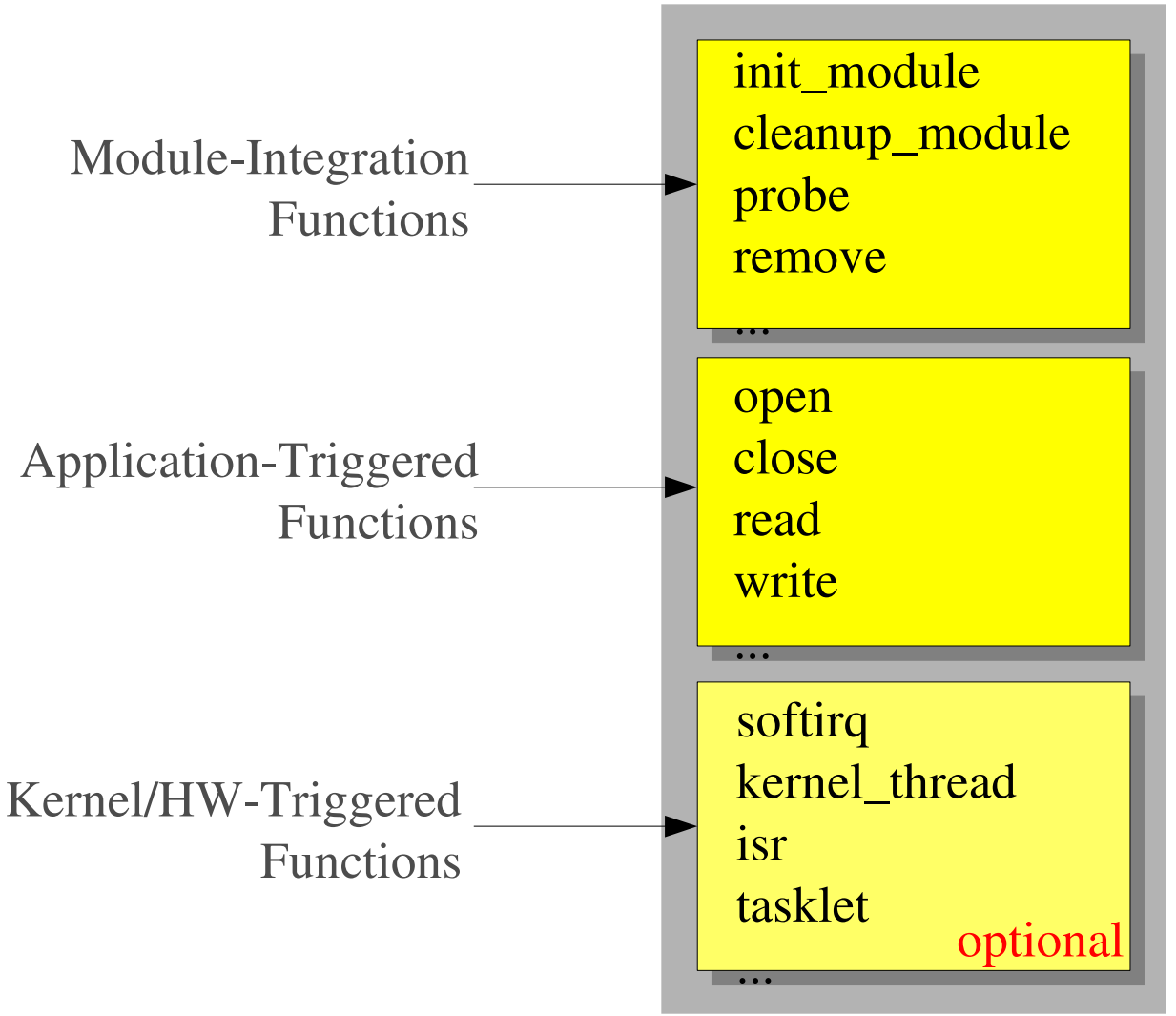
Inhalt

Asynchrone Treiberfunktionen

Infos:

- Quade/Kunst: Linux-Treiber entwickeln. 3. Auflage, dpunkt-Verlag, 2006, S. 143-166.

