1 Einleitung

Computersysteme bestehen aus einer Anzahl unterschiedlicher Hard- und Softwarekomponenten, deren Zusammenspiel erst die Abarbeitung komplexer Programme ermöglicht. Zu den Hardwarekomponenten gehören beispielsweise die eigentliche Verarbeitungseinheit, der Mikroprozessor mit dem Speicher, aber auch die sogenannte Peripherie, wie Tastatur, Maus, Monitor, LEDs oder Schalter. Diese Peripherie wird über Hardwareschnittstellen an die Verarbeitungseinheit angeschlossen. Hierfür haben sich Schnittstellen wie beispielsweise USB (Universal Serial Bus) oder im Bereich der eingebetteten Systeme auch I^2C, SPI oder GPIO-Interfaces etabliert. Im PC-Umfeld ist PCI (Peripheral Component Interconnect) und PCI-Express verbreitet. Die Netzwerk- oder Grafikkarte wird beispielsweise über PCI-Express, Tastatur, Maus oder auch Drucker über USB mit dem Rechner verbunden.

Zentrale Komponente für den HW-Zugriff

Zu den Softwarekomponenten gehören das BIOS, das den Rechner nach dem Anschalten initialisiert, und das Betriebssystem. Das Betriebssystem koordiniert sowohl die Abarbeitung der Applikationen als auch die Zugriffe auf die Peripherie. Vielfach ersetzt man in diesem Kontext den Begriff Peripherie durch *Hardware* oder einfach durch *Gerät*, so dass das Betriebssystem den Zugriff auf die Hardware bzw. die Geräte steuert. Dazu muss es die unterschiedlichen Geräte kennen, bzw. es muss wissen, wie auf diese Geräte zugegriffen wird. Derartiges *Wissen* ist innerhalb des Betriebssystems in den Gerätetreibern hinterlegt. Sie stellen damit als Teil des Betriebssystemkerns die zentrale Komponente für den Hardwarezugriff dar. Ein Gerätetreiber ist eine Softwarekomponente, die aus einer Reihe von Funktionen besteht. Diese Funktionen wiederum steuern den Zugriff auf das Gerät.

Für jedes unterschiedliche Gerät wird ein eigener Treiber benötigt. So gibt es beispielsweise jeweils einen Treiber für den Zugriff auf die Festplatte, das Netzwerk oder die serielle Schnittstelle.

Da das Know-how über das Gerät im Regelfall beim Hersteller des Gerätes und nicht beim Programmierer des Betriebssystems liegt, sind innerhalb des Betriebssystemkerns Schnittstellen offengelegt, über die der

vom Hersteller erstellte Treiber für das Gerät integriert werden kann. Kennt der Treiberprogrammierer diese Schnittstellen, kann er seinen Treiber erstellen und den Anwendern Zugriff auf die Hardware ermöglichen.

Der Anwender selbst greift auf die Hardware über ihm bekannte Schnittstellen zu. Bei einem Unix-System ist der Gerätezugriff dabei auf den Dateizugriff abgebildet. Jeder Programmierer, der weiß, wie er auf normale Dateien zugreifen kann, ist imstande, auch Hardware anzusprechen.

Für den Anwender eröffnen sich neben dem einheitlichen Applikationsinterface noch weitere Vorteile. Hält sich ein Gerätetreiber an die festgelegten Konventionen zur Treiberprogrammierung, ist der Betriebssystemkern in der Lage, die Ressourcen zu verwalten. Er stellt damit sicher, dass die Ressourcen – wie Speicher, Portadressen, Interrupts oder DMA-Kanäle – nur einmal verwendet werden. Der Betriebssystemkern kann darüber hinaus ein Gerät in einen definierten, sicheren Zustand überführen, falls eine zugreifende Applikation beispielsweise durch einen Programmierfehler abstürzt.

Reale und virtuelle Geräte

Treiber benötigt man jedoch nicht nur, wenn es um den Zugriff auf reale Geräte geht. Unter Umständen ist auch die Konzeption sogenannter virtueller Geräte sinnvoll. So gibt es in einem Unix-System das Gerät `/dev/zero`, das beim lesenden Zugriff Nullen zurückgibt. Mit Hilfe dieses Gerätes lassen sich sehr einfach leere Dateien erzeugen. Auf das Gerät `/dev/null` können beliebige Daten geschrieben werden; sämtliche Daten werden vom zugehörigen Treiber weggeworfen. Dieses Gerät wird beispielsweise verwendet, um Fehlerausgaben von Programmen aus dem Strom sinnvoller Ausgaben zu filtern.

Kernelprogrammierung

Der Linux-Kernel lässt sich aber nicht nur durch Gerätetreiber erweitern. Erweiterungen, die nicht gerätezentriert sind, die vielleicht den Systemzustand überwachen, Daten verschlüsseln oder den zeitkritischen Teil einer Applikation darstellen, sind in vielen Fällen sinnvoll als Kernelcode zu realisieren.

Zur Kernelprogrammierung und zur Erstellung eines Gerätetreibers ist weit mehr als nur das Wissen um Programmierschnittstellen im Kernel notwendig. Man muss sowohl die Möglichkeiten, die das zugrunde liegende Betriebssystem bietet, kennen als auch die prinzipiellen Abläufe innerhalb des Betriebssystemkerns. Eine zusätzliche Erfordernis ist die Vertrautheit mit der Applikationsschnittstelle. Das gesammelte Know-how bildet die Basis für den ersten Schritt vor der eigentlichen Programmierung: die Konzeption.

Ziel dieses Buches ist es damit,

- den für die Kernel- und Treiberprogrammierung notwendigen theoretischen Unterbau zu legen,
- die durch Linux zur Verfügung gestellten grundlegenden Funktionalitäten vorzustellen,
- die für Kernelcode und Gerätetreiber relevanten betriebssysteminternen und applikationsseitigen Schnittstellen zu erläutern,
- die Vorgehensweise bei Treiberkonzeption und eigentlicher Treiberentwicklung darzustellen und
- Hinweise für ein gutes Design von Kernelcode zu geben.

