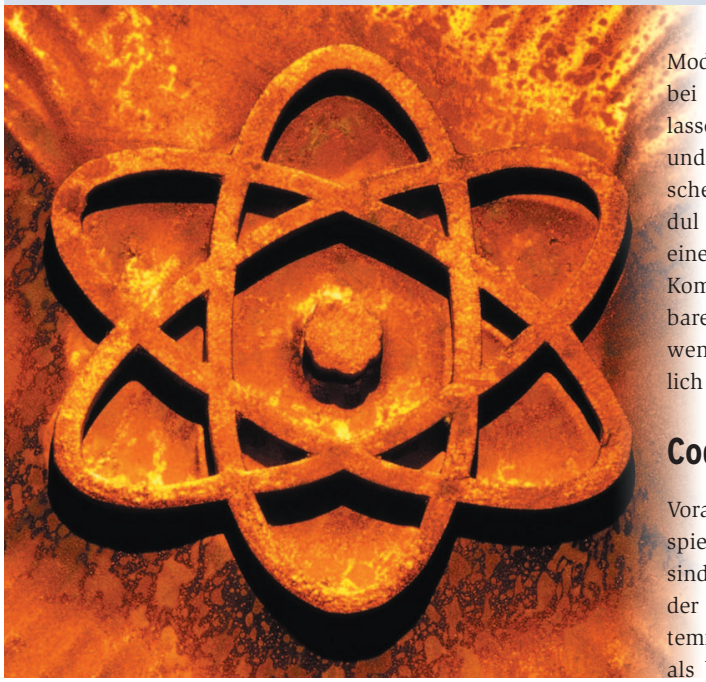
Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6

Kern-Technik

Die neue Serie „Kern-Technik“ untersucht den kommenden Linux-Kernel 2.6 und seine Bestandteile. Wer Treiber programmieren, Kernelfunktionen verändern oder einfach die Vorgänge im Inneren von Linux verstehen will, der findet hier einen praktischen Einstieg. Eva-Katharina Kunst, Jürgen Quade



Kernel- und Treiberprogrammierung ist das Thema der neuen Kern-Technik-Serie: Wie schreiben künftige Kernel-Gurus ihr erstes Modul, etwa einen zeichenorientierten Gerätetreiber für Kernel 2.6? Welche Anpassungen sind erforderlich, um einen 2.4-Treiber auf 2.6 zu portieren? Wie realisiert der neue Kernel Hardwarezugriffe? Und welche Werkzeuge und Techniken sind zur Treiberprogrammierung nötig?

An kurzen, abgeschlossenen Beispielen beschreiben die Artikel Technik, Hintergründe und Anwendung. Wer C kennt, mit Betriebssystemen vertraut ist und Basiskenntnisse über Computerhardware besitzt, kann sofort mit den aktuellen Kernelquellen loslegen. Die Struktur der Module, das Kernel-Build-System [3] sowie das Laden und Entladen von Modulen unter der Kernelgeneration 2.6 sind das Thema der ersten Folge.

Module sind die übliche Form bei Kernelerweiterungen. Sie lassen sich zur Laufzeit laden und wieder entladen. Der entscheidende Vorteil: Das Modul belegt – im Gegensatz zu einer statisch eingebundenen Komponente – nur dann kostbare Ressourcen im Kernel, wenn es unbedingt erforderlich ist.

Code im Kernel

Vorab eine Warnung: Die Beispiele in dieser Artikelserie sind Kernel-Code, also Code, der Zugriff auf sämtliche Systemressourcen hat. Anders als bei einer Anwendung ist es sehr einfach möglich, das gesamte System – nicht nur eine Applikation – zum Absturz zu bringen. Programmierer sollten daher sehr sorgfältig vorgehen, auf so manchen Reboot müssen sie sich dennoch einstellen.

Beim Programmieren sind kaum Unterschiede zwischen dem Entwickeln von Anwendungssoftware und Kernelerweiterungen (zum Beispiel Treiber) festzustellen. Bemerkenswert ist, dass der Entwickler kein eigenständiges Stück Software schreibt, sondern eine zusätzliche Komponente für ein bestehendes und sogar laufendes Programm, den Kernel. Da muss der schnelle Griff zu Funktionen aus der C-Standardbibliothek unterbleiben. Gleitkomma-Operationen sind generell tabu.

Die Kernelentwickler haben viel getan, um die Unterschiede zwischen Kernel- und Anwendungsprogrammierung unsichtbar werden zu lassen. Dass Funktionsaufrufe vielfach Makros sind oder sich dahinter gar Assembler-Code verbirgt, erfährt der Entwickler erst, wenn er in Kernelcode und Headerdateien stöbert. Zum Programmierstil sei auf das sehr lesenswerte Dokument „Linux kernel coding style“ [4] verwiesen; es ist auch in den Kernelquellen unter »Documentation/CodingStyle« zu finden. Insbesondere der Anfang könnte dem einen oder anderen ein Lächeln entlocken. Dass im Kernel Goto-Aufrufe erlaubt und sogar erwünscht sind, mag befremdlich wirken – eine spätere Folge wird dazu mehr erklären.

Starthilfe

Um loslegen zu können, müssen ein aktueller Entwicklerkernel und neue Modutils installiert sein (siehe Kasten „Entwicklerkernel installieren“). Neben dem Kernelimage benötigt jeder Programmierer den zugehörigen Quellcode und – das ist bereits neu – auch die Kernelkonfiguration. Ein heikler Punkt, denn bisher genügten Headerdateien und ein selbst geschriebenes Makefile. Die Konfiguration ist der Preis, der für die außergewöhnliche Skalierbarkeit und Portierbarkeit zu zahlen ist: Der Compiler benötigt unterschiedliche Flags, abhängig davon,

- für welche Plattform er den Kernel übersetzen soll,
- ob der Kernel für eine Ein-Prozessor- oder eine Mehr-Prozessor-Maschine gedacht ist und
- ob innerhalb des Kernels Unterbrechungen (Preemption) zugelassen sind.

Dieses Wissen ist im so genannten Kernel-Build-System hinterlegt. Diese in 2.5

komplett überarbeitete Systemkomponente ist für das Generieren des Kernels und seiner Module zuständig. Um sich darin einzuklinken, muss das Makefile den Pfad zu den konfigurierten Kernelquellen kennen. Linus Torvalds hat festgelegt, dass die Quellen über das Module-Verzeichnis zu finden sein sollen: »/lib/modules/*Version*/build/«. Meist ist dies ein Symlink auf das bekannte Directory »/usr/src/linux-*Version*«.

Modul-Interna

Der Aufbau eines Moduls ist seit Kernel 2.4 im Wesentlichen unverändert. Der Entwickler muss nach wie vor je eine Funktion zur Modul-Initialisierung (mit Namen »init_module«) sowie eine zur Deinitialisierung (»cleanup_module«) schreiben. Beim Laden des Moduls mit »insmod« oder »modprobe« wird »init_module« aufgerufen.

War die Initialisierung erfolgreich, gibt die Funktion »0« zurück. Andernfalls returniert sie einen negativen Wert, beispielsweise »-EBUSY«. »EBUSY« ist eine von vielen Fehlerkonstanten, die in den Headerdateien »asm/errno.h« beziehungsweise »asm-generic/errno-base.h« spezifiziert sind. Die Funktion »cleanup_module« hat hingegen keinen Rückgabewert. Die Prototypen beider Funktionen befinden sich in der Headerdatei »linux/module.h«.

Für ein minimalistisches Modul (siehe Listing 1) fehlen nur noch eine Versionsinformation und die Lizenzform. Um die Versionsnummer einzubinden, genügt es, die Headerdatei »linux/version.h« zu inkludieren. Die Modullizenz lässt sich mit dem Makro »MODULE_LICENSE« angeben, es nimmt den Namen der Lizenz als Parameter entgegen. Der Grund für diesen Schritt ist, dass der Linux-Kernel unter der GPL steht. Diese verlangt, dass der Entwickler bei erweiterten oder modifizierten Programmen auch den Quellcode veröffentlicht. Kernelmodule liegen dabei in einer Grauzone, schließlich muss man den Kernel für ein Modul nicht neu kompilieren.

Die Lizenzangabe im Modul soll hier Klarheit schaffen. Der Kernel wertet diese auch aus: Ein Modul, das unter der GPL oder der BSD-Lizenz steht, wird durch den Linux-Kernel respektive die Kernelentwickler uneingeschränkt unterstützt. Ein Modul mit proprietärer Lizenz markiert der Kernel dagegen als „tainted“ (verdorben). In diesem Fall darf man bei der Fehlersuche nicht auf die Hilfe der Kernelhacker hoffen. Auch darf ein solches Modul nicht auf sämtliche, insbesondere nicht auf neuere Kernelfunktionen zugreifen.

Kernel-Build-System

Um das Modul zu übersetzen, benötigt das Kernel-Build-System den Namen der Quellcodedatei. Diesen Namen ermittelt es anhand der Variablen »obj-m« im Makefile. Wer mehrere Module übersetzen will, initialisiert diese Variable mit den Namen aller Module. Das Makefile muss exakt den Namen »Makefile« tragen, wobei strikt auf Groß- und Kleinschreibung zu achten ist. Zudem muss es im selben Verzeichnis wie der Quellcode liegen. Falls die Quelldatei »mod1.c« heißt, besteht das Makefile nur aus der folgenden Zeile:

```
obj-m   := mod1.o
```

Das Kernel-Build-System kann nun mit seiner Arbeit beginnen. Es lässt sich per Make-Kommando mit drei Parametern starten: dem Pfad zu den Kernelquellen, dem Verzeichnis mit dem Modulcode und dem Makefile sowie mit dem Target. Das Target entspricht dem Make-Kommando, das das Build-System ausführen

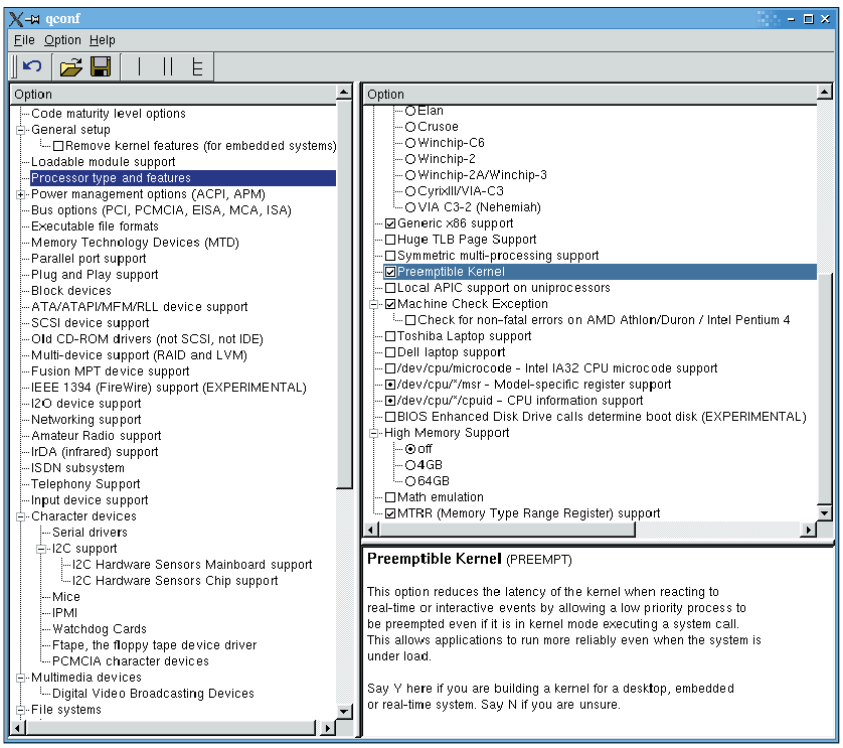


Abbildung 1: Kernel 2.5.x zeigt bei »make xconfig« ein neues GUI. Es ist mit Qt realisiert und deutlich übersichtlicher als die bisherige Oberfläche.