Funktionen eines Gerätetreibers

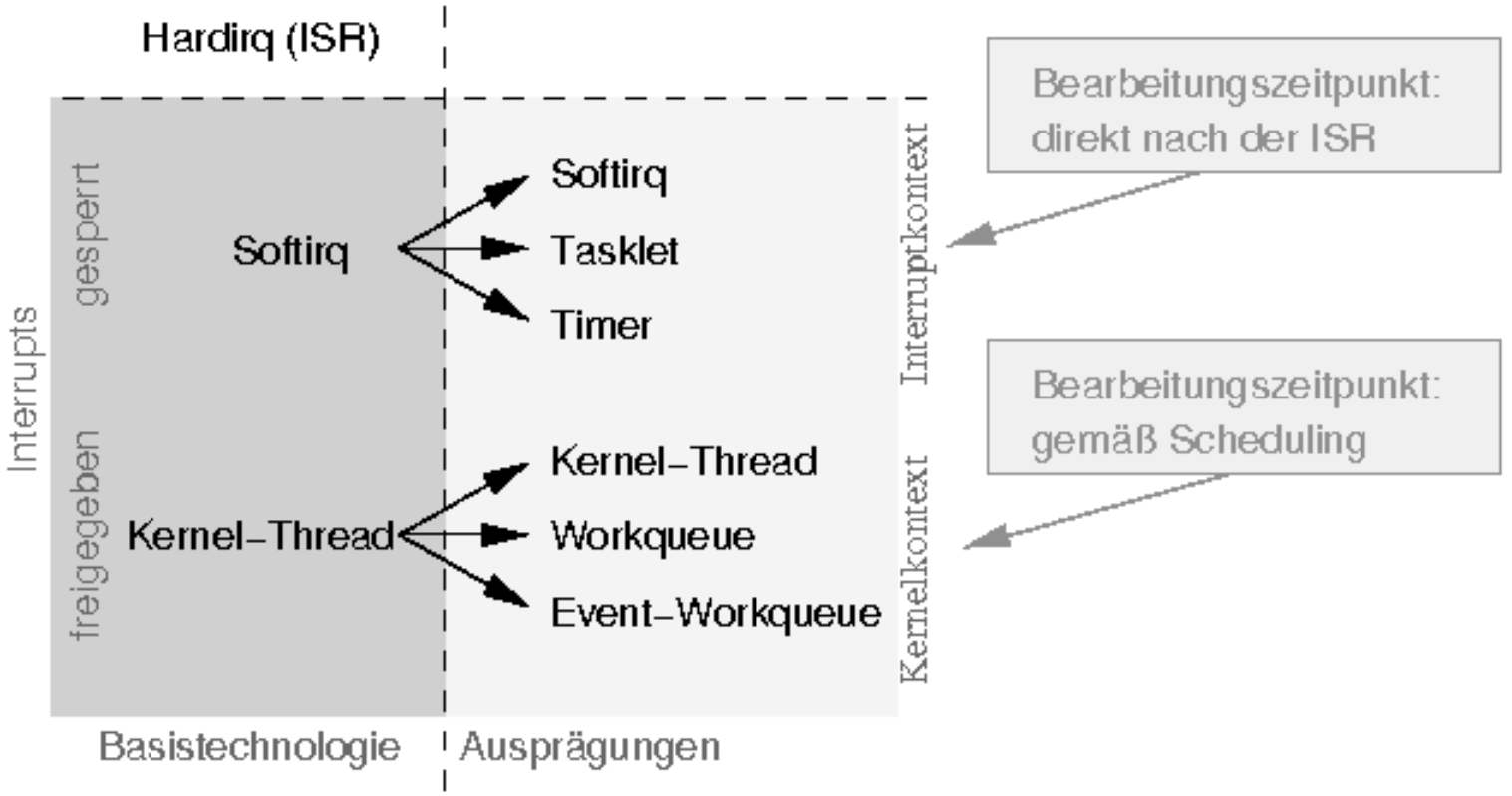


Übersicht

- Bisherige Funktionen wurden durch die Applikation „getriggert“.
- Jetzt: Im Treiber können Funktionen unabhängig von einer Applikation aufgerufen werden:
  - Durch Interrupt getriggert.
  - Zeitgesteuert.

Definitionen

Asynchrone Treiberfunktionen



Definitionen

- Kontext: Die „Umgebung“ gibt an, auf welche Dienste und Ressourcen ein Codefragment Zugriff hat.
- Interrupt-Kontext: Limitierte Funktionsauswahl, insbesondere kein „Schlafen legen“ und kein Zugriff auf Applikations-Speicherbereiche.
  - ISRs, Soft-IRQs, Tasklets und Timer.
  - GFP_ATOMIC

Definitionen

- Kernel-Kontext: Zugriff auf sämtliche Funktionen des Kernels, inklusive „Schlafen legen“.
  - Kernel-Threads, Workqueues, Event-Workqueue
  - GFP_KERNEL
- Prozess-Kontext: Wie Kernel-Kontext, zusätzlich noch Zugriff auf die Datenbereiche des gerade aktiven Rechenprozesses.
  - Applikationsgetriggerte Treiberfunktionen, Systemcalls
  - GFP_USER

Hardware Interrupt Service Routinen

- (Typischerweise) nicht unterbrechbar.

```
static irqreturn_t driver_isr( int irq, void *dev_id,
    struct pt_regs *regs )
{
    return IRQ_HANDLED;
}
```

ISR

```
static int device_init(struct pci_dev *dev,
    const struct pci_device_id *id)
{
    ...
    if(request_irq(dev->irq,driver_isr,SA_INTERRUPT|SA_SHIRQ,
        "pci_drv",dev)) {
        goto cleanup_mem;
    }
    ...
```

Anmelden der ISR

Hardware ISR (Kernel 2.6.19)

- (Typischerweise) nicht unterbrechbar.

```
static irqreturn_t driver_isr( int irq, void *dev_id,
    struct pt_regs *regs )
{
    return IRQ_HANDLED;
}
```

ISR

```
static int device_init(struct pci_dev *dev,
    const struct pci_device_id *id)
{
    ...
    if(request_irq(dev->irq,driver_isr,
         IRQF_DISABLED|IRQF_SHARED, "pci_drv",dev)) {
       goto cleanup_mem;
    }
    ...
```

