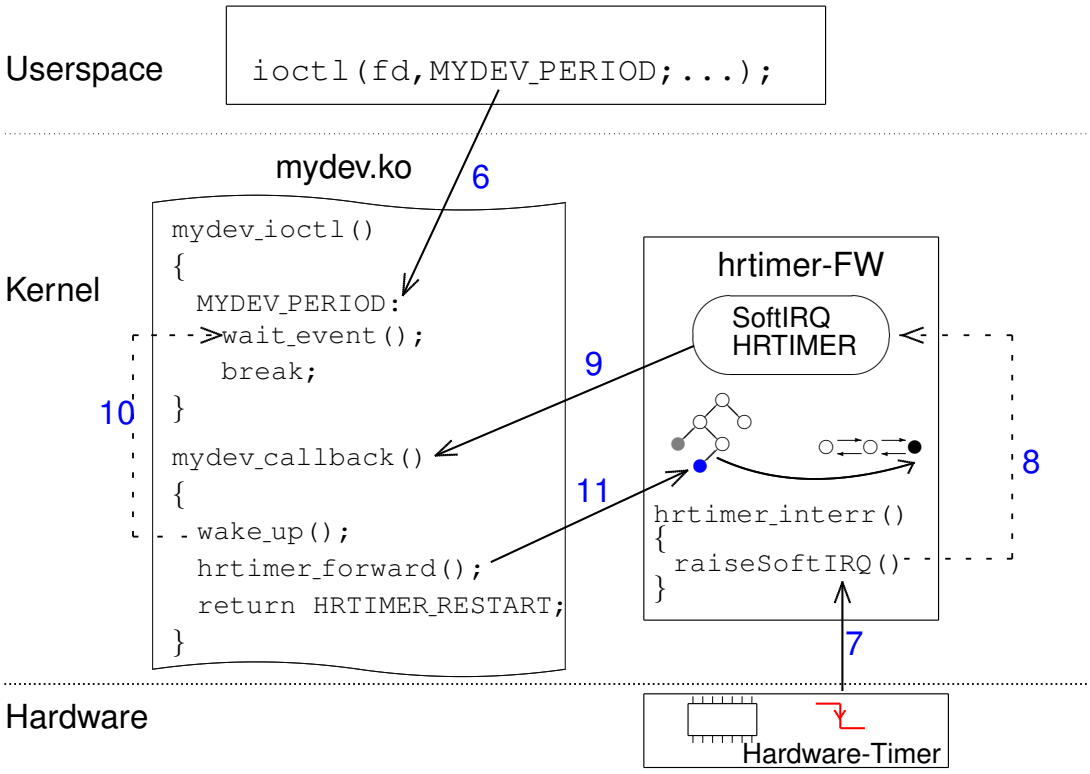
Wie werden hrtimer verwendet? I



linuxhotel

1. Userspace-Prozess öffnet Device-Node des Kernel-Treibers „mydev.ko“. In der `open()`-Funktion wird die hrtimer-Struktur initialisiert.
2. durch den Aufruf von `hrtimer_init()` wir die `hrtimer`-Struktur im Framework initialisiert
3. der Aufruf von `ioctl()` mit dem Kommando `MYDEV_CYCLE` bewirkt im Treiber das Starten des entsprechenden Timers
4. Struktur `hrtimer` wird in den rb-Tree des hrtimer-Frameworks eingefügt
5. (optional) falls dieses Ereignis das chronologisch nächste ist, wird der Hardware-Timer auf den Interrupt-Zeitpunkt programmiert

Wie werden hrtimer verwendet? II



linuxhotel

6 `ioctl()` mit dem Kommando `MYDEV_PERIOD` führt in Folge zu einem `wait_event()`; der aufrufende Userspace-Prozess wird auf einer Warteschlange schlafen gelegt

7 Hardware-Timer interrupted zum programmierten Zeitpunkt und die Interrupt-Service-Routine (ISR) des hrtimers wird ausgeführt:

- das zu diesem Timer-Interrupt gehörige Ereignis (in grau) wird aus dem rb-Tree entfernt und in die Liste der noch auszuführenden Ereignisse eingefügt (schwarz)
- der SoftIRQ `HRTIMER` wird „geraised“; das bedeutet er wird als rechenbereit gekennzeichnet
- das zeitlich nächstliegende Ereignis wird vom rb-Tree referenziert und der Timer auf den entsprechenden Zeitpunkt programmiert