Scope

Auch wenn viele der vorgestellten Technologien unabhängig vom Betriebssystem bzw. von der Linux-Kernel-Version sind, beziehen sich die Beispiele und Übungen auf den Linux-Kernel 4.x.

Ubuntu und Kernel 4.0.3

Die Beispiele sind auf einem Ubuntu-Linux (Ubuntu 14.04) und dem Kernel 4.0.3 beziehungsweise einem Raspberry Pi 2 unter dem Betriebssystem Raspbian in der Version 2015.03 und dem Kernel in Version 4.0.3 getestet worden. Welche Distribution, ob Debian (pur, in der Ubuntu- oder Raspbian-Variante), Arch Linux, Fedora, SuSE, Red Hat oder ein Selbstbau-Linux (beispielsweise auf Basis von Buildroot), dabei zum Einsatz kommt, spielt im Grunde aber keine Rolle. Kernelcode ist abhängig von der Version des Betriebssystemkerns, nicht aber direkt abhängig von der verwendeten Distribution (Betriebssystemversion).

UP und SMP

Das Gleiche gilt bezüglich des Einsatzfeldes. Dank seiner hohen Skalierbarkeit ist Linux das erste Betriebssystem, das in eingebetteten Systemen, in Servern, auf Desktop-Rechnern oder sogar auf der Mainframe läuft. Die vorliegende Einführung deckt prinzipiell alle Einsatzfelder ab. Dabei spielt es keine Rolle, ob es sich um eine Einprozessormaschine (Uniprocessor System, UP) oder um eine Mehrprozessormaschine (Symmetric Multiprocessing, SMP) handelt.

Aufbau des Buches

Zu einer *systematischen* Einführung in die Treiberprogrammierung gehört ein solider theoretischer Unterbau. Dieser soll im folgenden Kapitel gelegt werden. Wer bereits gute Betriebssystemkenntnisse hat und für wen Begriffe wie *Prozesskontext* und *Interrupt-Level* keine Fremdwörter sind, kann diesen Abschnitt überspringen. Im Anschluss werden die Werkzeuge und Technologien vorgestellt, die zur Entwicklung von Treibern notwendig sind. In der vierten Auflage wurde dieses Kapitel um einen Abschnitt über die Cross-Entwicklung ergänzt.

Bevor mit der Beschreibung des Treiberinterface im Betriebssystemkern begonnen werden kann, muss das Applikationsinterface zum Treiber hin vorgestellt werden. Denn was nützt es, einen Gerätetreiber zu schreiben, wenn man nicht im Detail weiß, wie die Applikation später auf den Treiber zugreift? Immerhin muss die von der Applikation geforderte Funktionalität im Treiber realisiert werden.

Das folgende Kapitel beschäftigt sich schließlich mit der Treiberentwicklung als solcher. Hier werden insbesondere die Funktionen eines Treibers behandelt, die durch die Applikation aufgerufen werden. In diesem Abschnitt finden Sie auch ein universell einsetzbares Treibertemplate.

Darauf aufbauend werden die Komponenten eines Treibers behandelt, die unabhängig (asynchron) von einer Applikation im Kernel ablaufen. Stichworte hier: Interrupts, Softirqs, Tasklets, Kernel-Threads oder auch Workqueues. Ergänzend finden Sie hier das notwendige Knowhow zum Sichern kritischer Abschnitte, zum Umgang mit Zeiten und zur effizienten Speicherverwaltung.

Mit diesen Kenntnissen können bereits komplexere Treiber erstellt werden, Treiber, die sich jetzt noch harmonisch in das gesamte Betriebssystem einfügen sollten. Diese Integration des Treibers ist folglich Thema eines weiteren Kapitels.

Neben den bisher behandelten Treibern für zeichenorientierte Geräte (Character Devices) werden für die Kernelprogrammierung relevante Subsysteme wie GPIO, I^2C, USB, Netzwerk und Blockgeräte vorgestellt. Hier zeigen wir Ihnen auch, wie Sie im Kernel existierende und eigene Verschlüsselungsverfahren verwenden.

Einen Treiber zu entwickeln, ist die eine Sache, gutes Treiberdesign eine andere. Dies ist Thema des letzten Kapitels.

Im Anhang schließlich finden sich Hinweise zur Generierung und Installation des Kernels für die PC-Plattform und für den Raspberry Pi. Die Referenzliste der wichtigsten Funktionen, die im Kontext der Kernelprogrammierung eine Rolle spielen, lassen das Buch zu einem Nachschlagewerk werden.

Notwendige Vorkenntnisse

C-Kenntnisse

Das vorliegende Buch ist primär als eine systematische Einführung in das Thema gedacht. Grundkenntnisse im Bereich der Betriebssysteme sind empfehlenswert. Kenntnisse in der Programmiersprache C sind

zum Verständnis unabdingbar. Vor allem der Umgang mit Pointern und Funktionsadressen sollte vertraut sein.

Zusätzliche Informationsquellen

Errata und Beispielcode zum Buch

Errata und vor allem auch den Code zu den im Buch vorgestellten Beispieltreibern finden Sie unter https://ezs.kr.hsnr.de/TreiberBuch/.

Quellcode online

Die sicherlich wichtigste Informationsquelle zur Erstellung von Gerätetreibern ist der Quellcode des Linux-Kernels selbst. Wer nicht mit Hilfe der Programme find und grep den Quellcode durchsuchen möchte, kann auf die »Linux Cross-Reference« (http://lxr.free-electrons.com/) zurückgreifen. Per Webinterface kann der Quellcode angesehen, aber auch nach Variablen und Funktionen durchsucht werden.