Anmelden der ISR

Soft-IRQs

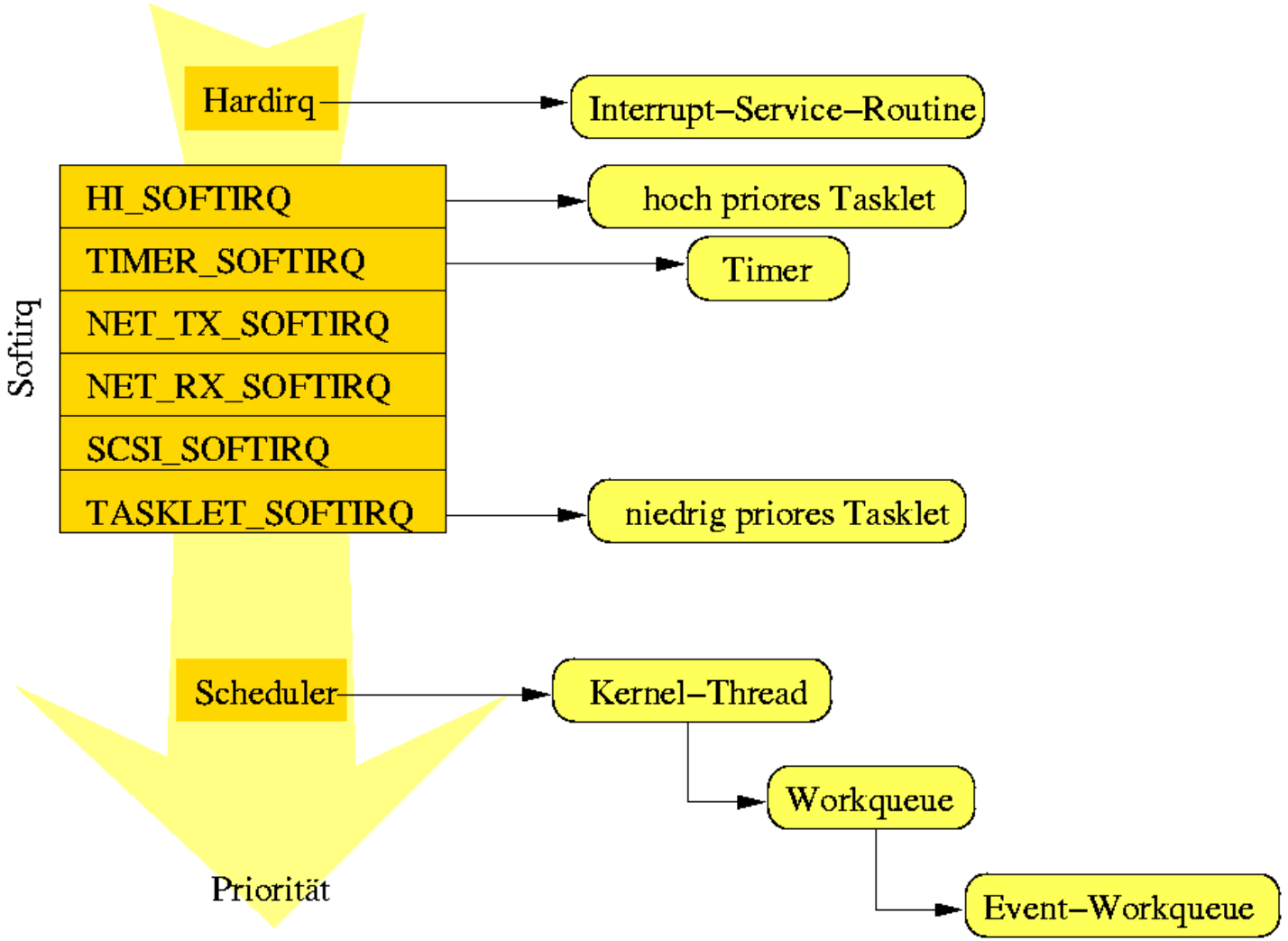
- Werden im Interrupt-Kontext abgearbeitet:
  - Kein Schlafen-Legen: wait_event()...
  - Kein Zugriff auf den User-Space: copy_to_user()...
- Abarbeitungszeitpunkt: direkt nach einer ISR.
- Werden durch Interrupts unterbrochen.
- Insgesamt stehen 32 Soft-IRQs zur Verfügung, die bei spezifischen Interrupts vom Kernel (nacheinander) bearbeitet werden. (Ein Soft-IRQ wird von der ISR aktiviert.)

Soft-IRQs

- 6 Soft-IRQs sind vordefiniert. Dort lassen sich Funktionen einhängen (schedulen), die bei Aufruf des Soft-IRQs abgearbeitet werden.
- Die übrigen Soft-IRQs sollen nicht verwendet werden.
- Wesentliche Soft-IRQs:
  - Tasklet: Wird mit dem nächsten Interrupt abgearbeitet.
  - Timer: Wird zum angegebenen Zeitpunkt abgearbeitet.

Soft-IRQs

Priorisierung



Tasklets

- Funktion, die im Kernel „so bald als möglich“ ausgeführt wird.
- Ein Tasklet wird maximal einmal aufgerufen.
- Einsatz:
  - Hochpriore Funktion
  - Interrupts sind freigegeben
- Race-Condition:
  - Falls Tasklet-Code vor dem Aufruf entladen wird.