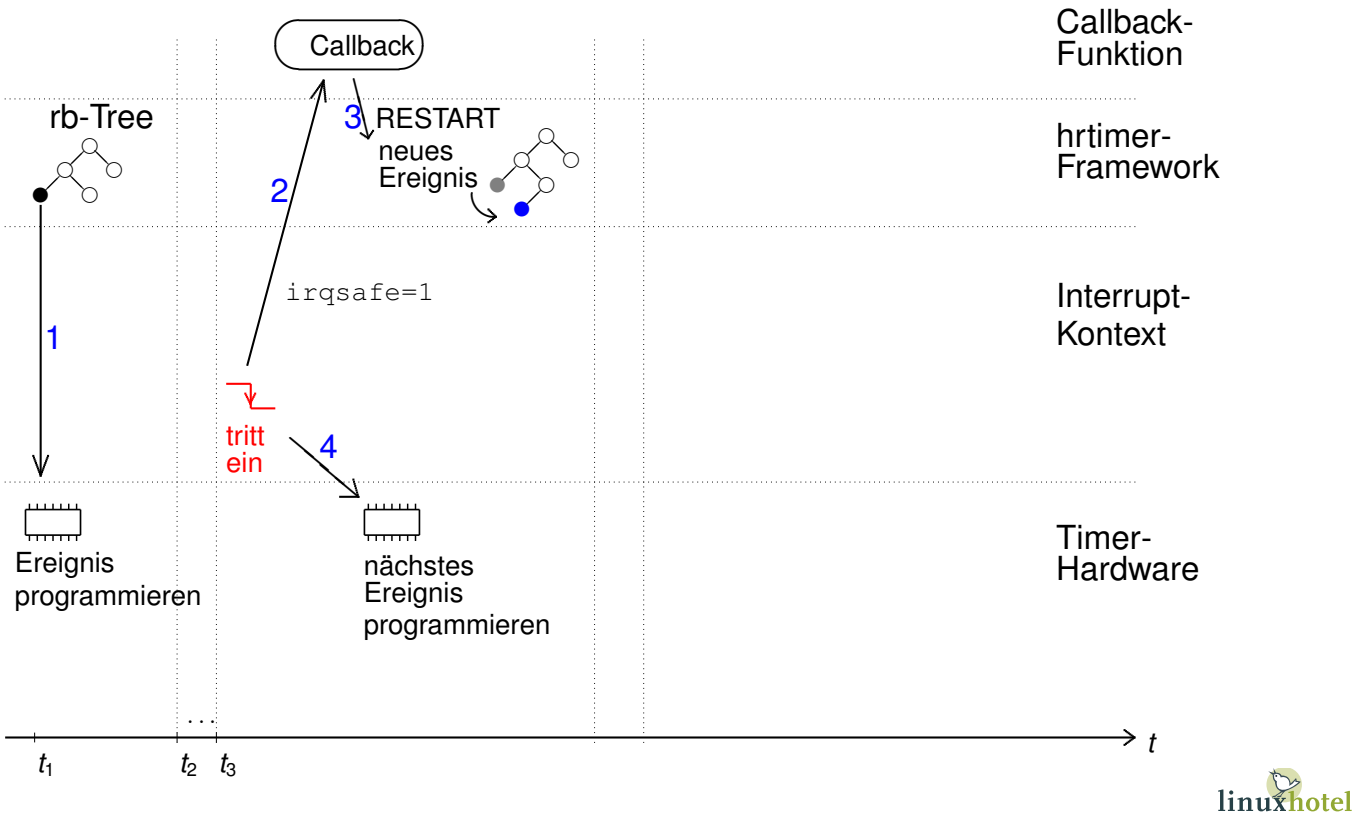
8 nachdem die ISR beendet wurde, keine weiteren Interrupts mehr anliegen und auch alle höher priorisierten oder zum Zeitpunkt des Interrupts laufenden SoftIRQ's abgearbeitet sind, wird dem geraisten SoftIRQ Rechenzeit zugeteilt

9 das SoftIRQ holt sich in der Liste der ausstehenden Ereignisse eines nach dem anderen heraus und ruft die dazu gehörende Callback-Funktion auf; im Beispiel die Funktion `mydev_callback()`

10 die Callback-Funktion weckt den schlafenden Prozess mit der Funktion `wake_up()` auf