In den Kernel-Quellen befindet sich eine sehr hilfreiche Dokumentation. Ein Teil der Dokumentation besteht aus Textdateien, die sich mit jedem Editor ansehen lassen. Ein anderer Teil der Dokumentation muss erst erzeugt werden. Dazu wird im Hauptverzeichnis der Kernel-Quellen (`/usr/src/linux/`) eines der folgenden Kommandos aufgerufen:

```
(root)# make psdocs    # für Dokumentation in Postscript
(root)# make pdfdocs   # für Dokumentation in PDF
(root)# make htmldocs  # für HTML-Dokumentation
```

Dokumentation als Teil der Kernel-Quellen

Sind die notwendigen DocBook-Pakete installiert (unter Ubuntu 14.04 unter anderem das Paket `docbook-utils`), werden eine Reihe unterschiedlicher Dokumente generiert und in das Verzeichnis `/usr/src/linux/Documentation/DocBook/` abgelegt. Insbesondere sind hier die folgenden Dokumente zu finden:

device-drivers Dieses Dokument enthält die Beschreibung von Betriebssystemkern-Funktionen, die insbesondere für Entwickler von Gerätetreibern interessant sind.

gadget Eine Einführung in die Erstellung von USB-Slavetreibern

genericirq Eine Einführung in die Interruptverarbeitung, insbesondere auch der Programmierschnittstellen, im Linux-Kernel

kernel-api Dieses Dokument enthält die Beschreibung von Funktionen des Betriebssystemkerns.

kernel-hacking Kernel-Entwickler Rusty Russell führt in einige Grundlagen der Kernel-Entwicklung ein. Leider ist das Dokument nicht mehr aktuell.

kernel-locking Rusty Russell: »Unreliable Guide to kernel-locking«. Hier finden sich einige Aspekte wieder, die die Vermeidung beziehungsweise den Schutz kritischer Abschnitte betreffen.

librs Dieses Dokument enthält die Beschreibung der Reed-Solomon-Bibliothek, die Funktionen zum Kodieren und Dekodieren enthält.
mac80211 Beschreibung des mac80211-Subsystems
parportbook Eine Einführung in die Erstellung von Treibern, die auf die parallele Schnittstelle über das Parport-Subsystem von Linux zugreifen
regulator Eine Beschreibung des Spannungs- und Regulator-Interface (linkstart;regulators driver interface)
uio-howto Howto zu Userspacetreiber (UIO siehe auch [KuQu11/07])
writing_usb_driver Eine Einführung in die Erstellung von USB-Hosttreibern

Neben der Dokumentation, die den Kernel-Quellen beiliegt, gibt es noch diverse Informationsquellen im Internet:

Online-Quellen

http://www.lwn.net Immer donnerstags gibt es hier aktuelle Kernel-News sowie Tipps und Tricks rund um die Kernel- und Treiberprogrammierung. Die ganz aktuelle Ausgabe steht jeweils nur der zahlenden Klientel zur Verfügung. Wer ohne Obolus auskommen will, kann die jeweils vorherige Ausgabe kostenlos lesen.
http://free-electrons.com Sehr wertvolle, praxisorientierte Infos zur Kernel- und Treiberprogrammierung in Form von Tutorials und Foliensätzen. Das Material ist allerdings vorwiegend in Englisch.
http://www.kernel.org Der Server »kernel.org« ist die zentrale Stelle für aktuelle und auch für alte Kernelversionen. Darüber hinaus finden sich hier die Patches einiger Kernelentwickler.
http://www.lkml.org Hier lässt sich die Kernel-Mailing-Liste aktuell mitlesen, ohne selbst eingeschrieben sein zu müssen.
http://www.kernelnewbies.org Hier finden sich viele Einsteigerinformationen und Programmiertricks.
http://www.heise.de/open Unter dem Titel »Kernel-Log« wird hier mit jeder Kernelversion eine detaillierte Zusammenfassung der neuen Features veröffentlicht.

Zu jeweiligen Spezialgebieten der Kernelprogrammierung und Treiberentwicklung gibt es im Internet überdies einige Texte oder Artikel. Hier ist der Leser allerdings selbst gefordert, mit Hilfe einer Suchmaschine Zusatzmaterial zu finden.

Ergänzungen

Zur Abrundung des Themas werden noch die beiden folgenden Bücher empfohlen:

- Quade, Mächtel: Moderne Realzeitsysteme kompakt. Eine Einführung mit Embedded Linux. dpunkt.verlag 2012 ([QuMä2012]). Das Buch behandelt verstärkt die Userland-Aspekte, also beispielsweise die Konzeption und Realisierung von realzeitfähigen Applikationen.
- Quade: Embedded Linux lernen mit dem Raspberry Pi. Linux-Systeme selber bauen und programmieren. dpunkt.verlag 2014 ([Quade2014]). Das Buch behandelt vor allem die Systemaspekte und zeigt, wie aus den einzelnen Komponenten (Kernel, Treiber, Userland, Applikation) komplette Systeme gebaut werden.

