Systemsoftware

Linux Gerätetreiber II

Prof. Dr. Michael Mächtel

Informatik, HTWG Konstanz

Version vom 07.05.17

Übersicht

1. Soft-IRQ's und Kernel-Threads
2. Schutz kritischer Bereiche
3. Hardware Anbindung

Übersicht

1. Soft-IRQ's und Kernel-Threads
2. Schutz kritischer Bereiche
3. Hardware Anbindung

Funktionen eines Kernel-Moduls

Funktionen zur Einbindung in das Betriebssystem

- init_module
- cleanup_module
- probe
- remove

nutzt

- register_chrdev
- request_region

Funktionen, die durch die Applikation getriggert werden.

- open
- close
- read
- write

Funktionen, die durch das Betriebssystem oder die HW getriggert werden.

- ISRs
- Soft-IRQ's
- Timer
- Kernel-Threads

Übersicht

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

Soft-IRQ: Asynchrone Treiberfunktionen

Unterbrechungsmodell

Definition Kontext

- Kontext: Die „Umgebung“ gibt an, auf welche Dienste und Ressourcen ein Codefragment Zugriff hat.

Übersicht Linux Kontexte

- *Interrupt-Kontext*: Limitierte Funktionsauswahl, insbesondere kein „Schlafen legen“ und kein Zugriff auf Applikations-Speicherbereiche.
  - ISRs, Soft-IRQs, Tasklets und Timer.
  - GFP_ATOMIC
- *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 ISR

Typischerweise nicht unterbrechbar !

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

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;
}
...
```

ISR

Anmelden der ISR

Soft-IRQ's

- 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.)
  - 10 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.

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.

Beispielcode für 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

Tasklet Priorisierung

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.

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…

Beispielcode für 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()*

Zusammenfassung IRQ's

- 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.

Kernel-Threads: Asynchrone Treiberfunktionen

Definition: KThread

- Kernel-Thread = Thread, der im Kernel abgearbeitet wird und typischerweise keine Ressourcen im Userland besitzt.
  - POSIX User-Thread ‘Style’
- Ist durch einen eigenen PCB definiert.
- „Steuerung“ des Threads über Signals.
- Race Condition:
  - Thread muss vor dem Entfernen des zugehörigen Codes beendet sein.

Übersicht: KThreads

- 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.

KThreads in Taskliste

Sitzung Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
root@ezs-mobil:~# ps xu
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0    160    80 ?        S    14:36   0:00 init [5]
root         2  0.0  0.0      0     0 ?        SN   14:36   0:00 [ksoftirqd/0]
root         3  0.0  0.0      0     0 ?        S<   14:36   0:00 [events/0]
root         4  0.0  0.0      0     0 ?        S<   14:36   0:00 [khelper]
root         9  0.0  0.0      0     0 ?        S<   14:36   0:00 [kthread]
root        18  0.0  0.0      0     0 ?        S<   14:36   0:00 [kacpid]
root        98  0.0  0.0      0     0 ?        S<   14:36   0:00 [kblockd/0]
root       106  0.0  0.0      0     0 ?        S    14:36   0:00 [khubd]
root       154  0.0  0.0      0     0 ?        S    14:36   0:00 [pdflush]
root       155  0.0  0.0      0     0 ?        S    14:36   0:00 [pdflush]
root       157  0.0  0.0      0     0 ?        S<   14:36   0:00 [aio/0]
root       156  0.0  0.0      0     0 ?        S    14:36   0:00 [kswapd0]
root       745  0.0  0.0      0     0 ?        S    14:36   0:00 [kseriod]
root       802  0.0  0.0      0     0 ?        S    14:36   0:00 [kjournald]
root      1045  0.0  0.0      0     0 ?        S    14:37   0:00 [kjournald]
root      1343  0.0  0.1   1624   620 ?        Ss   14:37   0:00 /sbin/syslogd
root      1346  0.0  0.1   1580   504 ?        Ss   14:37   0:00 /sbin/klogd
root      1428  0.0  0.0   1576   480 ?        S<s  14:37   0:00 /usr/sbin/cpudy
root      1437  0.0  0.1   1612   516 ?        Ss   14:37   0:00 /usr/sbin/inetd
root      1455  0.0  0.2   2620  1264 ?        S    14:37   0:00 /bin/sh /usr/bi
root      1492  0.0  0.0   1564   492 ?        S    14:37   0:00 logger -p daemo
root      1531  0.0  0.0      0     0 ?        S    14:37   0:00 [nfsd]
root      1532  0.0  0.0      0     0 ?        S    14:37   0:00 [nfsd]
```