Listing 1: Minimalistisches Modul

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03
04 MODULE_LICENSE("GPL");
05
06 int init_module(void)
07 {
08         return 0;
09 }
10
11 void cleanup_module(void)
12 {
13         return;
14 }
```

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := mod1.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 all:
09         $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
10 endif
```

soll, im vorliegenden Fall also »modules«. Für den Kernel 2.5.70 lautet der Aufruf:

```
make -C /lib/modules/2.5.70/build ➥
  SUBDIRS=`pwd` modules
```

Möglicherweise ist in dem Kommando noch der Pfad zum Kernelquellcode anzupassen. Zudem muss man Make von jenem Verzeichnis aus aufrufen, in dem sich das Makefile und der eigene Quellcode befinden. Wenn alles stimmt, flitzen jetzt viele Ausgaben über den Bildschirm.

Makefile für Bequeme

Findige Programmierer geben sich mit einem derart langen Kommando nicht zufrieden, es jedes Mal eingeben zu müssen, nur um ein Modul zu generieren, ist unnötig umständlich. Listing 2 zeigt ein Makefile, das Abhilfe schafft. Damit genügt ein »make«-Aufruf ohne weitere Parameter.

Dieses Makefile bemerkt, ob es durch das Kernel-Build-System aufgerufen wurde oder direkt vom User. In letzterem Fall ist die Variable »KERNELRELEASE« nicht gesetzt, sodass der zweite Teil des Makefile aktiviert ist. Es feuert die Default-Regel mit Namen »all«, die mit den notwendigen Parametern das Build-System startet (Zeile 9). Dies lädt wiederum das Makefile, nur ist jetzt die Variable »KERNELRELEASE« nicht mehr leer, sondern mit einem Wert initialisiert. Folglich setzt das Makefile die Variable »obj-m« mit den Namen der zu generierenden Module.

Das Makefile in Listing 2 geht davon aus, dass der aktuelle Kernel auch jener Kernel ist, für den es das Modul erstellen soll. Es verwendet nämlich das Kommando »uname -r«, um die Kernelversion zu ermitteln, die es in den Pfad zu den Kernelquellen einsetzt. Sollten die Kernelquellen nicht unter »/lib/modules/*Kernelversion*/build« zu finden sein, dann ist Zeile 5 anzupassen.

Das Kernel-Build-System ruft die notwendigen Werkzeuge auf, insbesondere den Compiler. Damit erzeugt es die Objektdatei »mod1.o« und linkt sie mit dem Kernelversions-File »init/vermagic.o« zum Modul »mod1.ko«. Die Zeit, in der Module die Erweiterung ».o« hatten, ist mit 2.5 vorbei. Ab jetzt lautet die Erweiterung ».ko« für Kernel Object.

Mit Root-Rechten lässt sich das neue Modul wie gewohnt laden: »insmod mod1.ko«. Diese Operation kann nur gelingen, wenn die Entwicklerversion des Kernels aktiv ist. Schlägt das Kommando fehl, gibt Insmod eine entsprechende Fehlermeldung aus. Bei Erfolg sollte

Entwicklerkernel installieren

Alle nötigen Werkzeuge, im Wesentlichen Compiler und Make, sind in den meisten Distributionen bereits enthalten. Der Entwickler muss nur sicherstellen, dass er die richtigen Versionen benutzt; genaue Angaben stehen in den Kernelquellen bei »Documentation/Changes«. Beim Compiler sollte es nach offiziellen Angaben GCC 2.95.3 sein. Torvalds gibt auch für Make eine Versionsnummer vor: GNU Make 3.78. Meist sind neuere Versionen unkritisch, nur die neuesten Releases sind gelegentlich ungeeignet. Bei den Autoren haben sich GCC 3.2.3 und GNU Make 3.80 bewährt.

Alles neu: Das Kernel-Build-System

Der Quellcode des aktuellen Linux-Kerns steht auf [1] bereit, im Folgenden dient 2.5.70 [2] als Beispiel. Um aus 32 MByte Archiv knapp 220 MByte Quellcode in »/usr/src/linux-2.5.70« abzulegen, benötigt der Rechner einige Zeit.

```
cd /usr/src
tar xvfj /Downloads/linux-2.5.70.tar.bz2
```

Trotz komplett überarbeiteter Makefiles (das neue Build-System) lässt sich der Kernel wie üblich über »make menuconfig« oder »make xconfig« konfigurieren. Letzteres benötigt jetzt auch Qt (Abbildung 1) – je nach installierter Compilerversion kann das durchaus eine Stolperfalle sein. Nur wenn die Qt-Bibliotheken mit einer passenden Compilerversion übersetzt wurden, lässt sich das Programm »qconf« generieren. Grund ist eine geänderte Namenscodierung für C++-Methoden.

Trotz der vielen neuen Optionen ist die Konfiguration übersichtlicher geworden. Da bei einem Entwicklerkernel oft einige Subsysteme nicht stabil arbeiten, ist man gut beraten, anfangs nur die wirklich nötigen Komponenten auszuwählen.

Geduld beim Übersetzen

Der Aufruf von »make bzImage« startet wie gewohnt den Compile-Vorgang. Jetzt ist Geduld angesagt, denn selbst bei einem schnellen Rechner dürften gut fünf Minuten vergehen, bis der Kernel generiert ist. Bricht der Compile-Vorgang mit einem Fehler ab, ist sehr wahrscheinlich eine einzelne Komponente der Auslöser. Die einfachste Lösung ist es, sie in der Konfiguration zu deaktivieren. Wer eine andere Ursache vermutet, sollte einen Blick in die »README«-Datei und in »Documentation/Changes« werfen.

Nach dem Generieren ist der Kernel nach bewährtem Muster zu installieren, etwa durch Kopieren in das Verzeichnis »/boot«. Danach gilt es, die Konfiguration des Bootmanagers (hier Lilo) anzupassen:

```
cp arch/i386/boot/bzImage /boot/linux-
2.5.70
vi lilo.conf
lilo
```

Im Anschluss kommen die Module an die Reihe, generieren und installieren:

```
make modules
make modules_install
```

Der wohl schwierigste Teil ist das Installieren der neuen Modutils. Kernel 2.5 und damit auch Kernel 2.6 benötigen eigene Versionen, die auf der Kernel-Webseite von Rusty Russell [5] zu holen sind. Zwei Dinge sind zu beachten: Die neuen Versionen sollen die alten nicht ersetzen, sondern zusätzlich installiert werden. Außerdem hat sich das Format der Konfigurationsdatei »modules.conf« geändert.

Neue Modutils nötig

Wie die neuen Modutils (etwa »module-init-tools-0.9.12.tar.bz2«) zu installieren sind, ist in der »README«-Datei gut beschrieben. Per Default landen sie in »/usr/local/sbin«; das zweite Kommando erzeugt in diesem Verzeichnis auch Symlinks zu den Originalversionen:

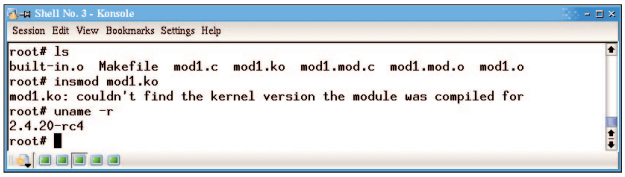
```
./configure
make links
make
make install
```

Für Debian (Testing und Unstable) gibt es die Modutils bereits als fertiges Paket »module-init-tools«, das sich mit »apt-get« einfach installieren lässt. Unter einem Kernel 2.4 würden die neuen Programmversionen automatisch die alten Varianten aufrufen, andernfalls laden sie die neuen Module (Endung ».ko« statt ».o«).

Zuletzt ist noch die Konfigurationsdatei anzupassen. Die neue »modprobe«-Version sucht die Konfiguration nicht mehr in »/etc/modules.conf«, sondern in »/etc/modprobe.conf«. Diese Datei ist schnell erzeugt:

```
./generate-modprobe.conf /etc/modprobe.conf
```

Wer das Device-Filesystem (Devfs) verwendet, sollte die Datei »modprobe.devfs« aus den Modutils-Init-Tools in das Verzeichnis »/etc« kopieren.



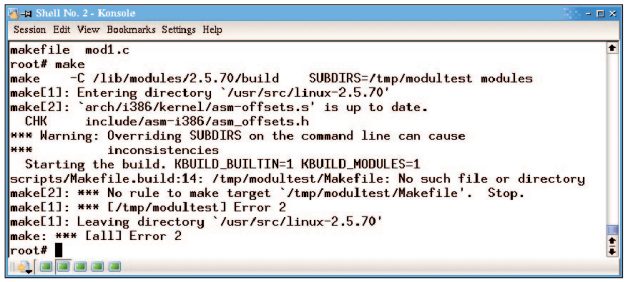


Abbildung 2 (oben): Kernelmodule aus Version 2.6 lassen sich nicht in Kernel 2.4 laden. Wer es trotzdem versucht, stößt auf die abgebildete Warnung.
Abbildung 3 (rechts): Weil das Makefile den falschen Namen trägt, wird es vom Kernel-Build-System nicht gefunden.

»lsmod« nun unter anderem das Modul »mod1« anzeigen:

```
Module       Size  Used by
mod1         1152  0
3c59x       34760  1
```

Die Liste wird in der Regel noch deutlich länger sein.

Bessere Vorlage

Das minimalistische Modul aus Listing 1 ist kaum als Basis für weitere Entwicklungen geeignet. Gerade Entwickler von eingebetteten Systemen benötigen Kernelerweiterungen, beispielsweise Treiber, die direkt in den Kernel integriert sind und nicht erst zur Laufzeit geladen werden. Diese so genannten Kerneltreiber unterscheiden sich von den Modultreibern im Wesentlichen durch unterschiedliche Namenskonventionen für die Initialisierungs- und Deinitialisierungsfunktionen. Während die Namen bei Modultreibern festgelegt sind, kann sie der Programmierer eines Kerneltreibers frei wählen.