Zu guter Letzt bleibt noch der Verweis auf unsere Artikelserie im Linux-Magazin ab Ausgabe 8/2003, die das Thema Kernelprogrammierung behandelt. In dieser Reihe sind inzwischen weit über 80 Artikel erschienen, die neben der Treiberentwicklung auch praxisorientiert den Linux-Kernel selbst vorstellen. Die Mehrzahl der Artikel kann kostenlos im Internet gelesen werden.

```
    unregister_chrdev_region( gpio_dev_number, 1 );
    free_irq(rpi_irq_17, driver_object);
    gpio_free( 17 );
    return;
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

6.3 Softirqs

Softirqs ermöglichen die hochpriore Verarbeitung von Funktionen.

Direkt nach Abarbeitung einer Hardware-ISR beziehungsweise nach Freigabe der Interrupts überprüft der Kernel, ob weitere wichtige Funktionen/Routinen abzuarbeiten sind. Diese Funktionen werden Softirqs genannt.

In den Kernelquellen ist hierzu ein Feld von 32 Softirq-Routinen (siehe Datei `<kernel/softirq.c>` im Kernelquellcode) angelegt, wovon zehn bereits vordefiniert sind. Ein zugehöriges Bit im Variablenfeld `irq_stat` gibt an, ob eine Routine aufgerufen werden muss (Bit=1) oder nicht (Bit=0).

Das Feld der Softirq-Routinen wird über die Funktion open_softirq belegt. Die vordefinierten Softirqs sind in Abbildung 6-4 dargestellt.

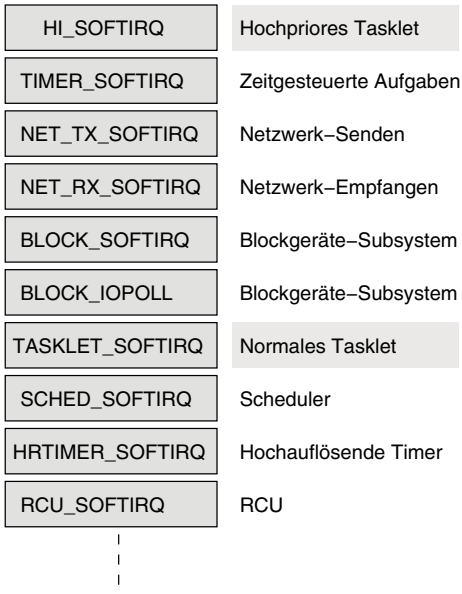

Abb. 6-4
Vordefinierte Softirqs

Für Softirqs gilt generell:

- Die Abarbeitung erfolgt im Interruptkontext. Interrupts selbst sind dabei aber zugelassen und unterbrechen – falls sie auftreten – die Abarbeitung des Softirqs.
- Die vordefinierten Softirqs werden entsprechend der oben dargestellten Reihenfolge priorisiert. Ein Timer-Softirq wird daher vor den Net-Softirqs abgearbeitet.
- Ein Softirq kann bei Mehrprozessorsystemen mehrfach parallel ablaufen.

Für den Treiberentwickler sind insbesondere die Ausprägungen Tasklet sowie Timer interessant. Sie werden im Folgenden genauer vorgestellt.

6.3.1 Tasklets

Tasklets stellen eine spezifische Ausprägung der Softirqs dar.

Der Kernelprogrammierer setzt Tasklets ein, um längere Berechnungen, die im Kontext eines Interrupts notwendig werden, abarbeiten zu lassen. Fänden diese Berechnungen innerhalb der Hardware-ISR statt, hätte das längere Interrupt-Latenzzeiten zur Folge, was in jedem Fall zu vermeiden ist. Daher teilt man die Verarbeitung einer ISR in zwei Schritte auf: Während des ersten Schrittes sind Interrupts gesperrt, nur die (zeit-)kritischen Aktionen werden durchgeführt. Die übrigen Berechnungen werden in einem zweiten Schritt – bei freigegebenen Interrupts – behandelt. Dieser früher Bottom-Half genannte Teil wird typischerweise in Form eines Tasklets realisiert.

Tasklets sind vom Entwickler kodierte Funktionen, die vom Kernel zusammen mit einem Parameter aufgerufen werden. Die Adresse der Tasklet-Funktion und das ihr als Parameter zu übergebende Datum werden in einer Instanz der Datenstruktur `struct tasklet_struct` (Prototyp in `<linux/interrupt.h>`) beschrieben. Die Initialisierung der Datenstruktur `struct tasklet_struct` kann statisch (also durch den Compiler) durch das Makro DECLARE_TASKLET oder dynamisch (also innerhalb einer Funktion zur Laufzeit) mit Hilfe der Funktion tasklet_init erfolgen (siehe Beispiele 6-6 und 6-7).

Beispiel 6-6 *Statische Initialisierung einer Tasklet-Struktur*

```
static void tasklet_function( unsigned long data );                    (1)
DECLARE_TASKLET( tl_descr, tasklet_function, 0L );                     (2)
                /* statische Def. */
```

(1) Deklaration der Funktion, die abgearbeitet werden soll, wenn das Tasklet aufgerufen wird

2 Mit DECLARE_TASKLET wird ein Element vom Typ `struct tasklet_struct` mit Namen »tl_descr« angelegt. Dieses Element wird mit der Adresse der aufzurufenden Funktion (tasklet_function) und einem zu übergebenden Datum (hier »0L«) initialisiert.

```
struct tasklet_struct tl_descr;
...
static int __init mod_init(void)
{
    tasklet_init( &tl_descr, tasklet_function, 0L );
    ...
}
```

Beispiel 6-7 *Dynamische Initialisierung einer Tasklet-Struktur*

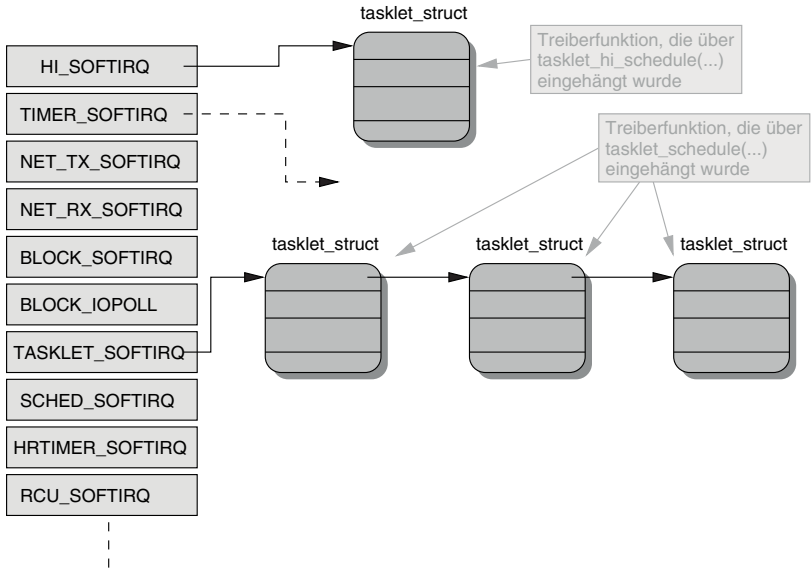

Abb. 6-5 *Tasklets als Teil der Softirq-Verarbeitung*

Zur Abarbeitung freigeben

Das so definierte Tasklet muss nur noch zum richtigen Zeitpunkt zur Abarbeitung freigegeben werden. Typischerweise findet dieser Vorgang innerhalb einer Hardware-ISR durch Aufruf der Funktion tasklet_schedule oder der Funktion tasklet_hi_schedule statt. Welche der beiden Funktionen verwendet wird, hängt von der Wichtigkeit der innerhalb des Tasklets zu erledigenden Aufgaben ab. Für die Bearbeitung stehen nämlich zwei Prioritätsstufen zur Verfügung. Ein mit tasklet_hi_schedule eingehängtes Tasklet wird auf der Prioritätsstufe `HI_SOFTIRQ` abgearbeitet. Das bedeutet, dass die Tasklet-Funktion wirklich direkt nach dem Ende einer Hardware-ISR die CPU zugeteilt bekommt (siehe Abbildung 6-5). Ein mit der Funktion tasklet_schedule eingehängtes Tasklet dagegen wird erst dann abgearbeitet, wenn kein anderer Softirq

mehr anliegt; dieses Tasklet hat also innerhalb der Gruppe der Softirqs die niedrigste Priorität (TASKLET_SOFTIRQ).

Um ein übersichtliches und abgeschlossenes Beispiel zur Hand zu haben, wird das Tasklet im Beispiel 6-8 nicht während einer ISR, sondern bereits während der Modulinitialisierung »geschedult«. Wird der Code kompiliert und das so generierte Modul geladen, erscheint in den Syslogs die Meldung »Tasklet called... «.

Beispiel 6-8 *Einfaches Tasklet*

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

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

DECLARE_TASKLET( tl_descr, tasklet_func, 0L );

static int __init mod_init(void)
{
    printk(KERN_INFO "mod_init called\n");
    tasklet_schedule( &tl_descr ); /* Tasklet sobald als moeglich ausfuehren */
    return 0;
}

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

module_init( mod_init );
module_exit( mod_exit );

MODULE_LICENSE("GPL");
```