Shell | Shell No. 2 | Shell No. 3 | Shell No. 4 | Shell No. 5

Body des Kernel-Thread Code

Kernel-Thread erzeugen

- Zum Erzeugen von Kernel-Threads stellt der Kernel die Funktion *kernel_thread()* zur Verfügung.
- Kernel-Threads mit Hilfe dieser Funktion zu erzeugen ist fehlerträchtig, da ein neu erzeugter Thread ein „zufälliges Erbgut“ (neuer Thread == Kindprozess, wer ist der Elternprozess?) mitbekommt. Einfacher:
  - Kthread-Daemon:
    - Der Kthread-Daemon ist selbst ein Kernel-Thread, in dessen Kontext der Kind-Thread erzeugt wird.
    - Eindeutiger Kontext.

Beispiel: KThread erzeugen/beenden

```
...
static int __init kthread_init(void)
{
        init_waitqueue_head(&wq);
        thread_id=kthread_create(thread_code,NULL,"mykthread");
        if( thread_id==0 )
                return -EIO;
        wake_up_process( thread_id );
        return 0;
}

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

Beispiel: KThread Code

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

static struct task_struct *thread_id;
static wait_queue_head_t wq;
static DECLARE_COMPLETION( on_exit );

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

        allow_signal( SIGTERM );
        for( i=0; i<5 && (kthread_should_stop()==0); i++ ) {
                timeout=HZ; /* wait 1 second */
                timeout=wait_event_interruptible_timeout(
                        wq, (timeout==0), timeout);
                printk("thread_function: woke up ... %ld\n", timeout);
                if( timeout==-ERESTARTSYS ) {
                        printk("got signal, break\n");
                        break;
                }
        }
        complete_and_exit( &on_exit, 0 );
}
```

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

Eigentliche Thread Funktionalität

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.

Objekt-Klassen für 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 ein Workqueue-Thread angelegt:
  - *create_workqueue()*, *create_singlethread_workqueue()*
- Race-Condition:
  - Falls der Workqueue-Code entladen wird ...

Workqueue Funktionen 1

- *create_workqueue*
  - Kein IR-Kontext!
- *flush_workqueue*
  - Abarbeitung erzwingen.
- *destroy_workqueue*
  - Freigeben des Objektes.

Workqueue Funktionen 2

- *DECLARE_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 Event-Workqueue erzwingen.

Beispielcode für Workqueue

```
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*

Beispielcode Event Workqueue

```c
#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 KThread/Workqueue

- 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.

Übersicht

1. Soft-IRQ's und Kernel-Threads
2. Schutz kritischer Bereiche
3. Hardware Anbindung

Problemstellung

- Durch die Parallelität und Unterbrechbarkeit (Preemption) gibt es Konflikte beim Zugriff auf gemeinsame Betriebsmittel (z.B. auf Speicherbereiche) und bei der Synchronisation.
- Bereits gesehen:
  - Globale Variablen im Treiber
  - Softirqs (Tasklets, Timer)
  - Kernel-Threads (Workqueues, Event-Workqueue)

Bekannte Lösung

- Gegenseitiger Ausschluss (mutual exclusion)
- Ausprägungen im Linux-Kernel:
  - Interruptsperre
  - Atomare Operationen
  - Semaphor/Mutex
  - Spinlocks
  - Memory Barriers

Interrupt- und Preemption-Sperre

- Preemptionsperre:
  - Verhindert das Scheduling (auf dem lokalen Prozessor)
- Interruptsperre:
  - Auftretende Interrupts werden zurückgehalten.
  - Nur für Einprozessor-Systeme geeignet.
- Beide Sperren sollten generell vermieden werden.

Funktionen für Interrupt-/Preemption-Sperre

- *cli()*
- *sti()*
- *preempt_disable()*
- *preempt_enable()*

Atomare Operationen

- Mit einer atomaren Variable kann atomar auf einen Integerwert (32 oder 64 Bit) oder einen Bitwert zugegriffen werden.
- Der Zugriff auf die atomare Variable erfolgt über zugehörige Funktionen.
- Der Zugriff ist sowohl im Kernel-, Prozess- als auch im Interruptkontext möglich.