Tasklet

```
#include <linux/module.h>
#include <linux/init.h>

static void tasklet_function( unsigned long data )
{
    printk("Tasklet called...\n");
    return;
}

DECLARE_TASKLET( tldescr, tasklet_function, 0L );

static int __init mod_init(void)
{
    printk("mod_init called\n");
    tasklet_schedule( &tldescr );
    return 0;
}

static void __exit mod_exit(void)
{
    printk("mod_exit called\n");
    tasklet_kill( &tldescr );
}

module_init( mod_init );
module_exit( mod_exit );
```

Tasklet-Funktion

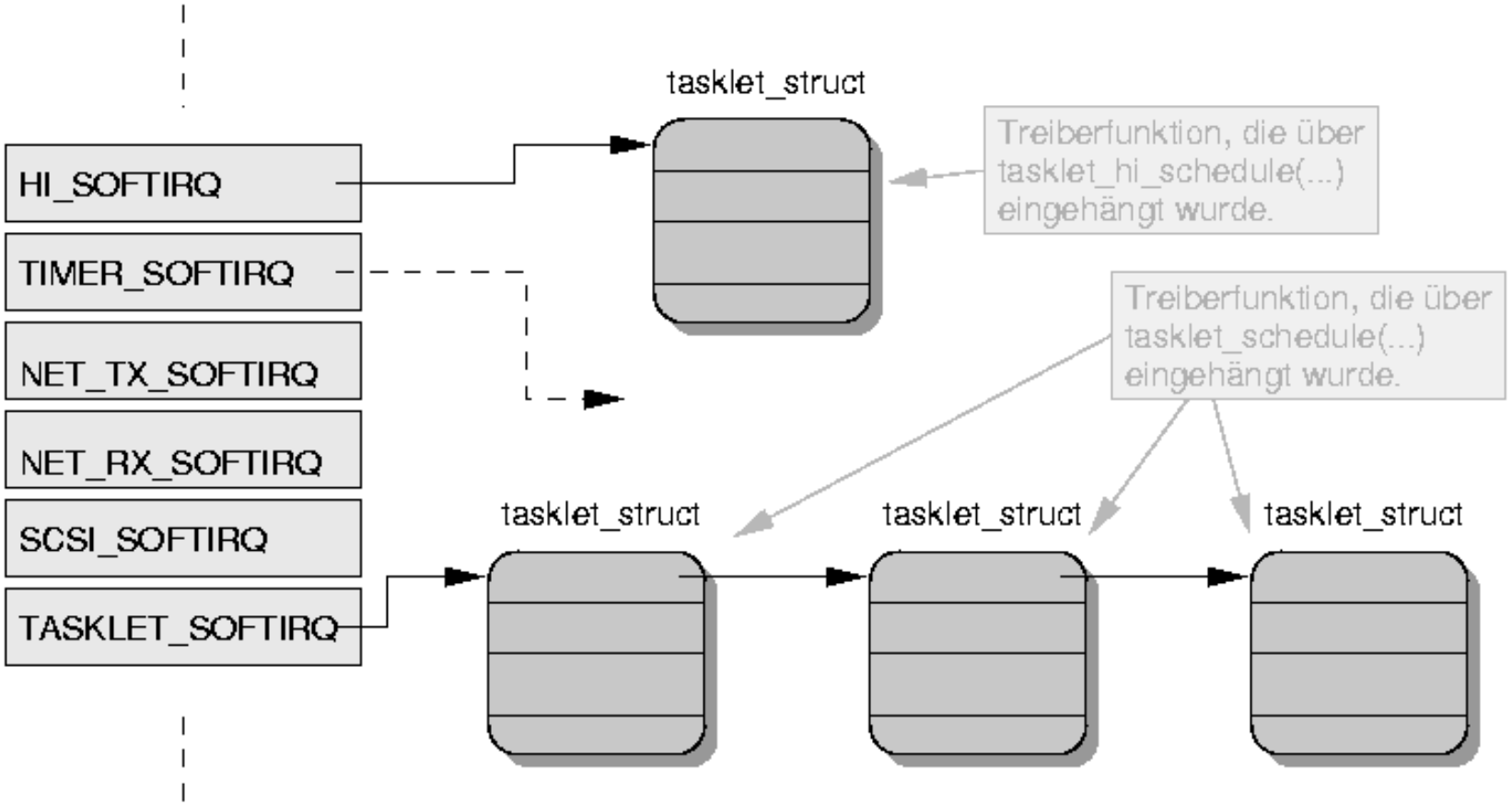
Tasklet anlegen

Das Tasklet wird nach Abarbeitung des nächsten Interrupts aufgerufen.

Tasklet entfernen

Soft-IRQs

Tasklets



Tasklet-Funktionen

- `DECLARE_TASKLET( struct tasklet_struct name, void (*func)(unsigned long), unsigned long data );`
  - Statische Definition.
- `void tasklet_init( struct tasklet_struct t, void (*func)(unsigned long), unsigned long data );`
  - Initialisierung zur Laufzeit.
- `void tasklet_schedule( struct tasklet_struct *t );`
  - Normales Tasklet zur Abarbeitung freigeben.
- `void tasklet_hi_schedule( struct tasklet_struct *t );`
  - Hochpriores Tasklet zur Abarbeitung freigeben.
- `void tasklet_kill( struct tasklet_struct *t );`
  - Freigegebenes Tasklet beenden.

Übung 1.11: Tasklet

- Erstellen Sie selbst ein Modul, das ein Tasklet realisiert (Name `tasklet.c`). Das Tasklet soll Ausgaben ins Syslog tätigen. Es kann direkt in der Modul-Initialisierungsfunktion gescheduled werden.
- Generieren und Testen Sie Ihr Modul.