Diese Verschiedenheit lässt sich durch zwei Makros aufheben. Der Entwickler gibt beiden Funktionen einen beliebigen Namen und ruft – meist am Ende des Moduls – das Makro »module_init« auf, dem er den Namen der Initialisierungsfunktion übergibt. Ebenso verfährt er mit dem Makro »module_exit« für die Deinitialisierungsfunktion. Listing 3 zeigt ein Gerüst, das sich als Basis für Kernelerweiterungen eignet.

Das Codestück verwendet zwei neue Direktiven: »__init« und »__exit«. Diese kernelspezifischen Schlüsselwörter sind in der Headerdatei »linux/init.h« definiert. Sie optimieren den Speicherverbrauch, indem sie dem Compiler zeigen, dass es sich um Initialisierungs- oder Deinitialisierungs-Code handelt. Der Initialisierungs-Code wird genau ein Mal durchlaufen, danach nie wieder. Es wäre Ressourcenverschwendung, diesen Programmabschnitt die ganze Laufzeit über im Speicher zu halten. Zumindest bei Kerneltreibern entfernt Linux diese Teile, sobald sie abgearbeitet sind.

Soviel zum Einstieg in den neuen Linux-Kernel. Die nächste Folge wird zeichenorientierte Gerätetreiber unter Kernel 2.5 respektive 2.6 vorstellen. *(fjl)* ■

Erste Hilfe

Scheitert das Generieren oder Laden des Moduls, dann ist Fehlersuche angesagt – die folgenden Hinweise helfen dabei, die Ursache einzugrenzen. Ein Fehler kann sich allerdings auf verschiedenen Plattformen unterschiedlich auswirken.

Symptom: »mod1.ko: couldn't find the kernel version the module was compiled for« (siehe Abbildung 2). **Fehler:** Es wurde versucht, ein Modul eines Kernels der nächsten Generation unter einem stabilen Kernel (zum Beispiel 2.4.20) zu laden. **Abhilfe:** Booten Sie einen Entwicklerkernel.

Symptom: »/tmp/modultest/Makefile: No such file or directory« (siehe Abbildung 3). **Fehler:** »Makefile« ist falsch geschrieben. **Abhilfe:** Korrigieren Sie den Namen des Makefile. Der erste Buchstabe muss groß geschrieben werden, die weiteren Buchstaben klein.

Symptom: »insmod: QM_MODULES: Function not implemented«. **Fehler:** Sie haben versucht, das Modul mit einer alten Version von »insmod« zu laden. **Abhilfe:** Installieren Sie die neue Version der Modutils (siehe Kasten „Entwicklerkernel installieren").

Symptom: Im Syslog taucht die Meldung auf: »mod1: no version magic, tainting kernel«. **Fehler:** Es handelt sich nur um eine Warnung, der Kernel hat das Modul dennoch geladen. Er konnte aber die Version nicht prüfen, da er das Modul nicht als »mod1.ko«, sondern als »mod1.o« erhielt. **Abhilfe:** Laden Sie das Modul »mod1.ko«.

Symptom: Im Syslog taucht die Meldung auf: »mod1: version magic '2.5.70 preempt PENTIUMII gcc-2.95' should be '2.5.70 preempt PENTIUMII gcc-3.2'«. **Fehler:** Hierbei handelt es sich um eine Warnung, der Kernel hat das Modul dennoch geladen. Der Kernel ist mit GCC 3.2, das Modul jedoch mit GCC 2.95 übersetzt worden. **Abhilfe:** Installieren Sie GCC 3.2 als Default-Compiler.

Infos:

[1] Kernelquellen: [http://www.kernel.org]
[2] Kernel 2.5.70: [http://www.kernel.org/pub/linux/kernel/v2.5/linux-2.5.70.tar.bz2]
[3] Kernel-Build-System: [http://lxr.linux.no/source/Documentation/kbuild/?v=2.5.56]
[4] Linux kernel coding style: [http://lxr.linux.no/source/Documentation/CodingStyle?v=2.5.56]
[5] Neue Modutils: [http://www.kernel.org/pub/linux/kernel/people/rusty/modules/]
[6] Weitere Infos siehe Kernel-Schwerpunkt im Linux-Magazin 4/03

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, gehören seit den Anfängen von Linux zu den Fans von Open Source.

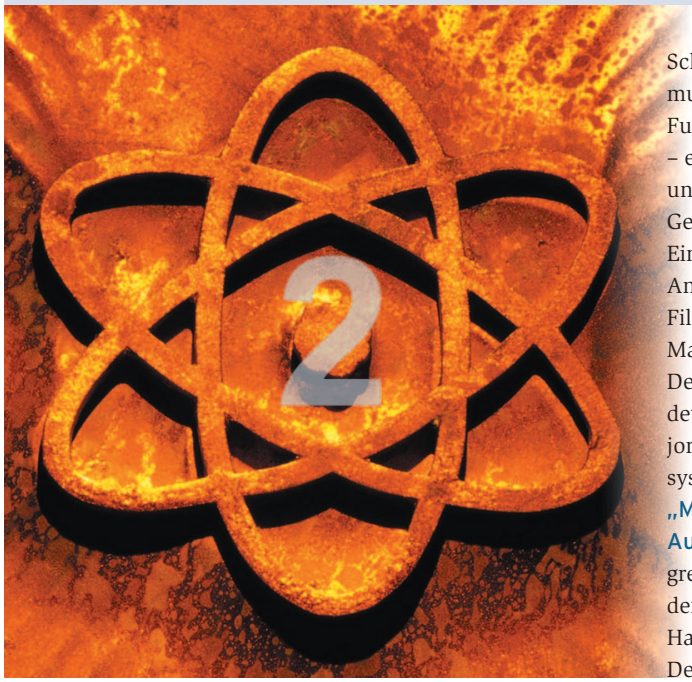
Listing 3: Code-Gerüst

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04
05 MODULE_LICENSE("GPL");
06
07 static int __init ModInit(void)
08 {
09         return 0;
10 }
11
12 static void __exit ModExit(void)
13 {
14         return;
15 }
16
17 module_init( ModInit );
18 module_exit( ModExit );
```

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 2

Kern-Technik

Die zweite Folge der Serie „Kern-Technik“ beschreibt zeichenorientierte Gerätetreiber im Linux-Kernel 2.6. Mit etwas Hintergrundwissen ist es leicht, die Daten von der Applikation über die Gerätedatei und deren Major-Nummer bis zum Kernelmodul und wieder zurück zu senden. Eva-Katharina Kunst, Jürgen Quade



Der erste Artikel der Kern-Technik-Serie hat gezeigt, wie sich der künftige Kernel 2.6 mit Modulen erweitern lässt **[1]** – die Änderungen zum Kernel 2.4 liegen meist im Detail. Der zweite Teil erklärt nun, wie aus einem Modul ein zeichenorientierter Gerätetreiber wird.

Reale und virtuelle Geräte

Ein Treiber ist lediglich eine Sammlung von Funktionen, die den Zugriff auf eine Hardware steuern. Es muss sich dabei nicht immer um reale Hardware handeln: Logische und virtuelle Geräte wie »/dev/null« oder »/dev/zero« sind ebenfalls über Treiber realisiert.

Eleganterweise präsentieren sich die Geräte gegenüber den Applikationen als Dateien, genauer als Gerätedateien (Device files). Elegant deshalb, weil der Anwendungsprogrammierer keine neue Schnittstelle (API) lernen muss, sondern die bekannten Funktionen zum Dateizugriff – etwa »open«, »close«, »read« und »write« – auch für den Gerätezugriff verwendet.

Eine Gerätedatei sieht für den Anwender wie ein normales File aus, mit dem jedoch eine Major-Nummer assoziiert ist. Der zugehörige Treiber meldet sich mit der gleichen Major-Nummer beim Betriebssystemkern an (siehe Kasten „Major-Nummern in zweiter Auflage“). Die Applikation greift über die Gerätedatei auf den Treiber und damit auf die Hardware zu.

Der Modulentwickler ruft im einfachsten Fall innerhalb der Modul-Initialisierungsfunktion (Zeile 17 bis 23 in Listing 1) die aus früheren Kerneln bekannte Funktion »register_chrdev« auf (Zeile 19). »chrdev« (Character device) kennzeichnet den Gerätetreiber als zeichenorientiert, der im Gegensatz zu blockorientierten Geräten Zeichen- oder Byte-weise arbeitet.

Zeichen oder Block

Die Unterscheidung der Treiber ist bei Unix-Systemen Tradition. Treiber für Festplatten, Bandlaufwerke, Disketten oder eine RAM-Disk verschieben die Daten immer in Blöcken. Zeichenorientierte Treiber gehen dagegen von einem zeichenweise ankommenden Datenstrom aus. Zu dieser Gruppe zählen die Tastatur, der Sound oder auch die serielle Schnittstelle. Eigenentwicklungen fallen meist in diese Kategorie.

Die Funktion »register_chrdev« hat drei Parameter: die Major-Nummer, den Treibernamen und eine Datenstruktur vom Typ »struct file_operations«. Hat sich der Treiber erfolgreich beim Kernel angemeldet, gibt die Funktion den Wert »0« zurück, andernfalls einen Fehlercode.

Rollenverteilung

Die als dritter Parameter übergebene Datenstruktur »Fops« spielt im weiteren Ablauf eine wichtige Rolle. Der Treiber muss in dieser Datenstruktur die Adressen von Funktionen hinterlegen, die der Kernel aufrufen soll, sobald die Applikation eine Interaktion mit dem Treiber verlangt (siehe Abbildung 1).

Öffnet ein User-Space-Programm die Gerätedatei (»open«-Aufruf in der Applikation), liest der Kernel aus der »Fops«-Struktur die Adresse der korrespondierenden Funktion und ruft sie auf – im vorliegenden Fall »DriverOpen«. Der Aufruf von »read« triggert über »Fops.read« die Funktion »DriverRead«, »write« triggert »DriverWrite« und »close« schließlich »DriverClose«. Es gibt noch mehr Funktionen, etwa »ioctl« und »select«, für die ebenfalls Treiberfunktionen vorgesehen sind.

Der Code in Zeile 14 (Listing 1) initialisiert die »Fops«-Datenstruktur mit der Treiberfunktion »DriverOpen«. Es ist unproblematisch, nur einen Teil der möglichen Funktionen bereitzustellen. Hat ein Funktionszeiger der Struktur den Wert »NULL«, dann ruft der Kernel eine Ersatzfunktion auf, die für die Anwendung einen Fehlercode erzeugt.

Vielleicht ist die Syntax zur Initialisierung von Strukturelementen über den Namen etwas fremd. Die hier darge-

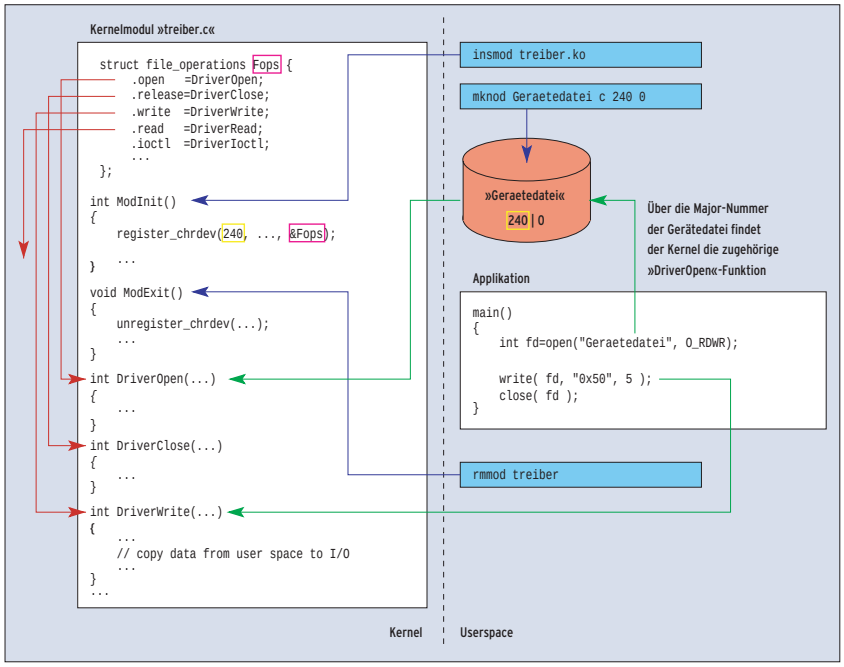


Abbildung 1: Der Zugriff einer Userspace-Applikation (rechts) auf einen Treiber (links) läuft über eine Gerätedatei: Der »open«-Aufruf öffnet die Gerätedatei; ihre Major-Nummer entscheidet, welcher Treiber zum Zuge kommt. Der Kernel ruft dann die passende Funktion auf – laut »Fops«-Struktur ist »DriverOpen« zuständig. Blau hinterlegt sind die Kommandos aus dem Modutils-Paket.

Major-Nummern in zweiter Auflage

Die Major-Nummer stellt die Beziehung zwischen der im Dateisystem angelegten Gerätedatei und dem Treiber her. Die Major-Nummer in Unix-Systemen ist traditionell ein 8-Bit-Wert, der 256 Treiber referenzieren kann. Ohne die fest belegte Major-Nummer 0 bleiben 255 Treiber übrig. Jeder von ihnen kann 256 Geräte bedienen: Hier kommt die ebenfalls 8 Bit breite Minor-Nummer ins Spiel, die ein zweites Merkmal der Gerätedatei ist. Zwei 8-Bit-Nummern sind nach dem Geschmack der Entwickler von IBM & Co. aber viel zu wenig, sie wollen 4000 Platten (und mehr) an ihre Linux-Box anschließen.

32 statt 8 Bit für die Major-Nummer

Linus Torvalds hat schließlich die Major-Nummer für den Kernel 2.6 erweitert. Nach dem aktuellen Stand werden 32 Bit für die Major- und 32 Bit für die Minor-Nummer zur Verfügung stehen. Damit einher geht eine weitere Änderung der Schnittstelle: Nicht mehr allein die Major-Nummer, sondern die Kombination aus Major- und Minor-Nummer wählt einen Treiber aus. Zwei Treiber können künftig dieselbe Major-Nummer verwenden, wenn sich die zugehörigen Bereiche der Minor-Nummern nicht überschneiden.
Mit den erweiterten Major- und Minor-Nummern sind die aus den vorherigen Kerneln bekannten Funktionen »register_chrdev« und »unregister_chrdev« zu Wrapper-Funktionen mutiert, die ihrerseits die neuen Funktionen »register_chrdev_region« und »unregister_chrdev_region« aufrufen.