Einhängen ohne direkten Aufruf

Zum Schedulen eines Tasklets werden innerhalb der Funktionen tasklet_schedule beziehungsweise tasklet_hi_schedule Interrupts gesperrt. Zwar ist dies in den meisten Fällen unproblematisch, doch gibt es Situationen, in denen Interrupts nicht gesperrt werden sollen oder können. Für solche Fälle bietet Kernel 4.0 die Möglichkeit, das Tasklet vorzeitig einzuhängen, die Abarbeitung aber durch Setzen eines Flags zu verhindern. Zum Setzen dieses Flags dient die Funktion tasklet_disable. Die Ausführung der Tasklet-Funktion kann zum gewünschten Zeitpunkt – ohne Interrupts sperren zu müssen – eingefordert werden. Der Aufruf von tasklet_enable reicht hierzu aus. Wird anstelle des Makros DECLARE_TASKLET das Makro DECLARE_TASKLET_DISABLED

verwendet, wird das Disable-Flag bereits direkt bei einer statischen Initialisierung gesetzt.

Mit einem Tasklet lässt sich zwar auf der einen Seite die Interrupt-Latenzzeit des Kernels verbessern, es beeinflusst aber auf der anderen Seite auch die Task-Latenzzeit. Aus diesem Grund sollte ein Tasklet so kurz wie möglich gehalten werden. Hinzu kommt, dass Tasklets im Interruptkontext abgearbeitet werden, also sich nicht schlafen legen dürfen! Spätestens wenn diese Funktionalität benötigt wird, muss der Treiberentwickler auf die Verwendung von Kernel-Threads ausweichen.

Ein Tasklet wird zu einem Zeitpunkt maximal einmal aufgerufen – so die Spezifikation. Das gilt auch für Mehrprozessorsysteme. Unterschiedliche Tasklets zur gleichen Zeit einzusetzen, ist allerdings möglich.

Achtung, Race Condition!

Abschließend noch eine Warnung: Beim Gebrauch von Tasklets kann der Entwickler leicht in eine Race Condition verstrickt werden. Wird nämlich das Modul entladen und ruft der Kernel im Anschluss ein von diesem Modul geschedultes Tasklet auf, soll der Prozessor Code abarbeiten, der längst nicht mehr vorhanden ist. Das aber ist unmöglich. Es kommt zu einer Oops-Meldung. Zu den Aufgaben des Treiberentwicklers gehört von daher, sicherzustellen, dass jegliches Tasklet vor Entladen des Moduls entweder abgearbeitet oder entfernt wurde. Die entsprechende Funktion heißt tasklet_kill. Sie kann auch dann gefahrlos aufgerufen werden, wenn zum Zeitpunkt des Aufrufes das Tasklet nicht geschedult ist.

6.3.2 Timer-Funktionen

Neben Tasklets ist für Treiberentwickler noch eine weitere Variante der Softirqs überaus interessant, die sogenannten Timer. Mit ihrer Hilfe kann der Entwickler den Kernel beauftragen, Funktionen zu einem definierten, späteren Zeitpunkt auszuführen.

Abarbeitung zu definierten Zeitpunkten

Dazu muss man zunächst wissen, dass Zeiten innerhalb des Kernels historisch nicht absolut, sondern relativ zum Einschaltzeitpunkt verarbeitet werden. Als Basis gilt die Anzahl der seit dem Einschalten ausgelösten, periodischen Timer-Interrupts. Sie wird in einem Zähler mit dem Namen `jiffies` gezählt (siehe Kapitel 6.6.1).

Um einen Kerneltimer zu verwenden, wird im Treiber eine Datenstruktur vom Typ `struct timer_list` alloziert. Die für den Kernel spezifischen Felder dieser Datenstruktur werden über die Funktion init_timer initialisiert (hier ist nur eine dynamische Initialisierung, also innerhalb einer Funktion, möglich). Anschließend sind die übrigen Felder mit der

Adresse der aufzurufenden Funktion (Feld `function`), mögliche Daten für die Funktion (Feld `data`) und dem Zeitpunkt, zu dem die Funktion aufgerufen werden soll (Feld `expires`), zu spezifizieren (Beispiel 6-9). Der Abarbeitungszeitpunkt wird absolut in `jiffies` angegeben. Um eine Funktion relativ zum momentanen Zeitpunkt aufzurufen, wird zum aktuellen Zeitpunkt der Relativwert aufaddiert. Liegt der Zeitpunkt, zu dem die Timer-Funktion aufgerufen werden soll, bereits in der Vergangenheit, wird die Routine sofort ausgeführt.