Timer

- Aufruf einer Funktion zu einem späteren Zeitpunkt.
- Zeiten werden absolut auf Basis von Jiffies angegeben.
- Timer dürfen zu einem Zeitpunkt maximal einmal eingehängt werden.
- Die Timer-Funktion wird im Interruptkontext abgearbeitet (Interrupts sind freigegeben).
- Race-Condition:
  - Falls der Timer-Code vor dem Aufruf entladen wird...

Timer

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

static struct timer_list mytimer;

static void inc_count(unsigned long arg)
{
    printk("inc_count called (%ld)...\n", mytimer.expires );
}

static int __init ktimer_init(void)
{
    init_timer( &mytimer );
    mytimer.function = inc_count;
    mytimer.data = 0;
    mytimer.expires = jiffies + (2*HZ); // 2 second
    add_timer( &mytimer );
    return 0;
}

static void __exit ktimer_exit(void)
{
    if( del_timer_sync( &mytimer ) )
        printk("Aktiver Timer deaktiviert\n");
}
```

Timer Objekt

Initialisierung

Instanzierung

Aufräumen

Timer-Funktionen

- `init_timer`
- `add_timer`
- `mod_timer`
- `del_timer_sync`
- `timer_pending`

Übung 12: Timer

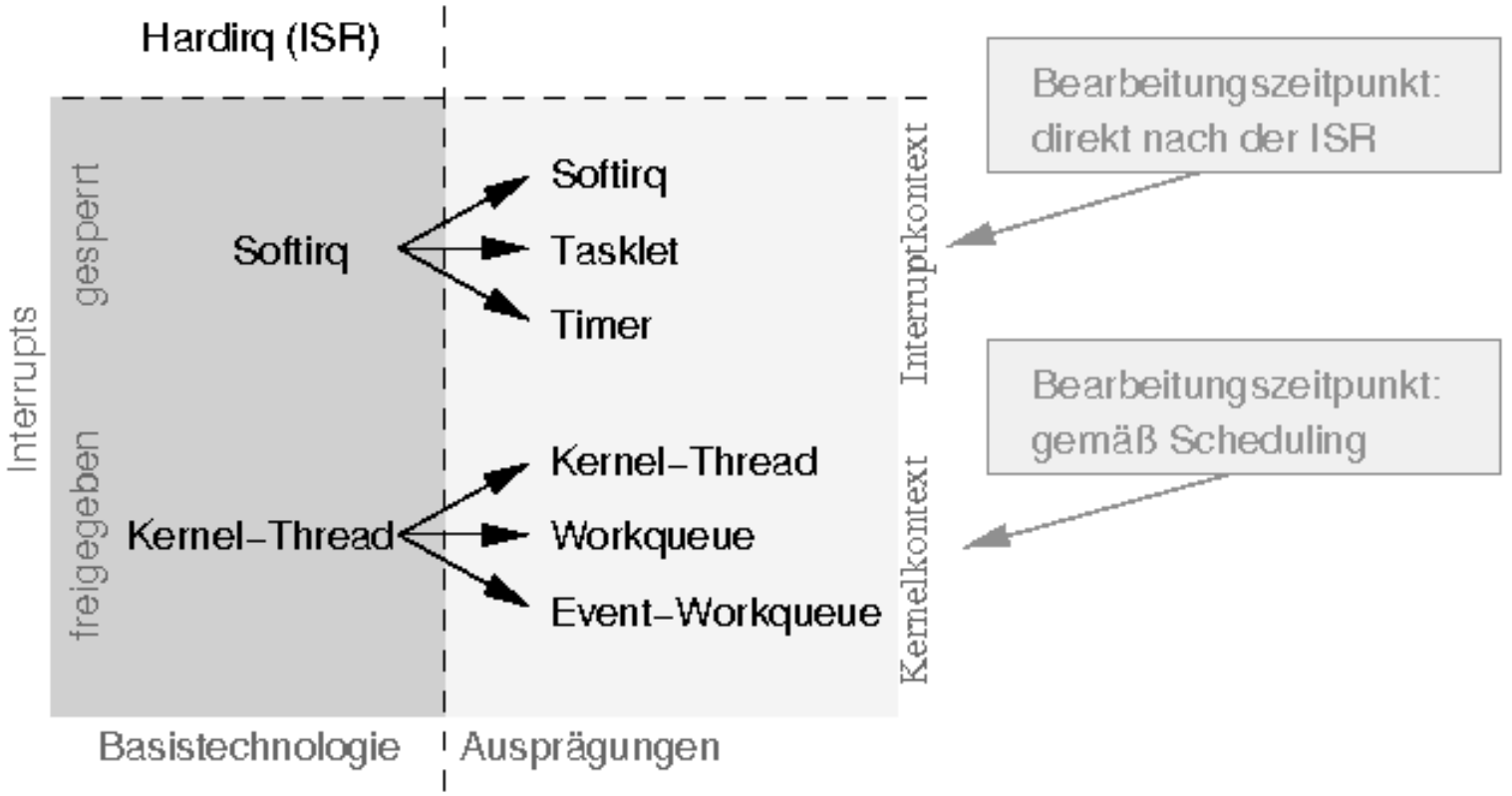
- Innerhalb des Kernels soll eine von Ihnen realisierte Funktion regelmäßig, alle 2 Sekunden eine Ausgabe ins Syslog machen (Name `mytimer.c`).
- Generieren und testen Sie Ihren Code.

Zusammenfassung „Interrupts“

- Über Soft-IRQs lassen sich hochprior Funktionen abarbeiten.
- Der Soft-IRQ „Tasklet“ wird nach dem nächsten Interrupt abgearbeitet. Hochpriore Tasklets werden direkt nach einem Interrupt abgearbeitet.