```
int register_chrdev_region(
  unsigned int major,
  unsigned int baseminor,
  int minorct, const char *name,
  struct file_operations *fops )
int unregister_chrdev_region(
  unsigned int major,
  unsigned int baseminor,
  int minorct, const char *name )
```

Die beiden neuen Funktionen erhielten zwei zusätzliche Parameter: »baseminor« legt die erste Minor-Nummer für den Treiber fest; »minorct« gibt an, wie viele Minor-Nummern der Kernel für den Treiber reservieren soll.

Dünner Wrapper für altes Interface

Die Wrapper-Funktion »register_chrdev« besteht damit im Wesentlichen aus der folgenden Zeile Code (»fs/char_dev.c« im Kernel):

```
register_chrdev_region(
  major, 0, 256, name, fops );
```

Wer das Device-Filesystem (Devfs) benutzt, ist es gewohnt, dass der Kernel die Gerätedateien selbstständig anlegt. Der Treiber muss dies aber unterstützen – bei den Beispielen in diesem Artikel ist das nicht der Fall. Der Mknod-Aufruf ist also in jedem Fall nötig.

stellte Zuweisung ».open = DriverOpen«, ist im C99-Standard [2] festgezurrt und wird mittlerweile von allen modernen Compilern unterstützt. Sie ist übersichtlicher als die bisherige Initialisierung und vermeidet insbesondere dann Fehler, wenn der Entwickler die Deklaration einer Datenstruktur nachträglich ändert.

Zeiger auf die Details

An die »DriverOpen«-Funktion (Zeilen 8 bis 11, Listing 1) übergibt der Kernel zwei Parameter: einen Zeiger auf ein Element vom Typ »struct inode« und einen auf ein Element vom Typ »struct file«. Der erste Parameter ist der Schlüssel zu allen Informationen, die mit der Gerätedatei zusammenhängen, zum Beispiel die Major- und Minor-Nummer, Besitzer (Owner) oder Gruppe. Der zweite Parameter informiert den Treiber über den zugreifenden Rechenprozess, die so genannte Treiberinstanz. Hier ist auch gespeichert, welche Zugriffsflags (nur lesender oder nicht blockierender Zugriff) die Treiberinstanz gewählt hat.
Die Implementierung von »DriverOpen« legt fest, ob eine Applikation den Treiber nutzen darf (Rückgabewert »0«) oder nicht (»-EBUSY« oder »-EIO«). In dieser Funktion implementiert der Entwickler seine treiberspezifische Zugriffs-Policy. Zusammen mit dem minimalistischen Modul aus dem ersten Kern-Technik-Artikel und der Headerdatei »linux/fs.h« ergibt sich der in Listing 1 dargestellte einfache Treiber.

Einfach öffnen

Beim Laden des Moduls ruft der Kern die Funktion »ModInit« auf. Sie meldet das Modul als Treiber beim Betriebssystemkern unter der Major-Nummer 240 an. Natürlich darf sie nicht einfach eine beliebige Major-Nummer verwenden, nahezu alle der 255 Major-Nummern sind fest an bestimmte Treiber gebunden. Einige aber stehen zur freien Verfügung: 60 bis 63, 120 bis 127 und 240 bis 254. Mit dem neuen Kernel stehen zwar mehr Major-Nummern bereit, allerdings haben die Kernelentwickler deren Verwendung noch nicht ausdiskutiert. Dateisysteme und Systemprogramme sind daher noch nicht darauf eingestellt.

Beim Entladen des Moduls muss die Funktion »ModExit« den Treiber beim Betriebssystemkern wieder abmelden (»unregister_chrdev«). Wer das vergisst, kommt um einen Reboot nicht herum. Der Code in Listing 1 lässt sich bereits als funktionierender Treiber übersetzen und testen. Befindet sich der Quellcode in einer Datei namens »treiber.c«, muss der Entwickler im Makefile aus dem ersten Artikel [1] nur den Inhalt der Variable »obj-m« auf »treiber.o« setzen (Listing 2, Zeile 2).

Der Aufruf von »make« führt dazu, dass das Modul »treiber« generiert wird. Tritt ein Fehler auf, hilft ein Blick in den Kasten „Erste Hilfe“ im ersten Artikel [1] der Kern-Technik-Serie. Nach dem Übersetzen ist der Treiber für einen Test bereit, dazu muss er vom Root-User per »insmod treiber.ko« geladen werden. Um zu prüfen, ob das System den Treiber akzeptiert, dient das Kommando »cat/proc/devices«. Unter der Nummer 240 bei den zeichenorientierten Geräten (Character devices) sollte sich der Treiber »Treiber-Test« wiederfinden.

Lesen und schreiben

Der Treiber erfüllt noch keine sinnvolle Aufgabe. Zwar kann ihn die Applikation öffnen, doch liefert er weder Daten noch konsumiert er welche. Dazu fehlen noch eine Read-Funktion

```
static ssize_t DriverRead (
  struct file *File,
  char __user *User,
  size_t count,
  loff_t *offset )
```

und zudem eine entsprechende Write-Funktion:

```
static ssize_t DriverWrite (
  struct file *File,
  const char __user *User,
  size_t count,
  loff_t *offset)
```

Beide Funktionen besitzen je vier Parameter: der erste (»File«) ist ein Zeiger auf die Treiberinstanz. Der zweite (»User«) zeigt auf den Speicherbereich, in den die Applikation die Daten vom Treiber kopiert haben möchte (Read-Funktion) beziehungsweise aus dem der Treiber die Daten abholen soll (Write). »count« gibt an, wie viele Bytes der Treiber an die Applikation maximal übergeben darf (Read) oder wie viele Bytes er lesen kann (Write). Denkt man sich das Gerät als Speicherblock, spezifiziert »offset«, ab welcher Stelle der Treiber Daten lesen oder speichern soll. Dieser Parameter ist bei vielen zeichenorientierten Gerätetreibern ungenutzt.

»DriverRead« und »DriverWrite« geben jeweils die Anzahl der Bytes zurück, die sie in den Speicherbereich der Applikation kopiert oder aus diesem in den Kernel transferiert haben. Der Wert »0« zeigt in beiden Fällen das Ende der Datei (EOF, End of File) an: Weder stehen Daten zum Lesen zur Verfügung noch können Daten geschrieben werden.

Die Implementierung eines Geräts »/dev/null«, das alle Eingaben frisst, ist damit leicht zu bewerkstelligen: Die Write-Funktion in Listing 3 bestätigt einfach, dass sie alle Daten geschrieben hat. Ein normaler Treiber wird die Daten aber tatsächlich weiterreichen. Er könnte sie

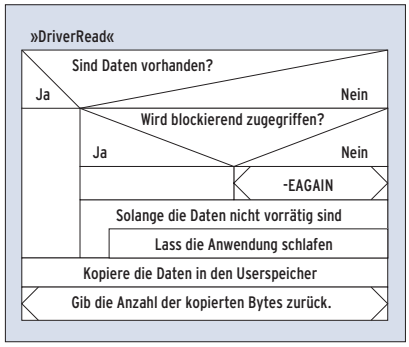


Abbildung 2: Das Struktogramm der Funktion »DriverRead«: Falls keine Daten vorhanden sind und die Applikation nicht blockierend zugreift, gibt der Treiber »-EAGAIN« zurück.

von der Hardware lesen oder in die Hardware schreiben. Wohin er sie beim Lesen kopieren soll, hat die Applikation im Parameter »User« angegeben (Adresse eines Speicherbereichs im Adressraum der Anwendung).

Kein direkter Zugriff

Der Treiber darf und kann auf diesen Speicherbereich aber nicht direkt zugreifen, der Kernel stellt daher Funktionen für den Datentransfer bereit. Die Funktionen »copy_to_user« und »copy_from_user« sind dem bekannten »memcpy« nachempfunden: Der erste Parameter ist das Ziel, zu dem die Daten zu kopieren sind, der zweite die Quelle, aus der sie stammen, der dritte bestimmt die Anzahl der zu kopierenden Bytes.

Bei »DriverRead« kommt »copy_to_user« zum Einsatz, »DriverWrite« nutzt »copy_from_user«. Da es sich um Inline-Funktionen handelt, muss der Quellcode die Headerdatei »asm/uaccess.h« einbinden, bevor er die Kopierfunktionen nutzt. Mit diesem Know-how lässt sich bereits das logische Gerät »/dev/zero« implementieren, das einer Applikation lauter Nullen zurückgibt (Listing 4).

Gerätedatei erzeugen

Vor einem Test der Treiberfunktionen muss der Entwickler eine passende Gerätedatei anlegen. Dazu dient das folgende Kommando:

```
mknod Geraetedatei c 240 0
```

Es erzeugt ein File mit Namen »Geraetedatei«, der Major-Nummer »240« und

Listing 1: Erster Treiber mit Open-Funktion

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/fs.h>
05
06 MODULE_LICENSE("GPL");
07
08 static int DriverOpen( struct inode
   *Geraetedatei, struct file *Instanz )
09 {
10     return 0;
11 }
12
13 static struct file_operations Fops = {
14     .open=    DriverOpen,
15 };
16
17 static int __init ModInit(void)
18 {
19     if( register_chrdev(240, "Treiber-Test",
   &Fops)==0 ) {
20         return 0; // Treiber angemeldet
21     }
22     return -EIO;  // Major-Nummer bereits belegt
23 }
24
25 static void __exit ModExit(void)
26 {
27     unregister_chrdev(240, "Treiber-Test");
28     return;
29 }
30
31 module_init( ModInit );
32 module_exit( ModExit );
```

der Minor-Nummer »0«. Dem Entwickler steht es frei, in welchem Verzeichnis er seine Gerätedateien anlegt. Alle für das System nötigen Gerätedateien liegen allerdings unter »/dev«. Das »c« im Aufruf steht für Character device, im Gegensatz zu »b« für Block device.

Damit der Kernel die Funktionen des Treibers aufruft, muss er von einer Applikation dazu aufgefordert werden. Applikationen, die »open«, »read« und »write« aufrufen, gibt es auf einem Unix-System zur Genüge. Die Kommandos »cat« und »echo« sind für einfache Treibertests völlig ausreichend – der Compiler darf also eine Pause einlegen.

Der Befehl »cat Geraetedatei« führt zunächst ein »open« und im Anschluss immer wieder »read« auf den Treiber aus. Da die vorliegende Treiberfunktion nie ein End of File (sprich »0«) zurückliefert, lässt sich das Kommando nur durch [Strg] + [C] oder Ähnliches abbrechen. Wer die Ausgabeumlenkung der Shell nutzt, kann mit einfachen Mitteln die Schreibfunktion (»DriverWrite«) aufrufen: »cat Geraetedatei > Geraetedatei«.

Blockieren oder nicht

Beim lesenden Datenzugriff ist noch etwas zu beachten: Was geschieht, wenn eine Applikation Daten lesen möchte, die noch gar nicht vorliegen? Normalerweise wird der Treiber die Applikation so lange schlafen legen, bis die Daten da sind. Nur wenn die Applikation den Zugriffsmodus auf »NON_BLOCKING« setzt, kehrt der Funktionsaufruf »read« direkt zurück.

Aus diesem Grund muss der Treiber den Zugriffsmodus bei jedem Aufruf der Funktionen »DriverRead« und »DriverWrite« auswerten. Diese Information befindet sich in dem Element »f_flags« der Struktur »struct file« (Treiberinstanz). Greift eine Anwendung im nicht blockierenden Modus zu und sind keine Daten vorhanden, muss ein Treiber den Fehlercode »-EAGAIN« zurückgeben:

```
if( File->f_flags & O_NONBLOCK )
    return -EAGAIN;
```

Eine auf Daten wartende Applikation wird wieder aufgeweckt, sobald der Treiber feststellt, dass die angeforderten Daten vorhanden sind. Nach dem Wecken kann er mit »copy_to_user« die Daten zur Applikation kopieren.

Fallstricke

Abbildung 2 veranschaulicht den Aufbau der Funktion »DriverRead«. Bei der Umsetzung der Struktur in Code ist jedoch Vorsicht angesagt: Normalerweise überprüft »DriverRead« erst, ob Daten vorhanden sind, ist dies nicht der Fall, legt sich die Treiberinstanz schlafen. Kommen die Daten dann, wird die Treiberinstanz beispielsweise von einer Interrupt-Serviceroutine (ISR) wieder aufgeweckt. Solche Interrupts können zu beliebigen Zeiten auftreten und die zugehörige ISR unterbricht die sonstigen Treiberfunktionen sofort.

Wenn der Interrupt genau zu dem Zeitpunkt eintrifft, an dem »DriverRead« festgestellt hat, dass keine Daten vorhanden sind, kommt das Aufwecken vor dem Schlafenlegen (siehe Abbildung 3). Eine typische Race Condition ist ein Fall, bei dem das Ergebnis einer Operation vom genauen zeitlichen Ablauf sowie vom Zufall abhängt. Wer auf ein Makro zurückgreift, kann diese Race Condition vermeiden. »wait_event_interruptible« implementiert die komplette Schleife, also sowohl das Überprüfen der Bedingung als auch das Schlafenlegen des Rechenprozesses.

Damit die Treiberinstanz später gezielt geweckt werden kann, ist zudem ein Identifikationsobjekt – eine so genannte

Portierungshilfe

Für diesen Artikel wichtige Änderungen zwischen Kernel 2.4 und 2.6:

Major-Nummern: Statt 8 Bit stehen für Major- und Minor-Nummern insgesamt 64 Bit zur Verfügung. Die Konsequenzen für den Treiberentwickler sind im Kasten „Major-Nummern in zweiter Auflage“ nachzulesen.

Schutz gegen vorzeitiges Entladen: Linux darf ein Modul so lange nicht entladen, wie Applikationen darauf zugreifen. Während in Kernel 2.4 der Entwickler mit den Makros »MOD_INC_USE_COUNT« und »MOD_DEC_USE_COUNT« einen Zähler-basierenden Entladeschutz realisieren konnte, hat Kernel 2.6 diese Aufgabe selbst übernommen. Hier ist der Zählmechanismus an die Datenstruktur »struct file_operations« gebunden. Um ihn zu nutzen, muss der Treiber das Element »owner« mit dem Wert »THIS_MODULE« initialisieren.

Direktive »__user«: Zeiger, die auf einen Speicherbereich im Adressraum einer Applikation zeigen, sind über die Direktive »__user« gekennzeichnet. Linus Torvalds entwickelt ein Werkzeug, das die korrekte Verwendung dieser Pointer verifizieren soll.

Rechenprozesse schlafen legen: Die bisherige Methode, einen Rechenprozess innerhalb des Kernels schlafen zu legen, ist im Kernel 2.6 nicht mehr zulässig. Das alte Verfahren mit den Funktionen »interruptible_sleep_on« und »sleep_on« hat zu vielen Race Conditions geführt. Für den neuen Kernel sollte ein Treiber die Funktionen »wait_event_interruptible« und »wait_event« verwenden.

Oops-Meldungen: Der neue Kernel gibt die Adressen in einer Oops-Meldung nicht nur als Hexwerte aus, sondern auch symbolisch (mit Funktionsnamen).

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := treiber.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 all:
09         $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
10 endif
```

Listing 3: Null-Write-Funktion

```
01 static ssize_t DriverWrite(
02     struct file *File, char __user *User,
03     size_t count, loff_t *offset )
04 {
05     return count;
06 }
```

Listing 4: Zero-Read-Funktion