Beispiel 6-9 *Initialisierung der `timer_list`*

```
static struct timer_list ptimer;
static void timer_funktion(unsigned long);
...
static int __init mod_init(void)
{
    init_timer( &ptimer );
    ptimer.function = timer_funktion;
    ptimer.data = 0;
    ptimer.expires = jiffies + (2*HZ); /* alle 2 Sekunden */
    ...
```

Periodische Timer

Sobald der im Expires-Feld angegebene Zeitpunkt erreicht beziehungsweise überschritten ist, wird die dort spezifizierte Funktion genau einmal aufgerufen. Soll eine Funktion periodisch abgearbeitet werden, muss sie die zugehörige `timer_list` mit dem nächsten Aufrufzeitpunkt initialisieren und dann dem Kernel erneut übergeben (so zu sehen in Beispiel 6-11; hier wird die Timer-Funktion periodisch alle zwei Sekunden aufgerufen).

Die Funktion add_timer schließlich übergibt den Treiber zur Ausführung an den Kernel. Damit ist der Timer »aktiviert«. Der Kernel sorgt dafür, dass die in der Struktur angegebene Funktion mit den Daten als Parameter zum spezifizierten Zeitpunkt aufgerufen wird.

Es ist nicht erlaubt, einen bereits aktivierten Timer ein zweites Mal zu aktivieren!

Wie andere Softirqs auch, wird die Timer-Funktion im Interruptkontext aufgerufen. Das bedeutet: Die Timer-Funktion ist möglichst kurz zu halten. Sie kann sich nicht schlafen legen! Überdies ist ein Zugriff auf User-Prozesse (Applikationen) innerhalb der Timer-Funktion nicht möglich.

Deaktivieren.

Ist der Timer Teil eines Moduls und soll das Modul wieder entladen werden, ist jeder noch nicht abgelaufene Timer aus dem System zu entfernen (Deaktivieren des Timers). Dazu existiert die Funktion del_timer_sync. Auf SMP-Maschinen deaktiviert diese Funktion nicht nur einen Timer, sondern wartet darüber hinaus noch so lange, bis die möglicherweise auf einer anderen CPU aktive Timer-Funktion beendet ist.

Auf einer Einprozessormaschine wird del_timer_sync auf die Funktion del_timer abgebildet, die den Timer nur deaktiviert.

```
static void __exit mod_exit(void)
{
    del_timer_sync( &ptimer );
    ...
```

Bevor ein Timer deaktiviert wird, ist zu verhindern, dass der Timer erneut eingehängt (add_timer) wird. Anders als bei den Tasklets muss dazu der Treiber seinen eigenen Mechanismus implementieren (beispielsweise über ein Flag, wie in Beispiel 6-10).

Beispiel 6-10
Stoppen periodischer Funktionen

```
static atomic_t nicht_mehr_einhaengen = 0;                          (1)

void timer_funktion( unsigned long parameter )
{
    if (atomic_read( &nicht_mehr_einhaengen )) {                    (2)
        complete( &on_exit );
    } else {
        add_timer( &ptimer );
    }
    ...
}

static void __exit mod_exit(void)                                   (3)
{
    atomic_set( &nicht_mehr_einhaengen, 1 );
    wait_for_completion( &on_exit );
    del_timer_sync( &ptimer ); /* nicht mehr notwendig */           (4)
    ...
```

(1) Mit diesem Flag wird synchronisiert, ob der Timer noch eingehängt werden darf oder nicht. Der Datentyp »atomic_t« wird in Tabelle 6-1 vorgestellt.

(2) Wenn das Flag nicht gesetzt ist, darf der Timer weiterhin verwendet werden.

(3) Das Modul wird entladen, der Timer ist aus dem System zu entfernen. Durch das Setzen des Flags wird sichergestellt, dass nicht zwischenzeitlich der Timer erneut aktiviert wird.

(4) Mit dieser Funktion wird ein Timer aus dem Kernel entfernt. Es sollte grundsätzlich die »sync«-Variante (del_timer_sync) verwendet werden.

Um festzustellen, ob ein Timer aktiviert ist oder nicht, gibt es die Funktion timer_pending. Diese gibt »1« zurück, falls der Timer aktiviert ist, andernfalls »0«.

```
if (timer_pending( &ptimer )) {
    printk( "Timer ist aktiviert.\n" );
} else {
    printk( "Timer ist nicht aktiviert.\n" );
}
```

Weitere Timer-Funktionen

Noch zwei weitere Funktionen erleichtern den Umgang mit Timern. Die Funktion mod_timer ermöglicht es, bei einem aktivierten Timer den Zeitpunkt zu modifizieren, zu dem die Timer-Funktion aufgerufen werden soll.

Mit der Funktion add_timer_on ist der Entwickler in der Lage, für ein SMP-System festzulegen, auf welchem Prozessor eine Timer-Funktion ablaufen soll.