11 (optional) hrtimer-Ereignis restartet sich selbst durch Aufruf der Funktion `hrtimer_forward()` und den Rückgabewert `HRTIMER_RESTART`; damit wird ein neues Ereignis (blau) in den rb-Tree einsortiert

direkter Callback — zeitlicher Ablauf



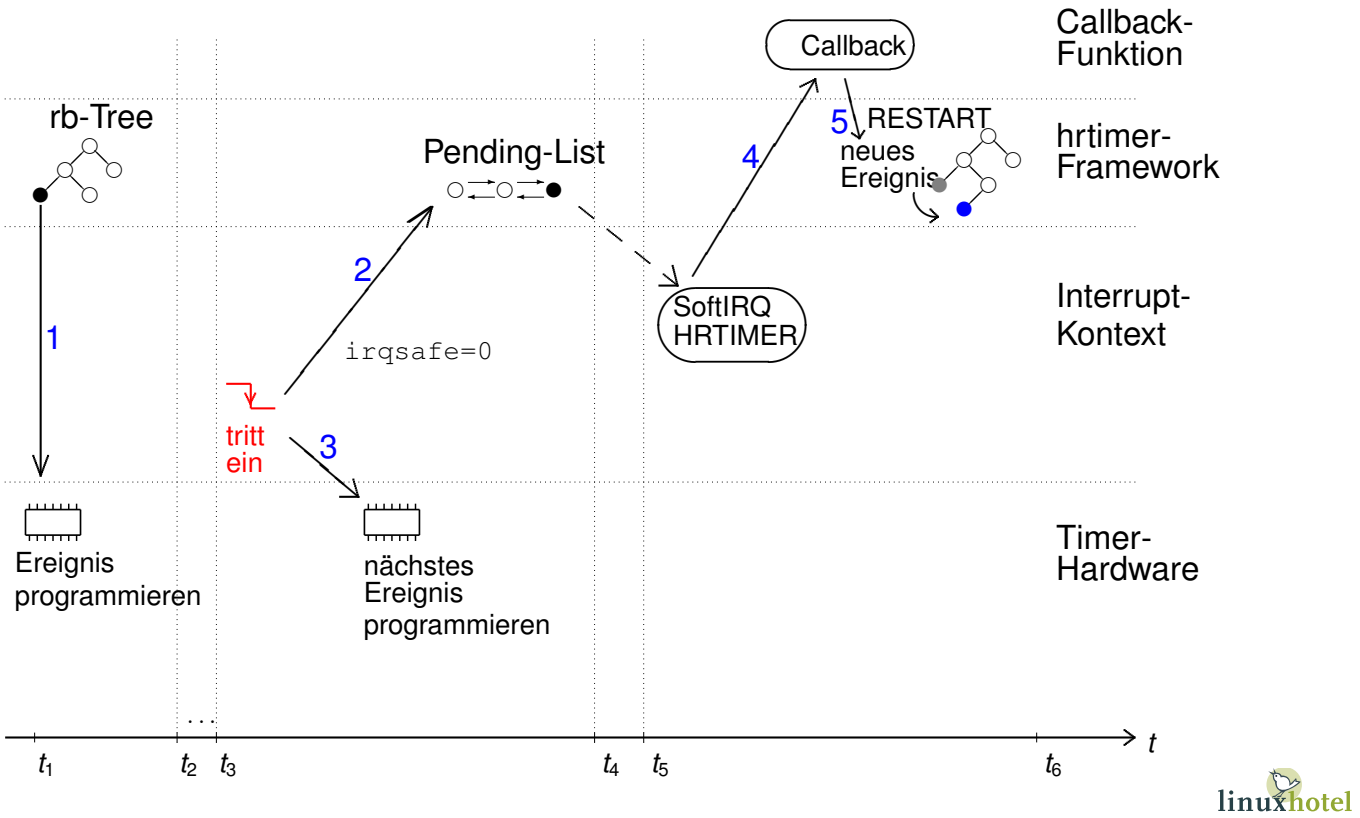
1 das programmierte Ereignis (schwarz) ist das zeitlich nächstliegende; der Hardware-Timer wird auf dessen Ablaufzeitpunkt programmiert

2 HW-Interrupt tritt ein und entfernt das Ereignis aus dem rb-Tree; im Modus `CB_IRQSAFE` (`irqsafe = 1`) wird die Callback-Funktion direkt im Interrupt-Kontext aufgerufen

3 optional kann die Callback-Funktion den Timer restartet und damit das Ereignis wieder in den rb-Tree einhängen (blau); nach dem Einhängen wird dies durch den Rückgabewert mitgeteilt

4 Timer-Interrupt programmiert den Hardware-Timer auf den nächsten Zeitpunkt

indirekter Callback — zeitlicher Ablauf



1 das programmierte Ereignis (schwarz) ist das zeitlich nächstliegende; der Hardware-Timer wird auf dessen Ablaufzeitpunkt programmiert

2 HW-Interrupt tritt ein und entfernt das Ereignis aus dem rb-Tree; im Modus `CB_SOFTIRQ` (`irqsafe = 0`) wird es in die Liste der noch austehenden Ereignisse (pending list) eingehängt (schwarz); der SoftIRQ „HRTIMER“ wird geraised, das heisst er wird als ablaufbereit markiert und kommt damit asynchron nach der Interrupt-Behandlung unter Berücksichtigung weiterer SoftIRQ's schnellstmöglich zum rechnen.