```
01 static ssize_t DriverRead(
02     struct file *File, char __user *User,
03     size_t count, loff_t *offset )
04 {
05     unsigned char buf[1]='\0';
06
07     if( count > 0 ) {
08         copy_to_user( User, buf, 1 );
09         return 1;
10     }
11     return 0;
12 }
13
14 static struct file_operations Fops = {
15     .open = DriverOpen,
16     .read = DriverRead,
17     .write= DriverWrite,
18 };
```

Waitqueue – nötig. Dieses Objekt vom Typ »wait_queue_head_t« muss der Treiber global definieren und innerhalb der Funktion »ModInit« mit einem Aufruf von »init_wait_queue« initialisieren.

Aufgeweckt

Die Funktion »wake_up_interruptible« weckt Rechenprozesse auf. Sie erwartet als einziges Argument die Waitqueue, über die der Kernel die aufzuweckenden Applikationen identifiziert. Der schlafende Rechenprozess wacht auch auf, wenn er ein Signal erhält, etwa weil der Benutzer [Strg] + [C] drückt. Das Makro »wait_event_interruptible« kehrt daraufhin mit »-ERESTARTSYS« zurück. Meist bricht der Treiber dann von selbst den Aufruf ab, indem er keine Daten in die Applikation kopiert, sondern den Fehlercode »-ERESTARTSYS« zurückliefert. Die Applikation sollte den Aufruf dann erneut durchführen.

Wer den Rechenprozess nicht durch ein Signal wecken möchte, kann im Treiber das Funktionspaar »wait_event« und »wake_up« verwenden. Aber Vorsicht: Sehr schnell übersieht man eine Kombination von Zuständen – und der Rechenprozess wacht nicht wieder auf.

Das einfach gehaltene Codebeispiel in Listing 5 zeigt, wie sich die Zugriffsarten im Treiber realisieren lassen. Er legt einen lesenden Prozess so lange schlafen (Zeile 22), bis ein schreibender Prozess dem Treiber Daten übergibt (Zeile 42). Wenn die Funktionen »DriverOpen« und »DriverClose« keine Zugriffskriterien überprüfen und nichts initialisieren müssen, kann der Treiber – wie in diesem Beispiel – auf sie verzichten.

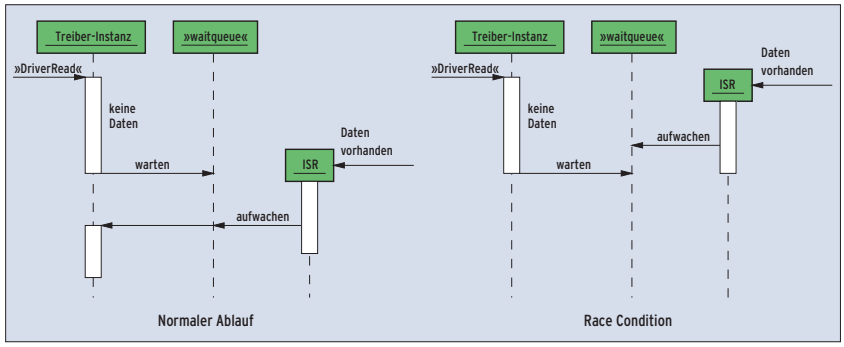


Abbildung 3: Sind keine Daten vorhanden, geht die Treiberinstanz in die Warteschleife, bis sie von der ISR (Interrupt Service Routine) wieder aufgeweckt wird (links). Weckt die ISR den Treiber zu früh (rechts), kommt es zu einem Deadlock: Das Aufwecken erfolgt vor dem Schlafenlegen.

Vorschau

Der Einstieg in die Welt der Gerätetreiber unter Kernel 2.6 ist geschafft: Mit dem vorhandenen Know-how kann man einfache virtuelle Geräte implementieren. Zum Ankoppeln realer Hardware ist jedoch weiteres Wissen nötig. Das Finden und Reservieren der zugehörigen Ressourcen sowie der Zugriff auf die Hardware ist das Thema der nächsten Ausgabe der „Kern-Technik“. *(fjl)* ■

Infos:

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 1: Linux-Magazin 8/03, S. 88

[2] Dare Obasanjo, „Are you ready for C99?“: [http://www.25hoursaday.com/C99.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

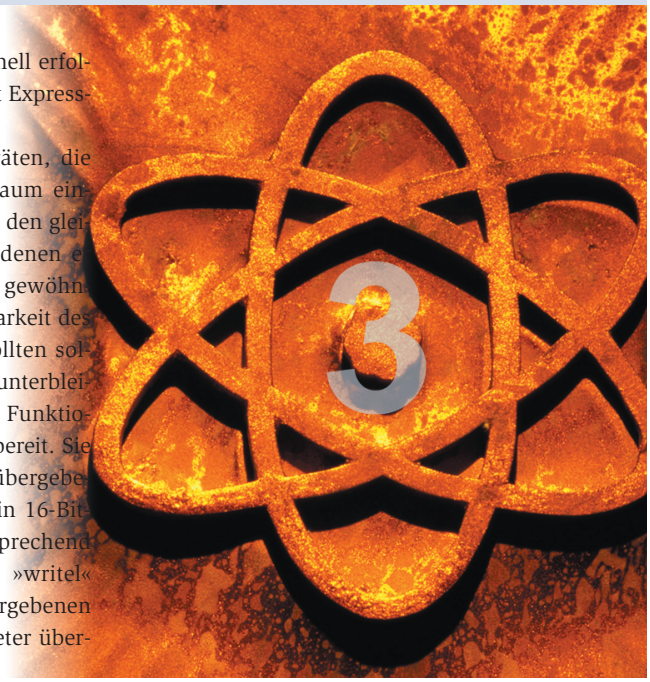
Listing 5: Beispiel Memory-Treiber

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/fs.h>    /* struct inode */
05 #include <linux/wait.h>  /*
   wait_event_interruptible */
06 #include <asm/uaccess.h> /* copy_to_user() */
07
08 MODULE_LICENSE("GPL");
09
10 static wait_queue_head_t ReadWq;
11 static int FillIndex=0;
12 static char MemBuf[256];
13
14 static ssize_t DriverRead(
15     struct file *File, char *User,
16     size_t count, loff_t *Offset )
17 {
18     if( FillIndex == 0 ) {
19         // Keine Daten vorhanden
20         if( File->f_flags & O_NONBLOCK )
21             return -EAGAIN;  // non blocking
   mode, return
22         if( wait_event_interruptible(ReadWq,
   (FillIndex>0)) )
23             return -ERESTARTSYS;  // Schleife
24     }
25     if( count>FillIndex )
26         count=FillIndex;
27     if( copy_to_user(User, MemBuf, count) )
28         return -EFAULT; // Daten nicht
   komplett kopiert
29     FillIndex -= count;
30     return count;
31 }
32
33 static ssize_t DriverWrite(
34     struct file *File, const char *User,
35     size_t count, loff_t *Offs)
36 {
37     if( count > sizeof(MemBuf) )
38         count = sizeof(MemBuf);
39     if( copy_from_user(MemBuf, User, count) )
40         return -EFAULT;
41     FillIndex=count;
42     wake_up_interruptible(&ReadWq);
43     return count;
44 }
45
46 static struct file_operations Fops = {
47     .owner=THIS_MODULE,
48     .read=DriverRead,    /* read  */
49     .write=DriverWrite,  /* write */
50 };
51
52 static int __init BufInit(void)
53 {
54     if(register_chrdev(240, "MemoryTreiber",
   &Fops) == 0) {
55         init_waitqueue_head( &ReadWq );
56         return 0;
57     }
58     return -EIO;
59 }
60
61 static void __exit BufExit(void)
62 {
63     unregister_chrdev(240, "MemoryTreiber");
64 }
65
66 module_init( BufInit );
67 module_exit( BufExit );
```

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6

Kern-Technik

Zeichenorientierte Treiber für virtuelle Geräte waren das Thema des letzten „Kern-Technik"-Beitrags. Die heutige Folge zeigt, wie der Programmierer echte PC-Hardware, insbesondere PCI-Geräte, mit dem neuen Kernel 2.6 anspricht. Eva-Katharina Kunst, Jürgen Quade



Da Hardwarekomponenten ausfallen können oder ihr Timingverhalten gewissen Unwägbarkeiten unterliegt, ist für ein auf Stabilität bedachtes Betriebssystem wie Linux der Zugriff auf Hardware-Ressourcen immer ein wenig heikel. Doch auch im Kernel 2.6 bleibt wie beim 2.4 das Programmieren eigener Hardwaretreiber simpel: Sowohl I/O-Bereich als auch der Speicheradressraum des Prozessors sind über einen Satz von Makros beziehungsweise Funktionen (siehe Abbildung 1) recht einfach ansprechbar.

Extra Befehle für portable Portzugriffe

Auf Adressen im I/O-Bereich, die Ports, greift die CPU mit separaten Ein-/Ausgabebefehlen zu. Um den Kernel portabel zu halten, kapselt er diese Befehle in Makros. Verlangt eine Applikation Daten von der Hardware, triggert die »read«-Funktion das zugehörige »DriverRead« im Treiber [1]. Erst wenn der Treiberentwickler in »DriverRead« ein spezielles Makro einfügt, kommt der Datenfluss von der Hardware zustande.

Dabei lesen die Makros »inb«, »inw«, »inl« von dem Port, den sie als Parameter übergebenen bekamen, 8, 16 oder 32 Bit. Das Schreiben auf I/O-Hardware ist Sache der Makros »outb«, »outw«, »outl«, die in »DriverWrite« unterzubringen sind. Sie leiten den als ersten Parameter übergebenen 8- (b), 16- (w) oder 32-Bit-Wert (l) an die als zweiten Parameter angegebene Portadresse.

Listing 1, Zeile 24, bedient sich einer Variante davon: »outb_p«. Das p steht für Pause. In dieser Form – es gibt sie für alle Portbefehle – wird nach jedem Zugriff eine kurze Pause eingelegt, sodass Hardwarezugriffe nicht zu schnell erfolgen. Einige Geräte kommen mit Expresstransfers nicht gut zurecht.

Auf Speicherbereiche von Geräten, die sich in den normalen Adressraum einblenden, könnte der Kernel mit den gleichen Befehlen zugreifen, mit denen er den regulären Arbeitsspeicher gewöhnlich anspricht. Um die Portierbarkeit des Kernels nicht zu gefährden, sollten solche direkte Zuweisungen aber unterbleiben. Dafür stehen nämlich die Funktionen »readb«, »readw«, »readl« bereit. Sie lesen von der als Parameter übergebenen Speicheradresse 1 Byte, ein 16-Bit- oder ein 32-Bit-Wort. Entsprechend schreiben »writeb«, »writew«, »writel« den als ersten Parameter übergebenen Wert an die im zweiten Parameter übergebene Speicherzelle.

Bit-Datentypen

Schon die Namensgebung der Makros macht klar: Beim Zugriff auf Hardware arbeitet man nicht mit den C-Typen »char«, »short« oder »long«, sondern mit Bitbreiten. Denn »int« & Co. in C sind maschinenabhängig: Für den einen Mikroprozessor beschreibt ein »int«-Datentyp ein 16-Bit-, für den nächsten Prozessor vielleicht ein 32-Bit-Wort.

Linux definiert seine Bit-Datentypen durch Inkludieren der Headerdatei »asm/types.h«. Damit stehen ihm die Typdefinitionen »u8«, »u16« und »u32« für 8, 16 und 32 Bit breite vorzeichenfreie Variablen sowie »s8«, »s16« und »s32« für 8, 16 und 32 Bit breite Variablen mit Vorzeichen zur Verfügung. Bit-Datentypen in abgewandelter Form gibt es auch für Applikationen: »u8« im Kernel wird zu »__u8« in der Applikation.

Listing 1 zeigt, wie mit den vorgestellten Makros der gute alte PC-Lautsprecher (PC-Speaker) das Tönen anfängt. Drei Portadressen bedienen den Speaker: »0x42«, »0x43« und »0x61«. Die ersten beiden gehören zum PIT-Chip, dem Programmable Interval Timer. »0x42« ist ein Datenregister zum Schreiben eines PIT-Zählerwerts. Über Port »0x43« lässt sich die Betriebsart dieses Bausteins einstellen (siehe Abbildung 2). »0xb6« steht für den Betriebsmodus drei.

In dieser Betriebsart erzeugt der PIT ein Rechtecksignal, das sich aus dem über Port »0x42« eingestellten Timerwert und »CLOCK_TICK_RATE« herleitet. Der eigentliche Timerwert ist zwar ein 2-Byte-Wert, muss jedoch Byte-weise geschrieben werden: zunächst das niederwertige, danach das höher wertige. Schließlich schaltet das Makro den Ausgang des PIT, aus dem das Rechtecksignal kommt,

zum Lautsprecher durch. Dazu setzt es die beiden niederwertigen Bits von Port »0x61« auf »1«. Tiefere Informationen zur Programmierung des PIT liefert [2].