- Der Soft-IRQ „Timer“ ermöglicht die zeitgesteuerte Abarbeitung einer hochprioren Funktion.
- Andere Soft-IRQs werden beispielsweise vom Netzwerk-Subsystem oder vom SCSI-Subsystem eingesetzt.

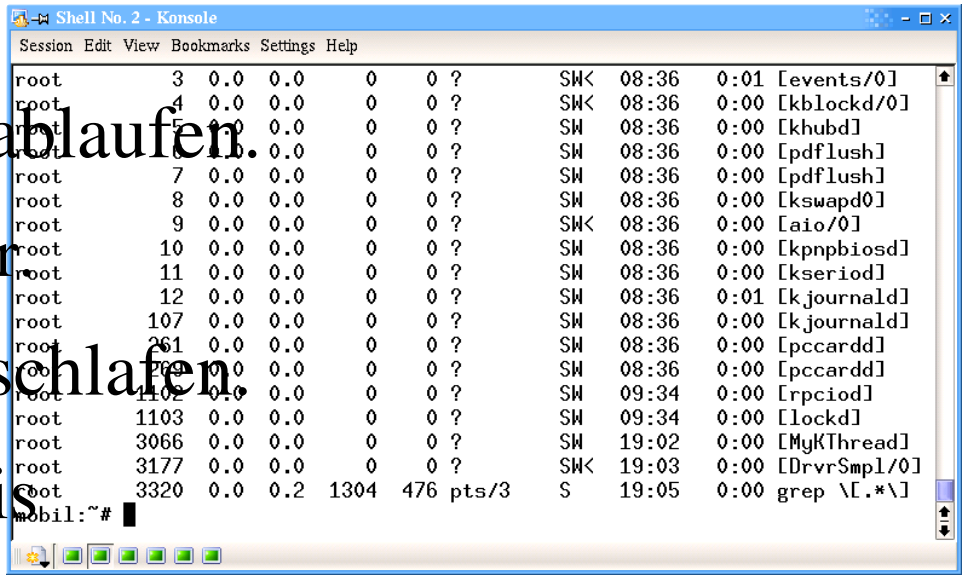
Definitionen

Asynchrone Treiberfunktionen

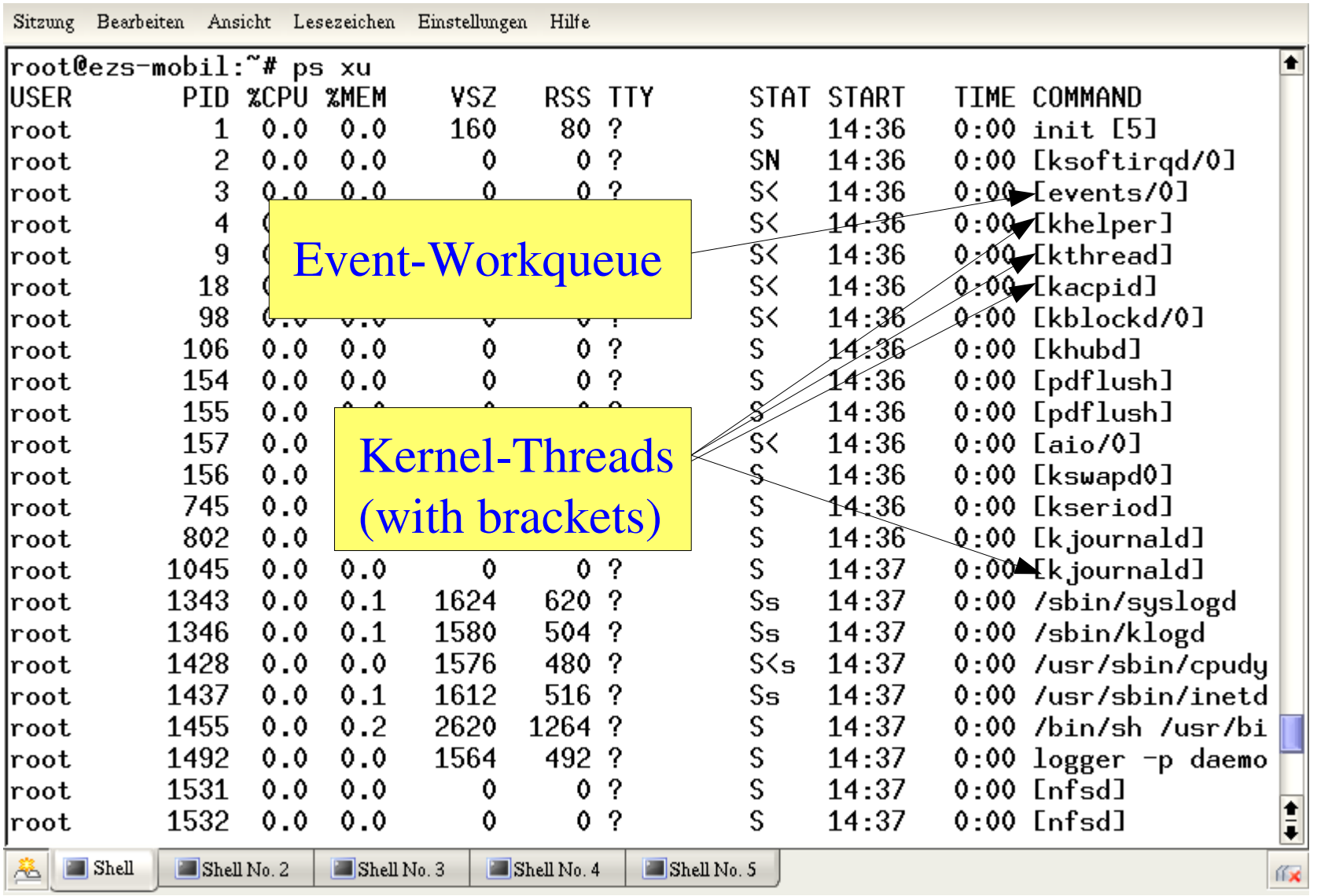


Kernel-Threads

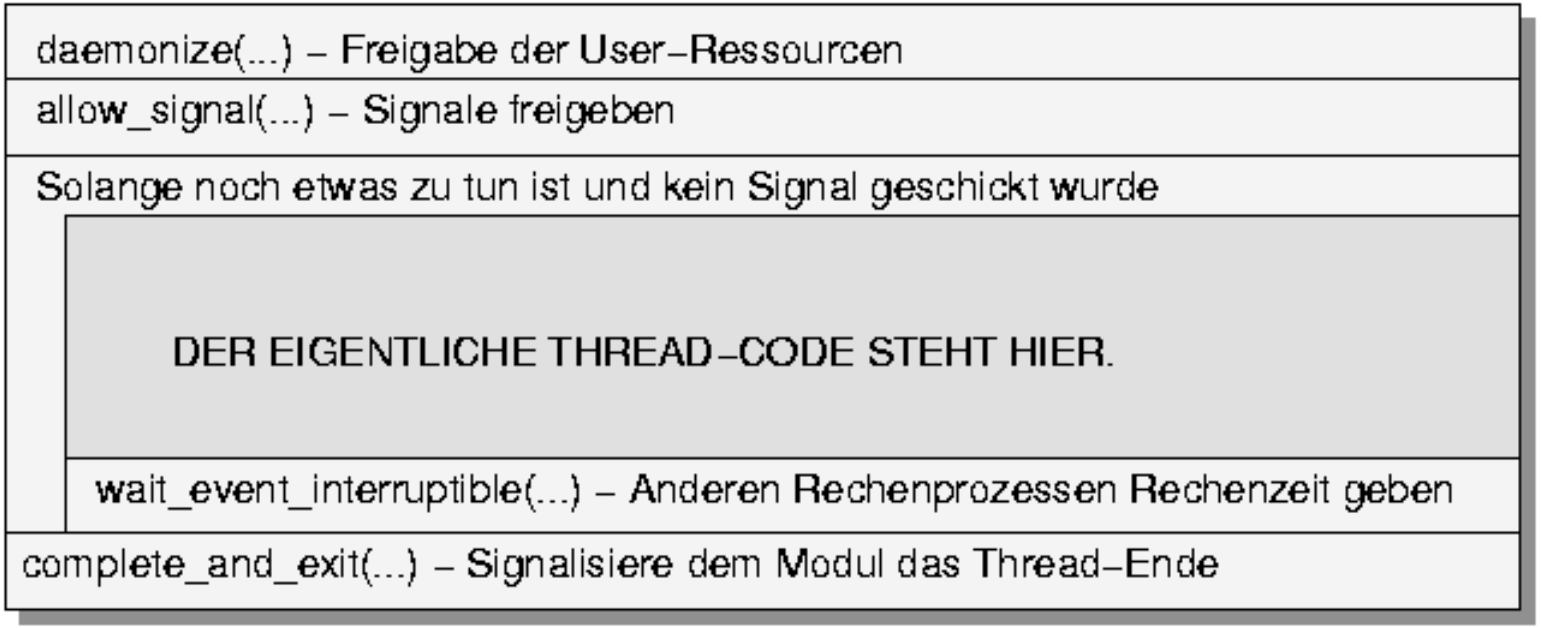
- Prozesse, die im Kernel ablaufen.
  - In der Taskliste sichtbar
- Kernel-Threads können schlafen.
- Kernel-Threads laufen bis
  - sie sich beenden
  - sie sich schlafen legen
- Race Condition:
  - Falls der Code eines aktiven Threads entladen wird.



Kernel-Threads



Kernel-Thread



Kernel-Threads und Signale