Atomare Variablen

- Die Modifikation einer Variablen vollzieht sich in 3 Schritten:
  - Variable in ein internes Register kopieren.
  - Registerinhalt modifizieren.
  - Registerinhalt zurückspeichern.
- Finden während dieser drei Schritte parallele Zugriffe auf eine Variable statt, kann es unter Umständen zu einem falschen Ergebnis kommen.
- Zum Schutz vor einer derartigen „Race Condition“ muss der Zugriff auf die Variable „atomar“, also ungeteilt stattfinden.
- Beim parallelen Zugriff auf eine (ungeschützte) Variable kann es zu falschen Ergebnissen kommen.

atomic_() Funktionen

- int atomic_read(atomic_t *v);
- void atomic_set(atomic_t *v, int i);
- void atomic_add_(int i, atomic_t *v)
- void atomic_sub(int i, atomic_t *v)
- void atomic_inc(atomic_t *v)
- void atomic_dec(atomic_t *v)
- int atomic_sub_and_test(int i, atomic_t *v)
- int atomic_inc_and_test(atomic_t *v)
- int atomic_dec_and_test(atomic_t *v)
- int atomic_add_negative(int i, atomic_t *v)v

atomic_long() Funktionen

- int atomic_long_read(atomic_long_t *l);
- void atomic_long_set(atomic_long_t *s, long i);
- void atomic_long_add_(long i, atomic_long_t *l)
- void atomic_long_sub(long i, atomic_long_t *l)
- void atomic_long_inc(atomic_long_t *l)
- void atomic_long_dec(atomic_long_t *l)

test_and_set() Funktionen

- void set_bit(int nr, unsigned long *a)
- void clear_bit(int nr, unsigned long *a)
- void change_bit(int nr, unsigned long *a)
- int test_and_set_bit(int nr, unsigned long *a)
- int test_and_clear_bit(int nr, unsigned long *a)
- int test_and_change_bit(int nr, unsigned long *a)
- int test_bit(int nr, void *a)

Mutex und Semaphore

- Zum Schutz komplexer Datenstrukturen.
- Der Zugriff ist nur im Prozesskontext möglich.
- Mutexe sind deutlich effizienter (weniger Speicher), daher sollten diese Semaphoren vorgezogen werden.

Mutex Funktionen

- DEFINE_MUTEX(name);
- mutex_init(mutex);
- void mutex_lock(struct mutex *lock);
- int mutex_lock_interruptible(struct mutex *lock);
  - Returnwert: 0 oder -EINTR
- int mutex_trylock(struct mutex *lock);
- void mutex_unlock(struct mutex *lock);
- int mutex_is_locked(struct mutex *lock);

Mutex Definition und Initialisierung

```
#include <linux/mutex.h>
...
/* Statische Definition, Initialisierung */
/* durch den Compiler */
DEFINE_MUTEX( my_mutex );

/* Initialisierung zur Laufzeit */
struct mutex my_mutex;
…
mutex_init( my_mutex );
```

Mutex Einsatz

```
mutex_lock( my_mutex );
. . . /* kritische Codesequenz */
mutex_unlock( my_mutex );
```

```
If( mutex_lock_killable( my_mutex ) ) {
        pr_info(„got signal\n“);
        return -ERESTARTSYS;
}
. . . /* kritische Codesequenz */
mutex_unlock( my_mutex );
```

Mutex Verwendungshinweise

- Freigabe nur durch den Besitzer möglich.
- Darf nicht mehrfach freigegeben werden.
- Darf nicht mehrfach (rekursiv) reserviert werden.
- Eine Task darf nicht mit einem gehaltenen Mutex beendet werden.
- Mutexe dürfen nicht im Interrupt-Kontext verwendet werden.
- Reservierte Mutexe dürfen nicht reinitialisiert werden.
- Realtime Mutexe ‘rt_mutex’ mit Priority Inheritance Protocoll

Semaphor: Anwendung und Einsatz

```
while( down_interruptible(&Semaphor) == -EINTR ) {
                                  Betreten des kritischen Abschnitts.
        ... // Behandlung des Signals
}
... // Modifikation der Daten
                                  Der kritische Abschnitt selbst.

up( &Semaphor );Verlassen des kritischen Abschnitts.
```

Semaphor: Ausprägungen

- „Normale“ Semaphore
  - DEFINE_SEMAPHORE()
  - sema_init()
  - down(), up(), down_trylock(), up_trylock()
- Schreib-/Lese-Semaphore
  - down_read
  - down_read_trylock
  - down_write
  - down_write_trylock
  - downgrade_write
  - up_read
  - up_write

Spinlock

- Synchronisationselement zum Schutz eines „kurzen“ kritischen Abschnitts.
- Realisiert ein aktives Warten.
- Im Prozess- und Interruptkontext einsetzbar!
- In der Theorie nur für Mehrprozessormaschinen geeignet.
- Praktisch auch auf Einprozessormaschinen einsetzbar.