3 Timer-Interrupt programmiert den Hardware-Timer auf den nächsten Zeitpunkt

4 SoftIRQ wird ausgeführt und holt sich aus der Pending List nacheinander alle anstehenden Ereignisse; für jedes Ereignis wird die dazugehörige Callback-Funktion aufgerufen.

5 optional kann die Callback-Funktion den Timer restartet und damit das Ereignis wieder in den rb-Tree einhängen (blau); nach dem Einhängen wird dies durch den Rückgabewert mitgeteilt

Beispiel: hrtimer im Kernel-Treiber I

```
#include <linux/ktime.h>
#include <linux/hrtimer.h>


static unsigned long long nSec = 1000000000ULL;  // ns
static struct hrtimer kt;


enum hrtimer_restart kt_fn (struct hrtimer *hrt)
{
  int ndt;
  ndt = hrtimer_forward(hrt, hrtimer_get_expires(hrt),
                              ns_to_ktime(nSec));

  return HRTIMER_RESTART;
}
```

linuxhotel

Beispiel: hrtimer im Kernel-Treiber II

```
// hrtimer einrichten und starten
hrtimer_init(&kt, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
kt.function = kt_fn;
kt.irqsafe = 0;                    // --> softirq
hrtimer_start(&kt, ns_to_ktime(nSec),
                        HRTIMER_MODE_REL);


// hrtimer abbrechen
hrtimer_cancel (&kt);


MODULE_LICENSE("GPL")
```

linuxhotel

1. Funktion `enum hrtimer_restart kt_fn (struct hrtimer *hrtimer)` ist die Callback-Funktion des Timers
   bei Aufruf der Funktion wird Timer erneut gestartet und zwar bezogen auf den theoretischen Ablaufzeitpunkt (`hrtimer->_expires`)
   $\Rightarrow$ konstante Anzahl an Timer-Aufrufen pro Zeiteinheit
   Rückgabewert `HRTIMER_RESTART` bewirkt das erneute Starten des Timers

2. das Einrichten des Timer-Events erfolgt mittels der Funktion `hrtimer_init()`
   im Beispiel wird die Zeitbasis MONOTONIC gewählt und die Angabe von Zeiten erfolgt relativ
   die Struktur `kt` vom Datentyp `struct hrtimer` wird mit der Callback-Funktion `kt_fn()` sowie der Angabe indirekt aufgerufen (`kt.irqsafe = 0`) initialisiert
   mit `hrtimer_start()` wird das Timer-Event in den rb-Tree hinzugefügt und aktiviert
   die angegebenen Zeiten verstehen sich als Nanosekunden und umgewandelt in den Datentyp `ktime`

3. Timer können mit `hrtimer_cancel()` abgebrochen werden, sofern diese noch nicht ausgeführt wurden

hrtimer-Diagnose I

Shell

```
cat /proc/timer_list
```

- liefert verfügbare und verwendete Timer
- minimale Auflösung des Timers
  $\rightarrow$ `.resolution`
- aktuellen jiffies-Wert
  $\rightarrow$ `jiffies`
- Flag, ob periodische Ticks abgeschaltet sind
  $\rightarrow$ `.nohz_mode`
- Flag, ob High-Resolution aktiviert
  $\rightarrow$ `.hres_active`

linuxhotel

hrtimer-Diagnose II

- rb-Tree der aktiven Timer als chronologische Liste:
  - Ablaufzeitpunkt
  - Callback-Funktion
  - `pid`, Prozessname des Wartenden

linuxhotel

Kernel: periodische $\Leftrightarrow$ dynamische Ticks

- Zeitfunktionen basierend auf periodischem Timer oder auf One-Shot-Timer
  `Processor type and features`
  $\rightarrow$ `Tickless System (Dynamic Ticks)`
  `CONFIG_NO_HZ`

- Timer-Frequenz bei periodischen Ticks
  `Processor type and features`
  $\rightarrow$ `Timer frequency`
  `CONFIG_HZ_100`, `CONFIG_HZ_250`, `CONFIG_HZ_1000`

linuxhotel

Kernel: Low-Resolution $\Leftrightarrow$ High-Resolution

- zeitliche Auflösung in jiffies oder in Nanosekunden-Granularität

```
Processor type and features
→ High Resolution Timer Support
CONFIG_HIGH_RES_TIMERS

Processor type and features
→ HPET Timer Support
CONFIG_HPET_TIMERS
```

linuxhotel

Inhaltsverzeichnis I

Gerätetreiber

linuxhotel

Inhaltsverzeichnis II

Kernel-Architektur

linuxhotel

Inhaltsverzeichnis III

Inhaltsverzeichnis IV

Timer