Beispiel 6-11 *Verwendung eines Timers*

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/timer.h>
#include <linux/sched.h>
#include <linux/init.h>

static struct timer_list mytimer;

static void inc_count(unsigned long arg)
{
    printk("inc_count called (%ld)...\n", mytimer.expires );
    mytimer.expires = jiffies + (2*HZ); /* 2 second */
    add_timer( &mytimer );
}

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( timer_pending( &mytimer ) )
        printk("Timer ist aktiviert ...\n");
    if( del_timer_sync( &mytimer ) )
        printk("Aktiver Timer deaktiviert\n");
    else
        printk("Kein Timer aktiv\n");
}
```

```
module_init( ktimer_init );
module_exit( ktimer_exit );

MODULE_LICENSE("GPL");
```

Ein Timer darf nur einmal aktiviert sein.

Etwas komplizierter wird der Umgang mit Timern, wenn ein und derselbe Treiber an mehreren Stellen die Funktion add_timer mit dem gleichen Timer-Objekt aufruft. Da es zu Abstürzen kommen kann, falls der Kernel damit beauftragt wird, ein einzelnes Timer-Objekt zu einem Zeitpunkt gleich mehrfach abzuarbeiten, muss der Entwickler vor dem Aktivieren des Timers überprüfen, ob der Timer nicht bereits aktiviert wurde. Der Vorgang des Überprüfens und das Einhängen stellen einen kritischen Abschnitt dar, der zu schützen ist. Dazu bietet sich ein Spinlock an (siehe Beispiel 6-12).

Beispiel 6-12 *Sicheres Einhängen eines Timer-Objekts*

```
static struct spinlock_t timer_lock;
...

void secure_add_timer( struct timer_list *ptimer )
{
    unsigned long flags;

    spin_lock_irqsave( &timer_lock, flags );
    if (!timer_pending( ptimer )) {
        add_timer( ptimer );
    }
    spin_unlock_irqrestore( &timer_lock, flags );
}
```

6.3.3 High Resolution Timer

Grundlagen

Ab Kernel 2.6.16 haben die Entwickler sukzessive die auf periodische Interrupts beruhende Zeitverwaltung gegen ein auf dynamischen Ticks basierendes Zeitmanagement (dyntick) ausgetauscht. Der Kernel berechnet intern den nächsten Zeitpunkt, zu dem er aktiv werden muss, und programmiert die Timer-Bausteine so, dass genau zu diesem Zeitpunkt ein Interrupt auftritt, der die notwendige Aktion anstößt. Der neue Code bildet nicht nur die Basis für wirklich hoch zeitauflösende Timer, sondern auch für ein beeindruckend hochgenaues Zeitverhalten (High Precision Timer). Der Programmierer hat Zugriff auf dieses Zeitmanagement über die sogenannten High Resolution Timer (hrtimer).

Zeitquellen

Das Zeitmanagement stellt zwei Zeitquellen zur Verfügung. Die Zeitquelle `CLOCK_MONOTONIC` repräsentiert die Zeit, so wie sie im Rechner mitgeführt wird. Sprünge kommen dabei nicht vor, auch wenn die Uhr mal vor oder mal zurückgesetzt werden sollte. Sie zählt immer weiter.

CLOCK_REALTIME repräsentiert die Zeit außerhalb des Rechners. Dreht der Admin auf seinem Rechner an der Zeitschraube, hat das direkten Einfluss auf diese Quelle.

Außerdem führt die auf hrtimer beruhende Zeitverwaltung den neuen Zeitdatentyp ktime_t ein, der auf Nanosekunden beruht. Dieser Datentyp und die Operatoren darauf werden in Abschnitt 6.6.1 auf S. 248 genauer vorgestellt.

Funktionen

Um hrtimer programmtechnisch zu nutzen, definieren Sie ein Objekt vom Typ struct hrtimer. Zur Initialisierung des Objekts mit Hilfe der Funktion hrtimer_init definieren Sie die Zeitquelle (CLOCK_MONOTONIC oder CLOCK_REALTIME) und ob der Zeitpunkt, zu dem der Timer aktiv werden soll, relativ oder absolut angegeben wird (HRTIMER_MODE_REL oder HRTIMER_MODE_ABS). Schließlich weisen Sie noch die Callback-Funktion zu, die im Fall des Auslösens aufgerufen wird. Zum eigentlichen Starten definieren Sie den Auslösezeitpunkt mit Hilfe der Funktionen, die Sie in Tabelle 6-2 finden, und übergeben den Zeitpunkt durch Aufruf von hrtimer_start_range_ns oder hrtimer_start. Allerdings sollten Sie – falls möglich – die Funktion hrtimer_start_range_ns einsetzen. Diese gibt dem Kernel die Chance, Aufgaben zu bündeln. Dadurch wird die Chance erhöht, häufiger nicht genutzte Systemteile abzuschalten und Energie zu sparen (Leistungsbündelung).

Callback

Die Callback-Funktion selbst bekommt, wenn sie aufgerufen wird, die Adresse des zugehörigen Timer-Objekts übergeben. Da sie im Interruptkontext abläuft, gelten die üblichen Einschränkungen beispielsweise beim Reservieren von Speicher. Rückgabewert der Funktion ist entweder HRTIMER_RESTART oder HRTIMER_NORESTART. Soll der Callback gleich wieder gestartet werden (HRTIMER_RESTART), dürfen Sie nicht vergessen, durch Aufruf beispielsweise der Funktion hrtimer_add_expires_ns einen neuen, in der Zukunft liegenden Zeitpunkt zu setzen.

Beispiel 6-13
High Resolution Timer

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

struct hrtimer hrt_monotonic, hrt_realtime;

static enum hrtimer_restart timer_function( struct hrtimer *hrt )
{
    printk("%ld: timer_function( %p )\n", jiffies, hrt );
    return HRTIMER_NORESTART;
}
```

```
static int __init mod_init(void)
{
    struct timespec tp;
    ktime_t tim, absolut;

    printk("mod_init called\n");
    hrtimer_get_res( CLOCK_MONOTONIC, &tp );
    printk("monotonic(%p): %ld, %ld\n",
        &hrt_monotonic, tp.tv_sec, tp.tv_nsec );
    hrtimer_get_res( CLOCK_REALTIME, &tp );
    printk("realtime(%p): %ld, %ld\n",
        &hrt_realtime, tp.tv_sec, tp.tv_nsec );