Den neuen Treiber übersetzen und benutzen

Mit Hilfe des Makefiles aus [3] lässt sich der neue Lautsprechertreiber übersetzen und danach per »insmod« laden. Nach dem Anlegen einer Gerätedatei

```
mknod Speaker c 240 0
```

geht es endlich los. Schreibt man auf die neue Gerätedatei einen Wert

```
echo "440" > Speaker
```

wird er als Frequenz interpretiert und auf dem Speaker – nicht schön, aber eindrücklich – ausgegeben. Das Kommando für „Ruhe am Arbeitsplatz“ ist:

```
echo "0" > Speaker
```

Eine Reservierung bitte

Der kundige Leser hat längst festgestellt, dass der vorgestellte Treiber zwar funktioniert, aber Unzulänglichkeiten hat. Eine davon: Der Treiber verwendet Ressourcen, sprich Ports, ohne sie vorher reserviert zu haben, das ist in einem modernen Betriebssystem äußerst unschön und soll sofort anders werden.
Damit der Kernel Ports und andere Ressourcen korrekt verwalten kann, dürfen sie nicht mehrfach verwendet werden. Daher ist jede Kernelkomponente dazu aufgefordert, Ressourcen zu reservieren und sie – sobald die Zugriffe für absehbare Zeit abgeschlossen sind – wieder freizugeben.
Der beschriebene Speaker-Treiber hat die Reservierung der Ports wohlweislich unterlassen. Eine Genehmigung hätte er nämlich nicht bekommen, da die betroffenen Ports durch andere Subsysteme belegt und damit reserviert sind. Zum Reservieren selbst bietet der Kernel dem Programmierer ein ganzes Bündel Funktionen an:

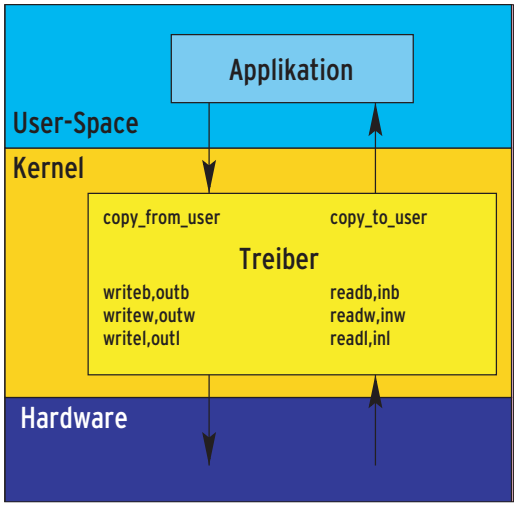


Abbildung 1: Der Kernel stellt dem Treiber eine Reihe von Funktionen bereit, die ihm dabei helfen, sowohl Daten zwischen Applikation und Treiber als auch zwischen Treiber und Hardware auszutauschen.

Für Ports

Die Prototypen sind in der Datei »linux/ioport.h« zu finden:

```
struct resource *request_region(
    unsigned long start,
    unsigned long count,
    const char *name );

void release_region(
    unsigned long start,
    unsigned long count );
```

Ports werden exklusiv für einen Treiber reserviert. Der Parameter »start« spezifiziert den Port, ab dem der Kernel »count« Ports reservieren soll. Die Reservierung wird unter der Kennung »name« geführt. Ein Rückgabewert der Funktion »request_region« ungleich null bedeutet, dass die angeforderten Ressourcen für den Treiber zur Verfügung stehen.

Für Speicheradressen

Die Prototypen verzeichnet die Datei »linux/ioport.h«:

```
struct resource *request_mem_region(
    unsigned long start,
    unsigned long count,
    const char *name );

void release_mem_region(
    unsigned long start,
    unsigned long count );
```

Listing 1: Zugriff auf Portadressen

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/fs.h>
05 #include <asm/uaccess.h>
06 #include <asm/io.h>
07
08 MODULE_LICENSE("GPL");
09
10 static ssize_t DriverWrite( struct file *File,
11     __user const char *User, size_t count,
   loff_t *Offs)
12 {
13     u16 tone;
14     u8 Save;
15     char buffer[6];
16
17     if( count > sizeof(buffer) )
18         count = sizeof(buffer);
19     copy_from_user( buffer, User, count );
20     buffer[sizeof(buffer)-1] = '\0';
21     tone = (u16) simple_strtoul( buffer, NULL,
   0 );
22     if( tone ) {
23         tone = CLOCK_TICK_RATE/tone;
24         outb( 0xb6, 0x43 ); // Mode 3:
   Generiere Rechteck
25         outb_p(tone & 0xff, 0x42);
26         outb((tone>>8) & 0xff, 0x42);
27         Save = inb( 0x61 );
28         outb( Save | 0x03, 0x61 );
29     } else {
30         outb(inb_p(0x61) & 0xFC, 0x61);
31     }
32     return count;
33 }
34
35 static struct file_operations Fops = {
36     .owner=THIS_MODULE,
37     .write=DriverWrite,
38 };
39
40 static int __init BufInit(void)
41 {
42     if(register_chrdev(240, "PC-Speaker",
   &Fops) == 0)
43         return 0;
44     return -EIO;
45 }
46
47 static void __exit BufExit(void)
48 {
49     outb(inb_p(0x61) & 0xFC, 0x61);
50     unregister_chrdev(240,"PC-Speaker");
51 }
52
53 module_init( BufInit );
54 module_exit( BufExit );
```

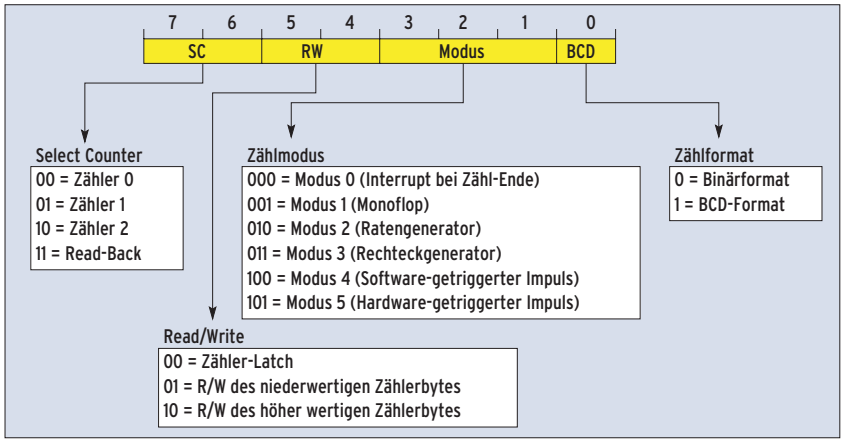


Abbildung 2: Das Steuerregister des PIT setzt sich aus mehreren Bits zusammen [2]. Zur Ansteuerung des PC-Speakers wird das Register mit 0xb6 initialisiert: Es wird Zähler 2 verwendet, beide Zähler-Bytes werden geschrieben und der Zählmodus 3 ausgewählt. Nun arbeitet der Zähler im Binärformat.

Die Parameter entsprechen denen der Portfunktionen. »start« ist die Anfangsadresse des zu reservierenden Bereichs, »count« die Anzahl der Adressen und »name« ist die Kennung, unter der der Kernel die Reservierung verwalten und identifizieren wird. Liefert die Funktion »request_mem_region« einen Wert ungleich null zurück, signalisiert das dem Treiber, dass er die angeforderten Ressourcen tatsächlich verwenden darf.

Für DMA-Kanäle

Die Prototypen sind in »asm/dma.h« zu finden:

```
int request_dma(
    unsigned int dmanr,
    const char *device_id);

void free_dma(
    unsigned int dmanr);
```

Der Parameter »dmanr« spezifiziert den angeforderten DMA-Kanal und »device_id« die zugehörige Identifikation. Ein Rückgabewert von null zeigt, dass die Reservierung erfolgreich war.

Für Interrupts

Die Prototypen sind in der Datei »linux/interrupt.h« zu finden:

```
int request_irq(
    unsigned int irq,
    irqreturn_t (*handler) (
        int, void *, struct pt_regs *),
    unsigned long irqflags,
    const char *devname,
    void *dev_id);

void free_irq(
    unsigned int irq,
    void *dev_id);
```

Interrupts sind die einzigen Hardware-Ressourcen, die der Kernel nicht zwingend an einen Treiber exklusiv vergibt. Das gemeinsame Nutzen eines Interrupts (Interrupt Sharing) ist immer dann möglich, wenn jede der betroffenen Interrupt Service Routinen (ISR) selbst feststellen kann, ob die eigene oder eine fremde Hardware die Unterbrechung ausgelöst hat.

Der Parameter »irq« gibt die Interrupt-Nummer an. »devname« ist der Name des zugehörigen Treibers. Der Parameter »dev_id« muss im System eindeutig sein. Er unterscheidet mehrere Reservierungen desselben Interrupts. Der Parameter »irqflags« ist ein Bitfeld, das – in gewissen Grenzen – das Verhalten der ISR steuert. Folgende Bits kann der Programmierer dabei kombinieren:

- »SA_SHIRQ«: Mehrere Treiber dürfen den Interrupt nutzen, da er nicht exklusiv einem einzigen Treiber zur Verfügung steht.
- »SA_INTERRUPT«: Auf dem lokalen Prozessor (das heißt, dem Prozessor, der den Interrupt gerade bedient) bleiben die Interrupts bis zum Ende der ISR gesperrt. Ist das Flag nicht gesetzt, gibt der Kernel den Interrupt sofort wieder frei.
- »SA_SAMPLE_RANDOM«: Das gesetzte Bit bewirkt, dass der Auftrittszeitpunkt des Interrupts zur Erzeugung von Zufallszahlen herangezogen wird.

Die Adresse der Interrupt Service Routine selbst wird über den Parameter »handler« übergeben. Im Gegensatz zu früheren Kerneln hat die ISR jetzt einen Rückgabewert:

- Sie gibt »IRQ_NONE« zurück, falls der ausgelöste Interrupt nicht für den Treiber bestimmt war.
- Hat die ISR den Interrupt erfolgreich bearbeitet, liefert sie »IRQ_HANDLED« zurück.
- Für das bessere Handling dieser Werte steht dem Programmierer das Makro »IRQ_RETVAL(*X*)« zur Verfügung. Dessen Rückgabewert ist »IRQ_HANDLED«, falls ihm für »*X*« ein Wert ungleich null (True) übergeben wurde. Ist dagegen der Parameter null (False) liefert das Makro »IRQ_NONE« zurück.

Das Gerät initialisieren

Die Reservierung der Ressourcen ist Teil der Geräte-Initialisierung. Dazu gehören auch Hardware-Erkennung und Hardware-Initialisierung. Eine Hardware-Initialisierung kann nur erfolgen, wenn überhaupt eine Hardware vorhanden ist. Von der Hardware selbst müssen die Adresslage und der Ressourcenumfang bekannt sein. Der Speaker im PC beispielsweise ist immer installiert und wird stets über die gleichen Portadressen angesprochen. Ergo sind diese Informationen bekannt und jeder PC-Hardware-Dokumentation entnehmbar.

Phantom PCI

Das Vorwissen über die PC-Standardkomponenten ist nicht auf jede Hardware übertragbar. Bei den schon länger vorherrschenden PCI-Geräten kann sich der Programmierer nicht mal darauf verlassen, dass sie sich zum Systemstart im System befinden, denn es sind per Linux-Definition so genannte Hotpluggable Devices. Damit werden Geräte bezeichnet, die während des laufenden Betriebes (!) hinzugesteckt und entfernt werden dürfen.

Da das PCI-Subsystem des Kernels dafür zuständig ist festzustellen, ob ein Gerät entfernt oder angesteckt wird, möchte die Routine auch die Initialisierung und Deinitialisierung des Geräts kontrollie-