Spinlock: Initialisierung

- statisch:
  - static spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
- dynamisch:
  - spin_lock_init(&my_lock);

Spinlock: Anwendung

- spin_lock
- spin_unlock
- spin_lock_irq
- spin_unlock_irq
- spin_lock_irqsave
- spin_unlock_irqsave
- spin_lock_bh
- spin_unlock_bh

Spinlock: Beispiel

```
#include <asm/spinlock.h>
...
unsigned long irqflags;
static struct spinlock_t
        my_lock=SPIN_LOCK_UNLOCKED;
...
spin_lock_irqsave( &my_lock, &irqflags );
... // kritischer Abschnitt
spin_unlock_irqrestore( &my_lock, &irqflags );

...
```

Spinlock: R/W Ausprägungen

- read_lock, write_lock
- read_unlock, write_unlock
- read_lock_irq, write_lock_irq
- read_unlock_irq, write_unlock_irq
- read_lock_irqsave, write_lock_irqsave
- read_unlock_irqrestore, write_unlock_irqrestore
- read_lock_bh, write_lock_bh
- read_unlock_bh, write_unlock_bh

Spinlock: Einsatzszenarios

- Beispiel: Kritischer Abschnitt zwischen einer Treiberfunktion driver_read() und einer ISR.
- Folgende Szenarien sind zu beachten:
  1. Gleichzeitiger Aufruf von driver_read() auf zwei CPU-Kernen.
  2. Aufruf der ISR auf einem anderen Kern.
  3. Gleichzeitiger Aufruf von driver_read() auf dem eigenen, lokalen CPU-Kern (Preemption).
  4. Unterbrechung durch die ISR auf der lokalen CPU (Preemption).

Spinlock: Lösungsansätze

- Szenario 1 und 2 (Paralleler Aufruf auf einer zweiten CPU) lässt sich durch ein einfaches Spinlock schützen.
- Szenario 3 und 4 (Paralleler Aufruf auf der lokalen CPU) lässt sich durch das einfache Spinlock **nicht** schützen:
  - Interrupts auf der lokalen CPU müssen zusätzlich gesperrt werden.

Spinlock: Code spin_lock_irq (1)

```
static ssize_t driver_read( struct file *instanz,
    char __user *userbuffer,
    size_t count, loff_t *offset )
{
    ...
    spin_lock_irq( &mylock );
    /* Auf der lokalen CPU
   sind jetzt zusätzlich Interrupts gesperrt. */
    ... /* Kritischer Abschnitt */
    spin_unlock_irq( &mylock );
    ...
}
```

Spinlock: Code spin_lock_irq (2)

```
irqreturn_t interrupt_service_routine( int irq,
    void *dev_id )
{
    ...
    /* Interrupts sind bereits gesperrt,
        spin_lock reicht aus. */
    spin_lock( &mylock );
    ... /* Kritischer Abschnitt */
    spin_unlock( &mylock );
    ...
}
```

Spinlock: Fragen (1)

- Warum werden im vorherigen Beispiel innerhalb der ISR nur die Funktionen „spin_lock()“ und „spin_unlock()“, nicht jedoch die Varianten „spin_lock_irq()“ und „spin_unlock_irq()“ verwendet?
- Wann verwenden Sie die Varianten „spin_lock_irqsave()“ und „spin_unlock_irqsave()“?

Spinlock: Code spin_lock_irqsave (1)

```
static ssize_t driver_read( struct file *instanz,
char __user *userbuffer, size_t count,
 loff_t *offset )
{
     ...
     modify_private_list();
     ...
}

irqreturn_t interrupt_service_routine( int irq,
    void *dev_id )
{
    ...
    modify_private_list();
    ...
}
```

Spinlock: Code spin_lock_irqsave (2)

```
static void modify_private_list( void )
{
    unsigned long iflags;

    spin_lock_irqsave( &mylock, &iflags );
    ... /* Kritischer Abschnitt */
   spin_unlock_irqrestore( &mylock, &iflags );
}
```

Spinlock: Fragen (2)

- Wann verwenden Sie die Varianten „spin_lock_bh()“ und „spin_unlock_bh()“?
  - _bh Funktionen sperren Software Interrupts
  - lassen Hardware Interrupts jedoch zu!