    hrtimer_init( &hrt_monotonic, CLOCK_MONOTONIC, HRTIMER_MODE_REL );
    hrtimer_init( &hrt_realtime, CLOCK_REALTIME, HRTIMER_MODE_ABS );
    hrt_monotonic.function = timer_function;
    hrt_realtime.function = timer_function;
    tim = ktime_set( 15, 0 );
    hrtimer_start( &hrt_monotonic, tim, HRTIMER_MODE_REL );
    ktime_get_real_ts( &tp );
    absolut = timespec_to_ktime(tp);
    tim = ktime_add( tim, absolut );
    hrtimer_start( &hrt_realtime, tim, HRTIMER_MODE_ABS );
    printk("Timer activated at %ld jiffies\n", jiffies );
    return 0;
}

static void __exit mod_exit(void)
{
    hrtimer_cancel( &hrt_monotonic );
    hrtimer_cancel( &hrt_realtime );
    printk("mod_exit called\n");
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

Beispiel 6-13 zeigt Ihnen sowohl die Verwendung der beiden Zeitquellen als auch den Umgang mit dem neuen Zeitdatentyp `ktime_t`. Die beiden Timer werden so aktiviert, dass sie nach 15 Sekunden die Callback-Funktion aufrufen, die eine Ausgabe per printk macht. Anhand der Ausgaben können Sie den Unterschied zwischen den beiden Zeitquellen sehen, falls Sie während der Wartezeit an der Zeitschraube drehen. Dieses Experiment sollten Sie natürlich keinesfalls auf einem Produktivsystem durchführen! Übersetzen Sie den Quellcode und laden Sie den Treiber. Beobachten Sie dabei die Ausgabe (Syslog), die erwartungsgemäß nach exakt 15 Sekunden zweimal erfolgt. Entfernen Sie den Treiber wieder und laden Sie ihn erneut. Danach stellen Sie die Systemzeit

um fünf Sekunden vor. Der Timer `hrt_realtime` bekommt diese Modifikation mit und verkürzt die Wartezeit um fünf Sekunden, bevor er die Funktion `print_jiffies` aufruft. Der Timer `hrt_monotonic` ignoriert den Zeithüpfer, wartet genau 15 Sekunden und ruft dann die Funktion auf. Absolut betrachtet kommt dieser Aufruf fünf Sekunden zu spät.

6.3.4 Tasklet auf Basis des High Resolution Timers

Neben den klassischen Timern bieten Kernel ab Version 2.6.16 die Möglichkeit, Tasklets auf Basis der hrtimer zu realisieren. Wie in Abschnitt 6.6 ausgeführt, stellen hrtimer hochauflösende und hochpräzise Timer zur Verfügung, die auf einem neuen Zeitverwaltungssystem beruhen.

Initialisierung

Diese Timer-Tasklets sind leicht einzusetzen. Sie benötigen ein Tasklet-Objekt, welches mit Hilfe der Funktion tasklet_hrtimer_init initialisiert wird. Dabei geben Sie die Zeitquelle (`CLOCK_REALTIME` oder `CLOCK_MONOTONIC`) an. Weiterhin benötigen Sie eine Callback-Funktion, die aufgerufen wird, wenn der Timer abläuft. Diese Funktion gibt am Ende »HRTIMER_NORESTART« oder »HRTIMER_RESTART« zurück, je nachdem, ob das Tasklet nur einmal aufgerufen werden soll oder – ohne erneute äußere Aktivierung – mehrfach. Falls die Callback-Funktion »HRTIMER_RESTART« zurückgibt, dürfen Sie jedoch nicht vergessen, vorher den Auslösezeitpunkt beispielsweise über die Funktion hrtimer_add_expires_ns neu zu setzen.

Aktivierung

Den Timer selbst aktivieren Sie, indem Sie die Funktion tasklet_hrtimer_start aufrufen. Die Zeit, zu der die Callback-Funktion aufgerufen wird, geben Sie dabei in der Form von ktime_t an. Sie haben die Möglichkeit, diese Zeit absolut (`HRTIMER_MODE_ABS`) oder relativ (`HRTIMER_MODE_REL`) anzugeben.

Aufräumen

Beim Entladen des Moduls stellen Sie durch Aufruf von tasklet_hrtimer_cancel noch sicher, dass keine durch das Modul aktivierte Callback-Funktion im System verbleibt.

Beispiel 6-14 zeigt Ihnen die Verwendung der Funktionen.

Beispiel 6-14
High Resolution Timer per Tasklet aufgerufen

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

static struct tasklet_hrtimer fireup;
```

```
static enum hrtimer_restart hrtimer_func( struct hrtimer *hrt )
{
    printk(KERN_INFO "hrtimer_func called at %ld...\n", jiffies);
    return HRTIMER_NORESTART;
}

static int __init mod_init(void)
{
    static ktime_t start_in;

    printk(KERN_INFO "mod_init called at %ld\n", jiffies);
    tasklet_hrtimer_init( &fireup, hrtimer_func, CLOCK_REALTIME,
        HRTIMER_MODE_REL );
    start_in = ktime_set( 10, 0 );
    tasklet_hrtimer_start( &fireup, start_in, HRTIMER_MODE_REL );
    return 0;
}

static void __exit mod_exit(void)
{
    printk(KERN_INFO "mod_exit called\n");
    tasklet_hrtimer_cancel( &fireup );
}

module_init( mod_init );
module_exit( mod_exit );

MODULE_LICENSE("GPL");
```

6.4 Kernel-Threads

Ein Kernel-Thread entspricht einem Thread auf User-Ebene mit dem Unterschied, dass er komplett im Betriebssystemkern abgearbeitet wird. Daher benötigt er im Userspace weder Code- noch Datensegmente. Ansonsten aber ist er wie jeder andere Rechenprozess im System durch eine Task-Struktur repräsentiert und erscheint beim Aufruf des Konsolen-Kommandos »ps awxu« in der Prozesstabelle, wo er anhand der eckigen Klammern (»[]«) als Kernel-Thread identifiziert werden kann. Oft beginnt der Name des Kernel-Threads auch mit dem Buchstaben »k«. Der Abarbeitungszeitpunkt des Kernel-Threads wird folglich durch den Scheduler festgelegt.

Besonderheiten

Da Kernel-Threads im Kernellevel ablaufen, bekommen sie die CPU so lange zugeteilt, bis sie sich schlafen legen (wait_event oder wait_event_interruptible) oder beenden. Anders als Tasklets und Timer können und müssen sich Kernel-Threads also schlafen legen. Andernfalls nehmen