```
if( !signale_pending( current ) ) {
   ... // ein Signal liegt vor
} else {
   ... // kein Signal
}
```

```
...
kill_pid(find_pid_ns(thread_id,&init_pid_ns),SIGTERM,1);
...
```

Kernel-Thread - Code-Beispiel

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

static int thread_id=0;
static wait_queue_head_t wq;
static DECLARE_COMPLETION( on_exit );

...

static int __init kthread_init(void)
{
    init_waitqueue_head(&wq);
    thread_id=kernel_thread(thread_code, NULL, CLONE_KERNEL );
    if( thread_id==0 )
        return -EIO;
    return 0;
}

static void __exit kthread_exit(void)
{
    if( thread_id )
        kill_pid(find_pid_ns(thread_id,&init_pid_ns), SIGTERM, 1 );
    wait_for_completion( &on_exit );
}
```

Objekte, die für das Thread-Management (erzeugen, schlafen-legen, entfernen) notwendig sind.

Erzeugen des Threads.

Beenden des Threads.

Kernel-Thread - Code-Beispiel

```
static int thread_code( void *data )
{
    unsigned long timeout, retval;
    int i;

    daemonize("MyKThread");
    allow_signal( SIGTERM );
    ...
    while( retval!=-ERESTARTSYS ) {
        ...
        retval=
        wait_event_interruptible_timeout(wq, (timeout==0), timeout);
    }
    thread_id = 0;
    complete_and_exit( &on_exit, 0 );
}
```

Einstieg

Eigentliche Threadfunktionalität

Ausstieg

Übung 13: Kernel-Threads

- Schreiben Sie einen Kernel-Thread, der sich für jeweils 2 Sekunden schlafen legt und – jedes Mal wenn er geweckt wird - eine Ausgabe ins Syslog macht (Name `kthread.c`).
- Generieren und Laden Sie Ihren Code.
- Schicken Sie Ihrem Thread über die Konsole ein Signal.
- Beobachten Sie vor- und nachher die Taskliste.

Workqueue

- Das Management von zyklischen/periodischen Kernel-Threads ist kompliziert:
  - Anlegen des Workqueue-Objektes
  - Schlafen-Legen
  - ...
- Workqueues sind „vorgefertigte“ Kernel-Threads, in deren Kontext Funktionen (einmal) abgearbeitet werden.

Workqueue

- Zwei Objekt-Klassen sind notwendig:
  - Workqueue-Objekt: repräsentiert den Kernel-Thread
  - Work-Objekt: repräsentiert die abzuarbeitende Funktion.
- Eine dem Workqueue-Objekt übergebene Funktion (Work-Objekt) wird einmal abgearbeitet.
- Falls gewünscht wird pro CPU wird ein Workqueue-Thread angelegt:
  - create_workqueue(), create_singlethread_workqueue();
- Race-Condition:
  - Falls der Workqueue-Code entladen wird ...

Workqueue: Beteiligte Funktionen

• create_workqueue	→ Kein IR-Kontext!
• flush_workqueue	→ Abarbeitung erzwingen.
• destroy_workqueue	→ Freigeben des Objektes.
• DECLARE_WORK	→ Statische Definition.
• DECLARE_DELAYED_WORK	→ Statische Definition.
• INIT_WORK	→ Initialisierung zur Laufzeit.
• queue_work	→ Zur Abarbeitung freigeben.
• schedule_work	→ Für die Event-Workqueue.
• schedule_delayed_work	→ Für die Event-Workqueue.
• flush_scheduled_work	→ Abarbeitung in der E-Wq erzwingen.

```
static struct workqueue_struct *wq;

static void work_queue_func( void *data )
{
    pr_debug( "work_queue_func...\n" );
    return;
}

static DECLARE_WORK( work_obj, work_queue_func, NULL );

static int __init drv_init(void)
{
    wq = create_singlethread_workqueue( "DrvrSmpl" );
    if( queue_work( wq, &work_obj ) )
        pr_debug( "queue_work successful ...\n");
    else
        pr_debug( "queue_work not successful ...\n");
    return 0;
}

static void __exit drv_exit(void)
{
    pr_debug("drv_exit called\n");
    if( wq )
        destroy_workqueue( wq );
}
```

Event-Workqueue

- Für einfache, wenig zeitkritische Aktionen gibt es einen noch einfacheren Mechanismus, eine Funktion im Prozesskontext abarbeiten zu lassen:
  - Für jede CPU instanziert der Kernel eine Workqueue (Event-Workqueue genannt).
  - Der Event-Workqueue kann direkt das Work-Objekt (mit der Funktionsadresse) übergeben werden.
- Funktionen:
  - schedule_work
  - schedule_delayed_work
  - flush_scheduled_work

Event-Workqueue: Code-Beispiel

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

static void work_queue_function( void *data )
{
    pr_debug( "work_queue_function()\n“ );
    return;
}

static DECLARE_WORK(work, work_queue_function, NULL );

static int __init mod_init(void)
{
    if( schedule_work( &work )==0 )
        return -EFAULT;
    return 0;
}

static void __exit mod_exit(void)
{
    flush_scheduled_work();
}
```

Zusammenfassung

- Der Kern stellt folgende Konstrukte zur Verfügung, mit denen - unabhängig von einer Applikation - Code im Kernel ausgeführt wird:
  - Tasklets
  - Timer
  - Kernel-Threads
  - Workqueues
  - Event-Workqueue
- Problem: Es muss sichergestellt werden, dass vor dem Entladen eines Moduls keine Objekte des Moduls aktiv sind.

Übung 14: Workqueues

- Implementieren Sie ein Modul, welches beim Laden eine Workqueue erzeugt und dieser eine Funktion übergibt, die 4 Sekunden später aufgerufen wird. Es reicht, wenn diese Funktion eine Ausgabe im Syslog macht (Name `mywq.c`).
- Generieren und Laden Sie Ihren Code.
- Funktioniert Ihr Modul auch fehlerfrei, wenn Sie das Modul nach dem Laden direkt wieder entladen?