Spinlock: Auswahl der effizientesten Methode

	Treiberinstanz	Kernel–Thread	Workqueue Event–Workqueue	Softirq	Tasklet Timer	Hardirq
Treiberinstanz	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Kernel–Thread	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Mutex RT–Mutex RW–Semaphore Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Workqueue Event–Workqueue	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock RW–Lock Seqlock	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Softirq	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Tasklet Timer	Spinlock–bh RW–Lock–bh Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock–irqsave RW–Lock–irqsave	Spinlock RW–Lock	Spinlock RW–Lock 1)	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave
Hardirq	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock–irqsave RW–Lock–irqsave Seqlock–irqsave	Spinlock RW–Lock 2)

1) Nur bei unterschiedlichen Tasklets. Ein Tasklet läuft zu einem Zeitpunkt garantiert nicht zweimal.
2) Nur bei unterschiedlichen ISRs. Ein und dieselbe ISR läuft zu einem Zeitpunkt nicht zweimal ab.

Reordering

- Problem:
  - Reordering durch die CPU
  - Reordering durch den Compiler
- Beispiel Hardware-Zugriff:
  - Notwendige Reihenfolge: Daten, Kommando
  - Aus Sicht der CPU sind das zwei voneinander unabhängige Zugriffe – Reordern ist erlaubt.
  - Aus Sicht der Hardware: Das Kommando kommt vor den Daten.

Reordering: Beispiel

Reordering: Lösung

- Spinlocks/Semaphore
- Besser: Memory Barriers
- Memory Barriers stellen sicher, dass die programmierte Abarbeitungsreihenfolge eingehalten wird:
  - mb()
  - rmb()
  - wmb()
  - barrier() (verhindert Reordering durch den Compiler)

Reordering: Code

- Ohne Barriers (Reordering Problem!)

```
...
outw( DATA_PORT, 0x0001 );
outw( COMMAND_PORT, 0x1122 );
...
```

- Mit Barriers

```
...
outw( DATA_PORT, 0x0001 );
wmb();
outw( COMMAND_PORT, 0x1122 );
wmb();
...
```

RaceCondition zw. KernelFunktion und mod_exit()

driver_read

```
if (no_data_available) {

    ....

    sleep_interruptible(&wait_queue);
```

ISR

```
no_data_available = FALSE;
wake_up_interruptible (&wait_queue)
...
```

Completion Objekt

- Dient der Synchronisation zwischen zwei Jobs:
  - Job 2 wartet darauf, dass Job 1 eine Aktion beendet.
- Funktionen:
  - DECLARE_COMPLETION
  - init_completion
  - complete
  - complete_all
  - complete_and_exit
  - wait_for_completion

Completion: Code (1)

```
DECLARE_COMPLETION( on_exit );
...
static int kernelthread( void *data )
{
    ...
    if( signal_pending( current ) ) {
        complete_and_exit( &on_exit, 0 );
        return 0;
    }
...
complete_and_exit( &on_exit, 0 );
return 0;
}
```

Completion: Code (2)

```
static void __exit mod_exit( void )
{
    kill_proc( thread_id, SIGTERM );
    wait_for_completion( &on_exit );
    ...
}
```

Sequenze Lock

- Basieren auf Spinlocks, stellen eine Abwandlung der R/W Locks dar
- Schreibender Zugriff wird sofort gewährt
- Lesender Zugriff muss überprüfen, ob während des Lesevorgangs sich die Daten geändert haben
- Software-Zugriff auf kritischen Bereich:
  - klassisch
    - Betreten, Zugreifen, Freigeben
  - Sequenz Locks:
    - Solange kritischen Bereich in einer Schleife lesen, bis sichergestellt ist, dass sich die Daten nicht geändert haben.

Konfliktvermeidung

- Kritische Abschnitte lassen sich beispielsweise durch alternative Datenstrukturen vermeiden.
- Auf Mehrkernmaschinen benötigt man für jede CPU eine eigene Datenstruktur:
  - Per-CPU-Variable

Zusammenfassung

- Linux bietet eine Reihe unterschiedlicher Methoden zum Schutz kritischer Abschnitte.
- Die Vielfalt resultiert aus dem Wunsch, ein möglichst preiswertes Verfahren anbieten zu können.
- Das Erkennen und das daraus erfolgende Absichern von kritischen Abschnitten ist eine der großen Herausforderungen bei der Kernel-Programmierung.
- Ebenfalls können im Kernel beim Warten auf kritische Bereiche auftretenden Prioritätsinversionen behandelt werden (rt_mutex).

Übersicht

1. Soft-IRQ's und Kernel-Threads
2. Schutz kritischer Bereiche
3. Hardware Anbindung

Datentransfer User/Kernel/Hardware

copy_to_user: Kopieren eines Speicherbereichs (wie

- *copy_to_user*: Kopieren eines Speicherbereichs (wie memcpy)
- *copy_from_user*: Kopieren eines Speicherbereichs (wie memcpy)
- *put_user*: Kopieren eines „orginären“ Datentyps (char, short, int, long)
- *get_user*: Kopieren eines „orginären“ Datentyps (char, short, int, long)

Hardwarezugriff

Hardwareanbindung

- „low level“-Treiber greifen direkt auf die Hardware zu.
- „high level“-Treiber nutzen Funktionen, die ihnen die „core driver“ zur Verfügung stellen. USB-Geräte greifen beispielsweise nie selbst direkt auf Hardware zu.

Datentransfer Kernel/Hardware (low level)

- Memory Mapped:
  - write[bwl]
  - read[bwl]
- IO-Ports:
  - out[bwl], out[bwl]_p, outs[bwl]
  - in[bwl], in[bwl]_p, ins[bwl]

Datentypen

Tabelle Datentypen

Datentyp	Erklärung
u8, u16, u32, u64	vorzeichenlose Datentypen
s8, s16, 32, s64	vorzeichenbehaftete Datentypen
__le16, __le32, __le64	little endian
__be16, __be32, __be64	big endian

Datenstrukturen

Normale Ablage

Adresse	Variable	
0xbffffa8c	u8	a
0xbffffa8d	s8	b
0xbffffa8e	frei	
0xbffffa8f		
0xbffffa90	s32	c
0xbffffa91		
0xbffffa92		
0xbffffa93		
0xbffffa94	s8	d
0xbffffa95	frei	
0xbffffa96	u16	e
0xbffffa97		

Gepackte Ablage

Adresse	Variable	
0xbffffa8c	u8	a
0xbffffa8d	s8	b
0xbffffa8e	s32	c
0xbffffa8f		
0xbffffa90		
0xbffffa91		
0xbffffa92	s8	d
0xbffffa93	u16	e
0xbffffa94		
0xbffffa95		
0xbffffa96		
0xbffffa97		

```
struct _LetterVar {
    __u8 a;
    __s8 b;
    __s32 c;
    __s8 d;
    __u16 e;
} __attribute__ ((packed));
```

Hardware-Erkennung

Vor dem Zugriff müssen die Hardware-Ressourcen reserviert werden.
Ressourcen:

- IO-Ports:
  - *request_region(), release_region()*
- Interrupts:
  - *request_irq(), free_irq()*
- Memory:
  - *request_mem_region(), release_mem_region()*
- DMA-Kanäle:
  - *request_dma(), release_dma()*

Manuelle Hardware-Erkennung

Manuelle Ressourcen-Detektion Hardware-Anbindung

Anbindung einer PCI-Hardware:

- Der Treiber spezifiziert die Geräte (Vendor- und Device-ID), für die er verantwortlich ist, sowie eine „Probe“ und eine „Remove“ Funktion.
- Der Treiber meldet sich beim PCI-Subsystem an.
- Ist eine entsprechende Hardware vorhanden, wird die Probefunktion aufgerufen. Der Treiber bekommt die zu allozierenden Ressourcen mitgeteilt.
- Nachdem die Ressourcen alloziert wurden, ist der „low level“ Zugriff wie beschrieben möglich.

Hardware-Anbindung: PCI

- Treiber- und Geräteinitialisierung sind voneinander getrennt.
- Standardnamen für Hardware-Erkennung und Geräteinitialisierung:
  - probe
  - init_one

PCI

&Probe–Funktion
&Remove–Funktion
Geräteidentifikation (ID–Table)
Anmelden beim PCI–Subsystem
Anmelden beim Kernel
module_init
Gerätebeschreibung
Reservierung der Ressourcen
Initialisierung der Hardware
Übernahme des Gerätes
Probe–Funktion
struct pci_dev
Aufruf, wenn ein Gerät des angegebenen Typs vorhanden ist oder angesteckt wird.
Freigabe der Ressourcen
Remove–Fktn.
Aufruf, wenn das Gerät entfernt wird oder sich der Treiber abmeldet.
Abmelden beim PCI–Subsystem
module_exit
Treiber
PCI–Subsystem