```
static struct pci_device_id pci_drv_tbl[] __devinitdata = {
    { 0x10b7,        <-- Herstellerkennung
      0x5950,        <-- Gerätekennung
      PCI_ANY_ID,    <-- Subsystem-Herstellerkennung
      PCI_ANY_ID,    <-- Subsystem-Gerätekennung
      PCI_CLASS_NOT_DEFINED,   <-- Geräteklasse
      0,             <-- Maskierung für die Geräteklasse
      NULL },        <-- Treiberspezifische Daten
    {0,}             <-- Tabellenende
};
```

Abbildung 3: Der Treiber spezifiziert in der Datenstruktur vom Typ »struct pci_device_id« sämtliche Geräte, die er bedienen kann. Hier wird ein Gerät mit der Kennung »0x5950« vom Hersteller 3com »0x10b7« (»PCI_VENDOR_ID_3COM«) ausgewählt.

ren. Darum ist in PCI-Treibern die Treiber-Initialisierung (der Code in der Funktion »init_module« beziehungsweise »modules_init«) sauber von der Geräte-Initialisierung getrennt.

Der Treiber muss sich anmelden

Während der Treiber-Intialisierung meldet sich der Treiber nur beim PCI-Subsystem an. Dazu ruft er die Funktion »pci_module_init« auf, deren einziger Parameter vom Typ »struct pci_driver« ist:

```
struct pci_driver {
    struct list_head node;
    char *name;
    const struct pci_device_id *id_table;
    int  (*probe)  (struct pci_dev *dev,
        const struct pci_device_id *id);
    void (*remove) (struct pci_dev *dev);
    ...
};
```

Die wichtigsten zu belegenden Felder sind: Name des Treibers (»name«), eine Tabelle, über die der Treiber festlegt, für welche Geräte er zuständig ist (»id_table«), und jeweils eine Funktion für die Geräte-Initialisierung und -Deinitialisierung (»probe« und »remove«). Die Identifikation eines PCI-Geräts wird über insgesamt sechs Merkmale eingestellt: die Herstellerkennung, die Gerätekennung, die Hersteller- sowie die Gerätekennung für ein Subsystem, die Geräteklasse und eine Maskierung für die Geräteklasse (alle jeweils 16 Bit lang).

Die Datenstruktur vom Typ »struct pci_device_id«, die die Identifikation speichert, hat ein weiteres Element: Wie bei vielen Kernel-Datenstrukturen üblich ist hier Speicherplatz reserviert, den der Treiber für eigene Zwecke verwenden darf (Abbildung 3). Glücklicherweise muss ein PCI-Treiber nicht alle Merkmale vollständig angeben, um später ein Gerät bedienen zu können. Für alle Merkmale darf der Programmierer die Joker »PCI_ANY_ID« und »PCI_CLASS_NOT_DEFINED« einsetzen. Meist reicht es aus, die ersten beiden Merkmale explizit zu besetzen.

Findet das PCI-Subsystem ein Gerät, dessen Merkmale mit denen übereinstimmen, die die Struktur angibt, ruft es die Probefunktion – die Treiberfunktion zur Geräte-Initialisierung – auf. Sie entscheidet anhand der vom PCI-Subsystem übergebenen Informationen selbst, ob sie das Gerät bedienen möchte oder nicht. Wenn ja, muss der Treiber das Gerät mit der Funktion »pci_enable_dev« einschalten und dem PCI-Subsystem den Wert »0« zurückgeben. Jeder andere Wert heißt für das PCI-Subsystem „Such dir einen anderen Treiber“.

Das PCI-Subsystem ruft die Removefunktion des Treibers auf, wenn entweder das Gerät gerade entfernt wurde oder wenn sich der Treiber beim PCI-Subsystem mit Hilfe der Funktion »pci_unregister_driver« abmeldet.

PCI-Informationen beschaffen

Bleibt noch die Frage zu klären, wie der Treiber an die für das Gerät zu reservie-

Die sieben Komponenten eines PCI-Treibers

- »struct pci_device_id«: Die Datenstruktur spezifiziert die Geräte, für die der Treiber zuständig ist.
- »struct pci_dev«: Die Datenstruktur übergibt dem PCI-Subsystem die Treiberbeschreibung. Hierzu gehören die Gerätetabelle (»struct pci_device_id«) und die Adressen der Funktionen zur Geräte-Initialisierung und -Deinitialisierung.
- »pci_modul_init«: Mit dieser Funktion meldet sich der Treiber bei dem PCI-Subsystem an.
- Geräte-Initialisierung: Der Treiber muss eine Funktion zur Geräte-Initialisierung zur Verfügung stellen. Ihr Name ist frei wählbar. Die Adresse der Funktion bekommt das PCI-Subsystem über die Datenstruktur »struct pci_dev« übergeben. Im Rahmen dieser Funktion werden auch die notwendigen Ressourcen reserviert.
- »pci_enable_device«: Die Funktion wird im Rahmen der Geräte-Initialisierung aufgerufen. Ihre Aufgabe ist es, das Gerät einzuschalten.
- Geräte-Deinitialisierung: Der Treiber muss auch eine Funktion zur Geräte-Deinitialisierung vorhalten. Der Name der Funktion ist ebenfalls frei wählbar. Die Adresse der Funktion bekommt das PCI-Subsystem über die Datenstruktur »struct pci_dev« übergeben. Im Rahmen dieser Funktion werden die zuvor reservierten Ressourcen wieder freigegeben.
- »pci_unregister_driver«: Wird der Treiber entladen, muss er sich vom PCI-Subsystem über diese Funktion abmelden.

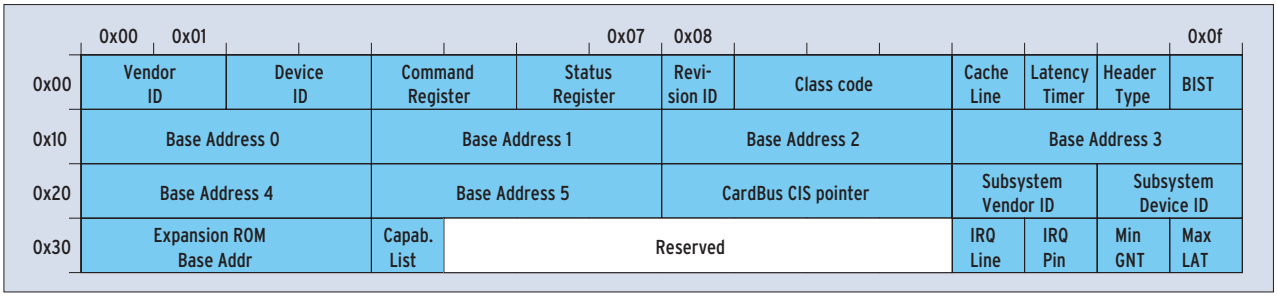


Abbildung 4: Jedes PCI-Gerät besitzt einen 256 Byte großen Konfigurationsspeicher, in dem die wichtigsten Hardwareparameter – teils kodiert – abgelegt sind. Das PCI-Subsystem des Kernels stellt die wesentlichen Parameter dem Treiber über eine Datenstruktur vom Typ »struct pci_dev« bereit.

renden Ressourcen gelangt: Jedes Mal, wenn das PCI-Subsystem die Probefunktion aufruft, übergibt der Kernel dem Treiber eine Datenstruktur vom Typ »struct dev_id«, die alle gewünschten Informationen enthält.

Sechs mögliche Konfigurationen pro PCI-Gerät

Jedes PCI-Gerät kann bis zu sechs unterschiedliche Speicher- oder I/O-Bereiche besitzen. Die Adressen, die Art und die Größe dieser Bereiche speichern Zellen, die im Konfigurationsspeicher der PCI-Hardware (siehe Abbildung 4) mit den Namen »Base Address 0« bis »Base Address 5 « gekennzeichnet sind. Da die Informationen über Adresslage, Art und Größe in einem 32-Bit-Wert kodiert sind, muss der Treiber die Werte zuerst geeignet dekodieren.

Dabei helfen dem Treiberentwickler die im Headerfile »linux/pci.h« definierten Makros. Jedes davon erwartet zwei Parameter: den Zeiger auf die übergebenen PCI-Daten zum Gerät (»struct pci_dev«) sowie die Nummer der Zelle im Konfigurationsspeicher (also von 0 bis 5). Das Makro »pci_resource_start« liefert die Anfangsadresse des Speicherbereichs zurück. Jeder Bereich hat eine Mindestlänge von 16 Byte. Die genaue Länge bringt das Makro »pci_resource_len« in Erfahrung.

Ein Makro für jeden Zweck

Darüber hinaus kann sich der Treiberentwickler das Ende des Speicherbereichs mit dem Makro »pci_resource _end« angeben lassen. Auch zur Typbestimmung der Ressource – ob Speicher- oder I/O-Adresse – gibt es ein Makro: »pci_resource_flag«. Die Header-

Listing 2: Codefragment für eigene Treiber

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/pci.h>
06 #include <linux/interrupt.h>
07
08 MODULE_LICENSE("GPL");
09
10 // TODO: Hier müssen die eigenen IDs eingetragen werden.
11 #define MY_VENDOR_ID 0x10b7
12 #define MY_DEVICE_ID 0x5157
13
14 static unsigned long ioport=0L, iolen=0L, memstart=0L, memlen=0L;
15
16 static irqreturn_t pci_isr( int irq, void *dev_id, struct pt_regs *regs )
17 {
18         return IRQ_HANDLED;
19 }
20
21 static int DeviceInit(struct pci_dev *dev, const struct pci_device_id *id)
22 {
23         printk("0x%4.4x|0x%4.4x: \"%s\"\n", dev->vendor, dev->device,
24                                 dev->dev.name );
25         if( request_irq(dev->irq,pci_isr,SA_INTERRUPT|SA_SHIRQ,"pci_drv",
   NULL) ) {
26                 printk( KERN_ERR "pci_drv: IRQ %d not free.\n", dev->irq );
27                 return -EIO;
28         }
29         ioport = pci_resource_start( dev, 0 );
30         iolen = pci_resource_len( dev, 0 );
31         if( request_region( ioport, iolen, dev->dev.name )==NULL ) {
32                 printk(KERN_ERR "I/O address conflict for device \"%s\"\n",
33                                 dev->dev.name);
34                 goto cleanup_irq;
35         }
36         memstart = pci_resource_start( dev, 1 );
37         memlen = pci_resource_len( dev, 1 );
38         if( request_mem_region( memstart, memlen, dev->dev.name )==NULL ) {
39                 release_region( ioport, iolen );
40                 printk(KERN_ERR "Memory address conflict for device \"%s\"\n",
41                                 dev->dev.name);
42                 goto cleanup_ports;
43         }
44         pci_enable_device( dev );
45         return 0;
46 cleanup_ports:
47         release_region( ioport, iolen );
48 cleanup_irq:
49         free_irq( dev->irq, "pci_drv" );
50         return -EIO;
51 }
52
53 static void DeviceDeInit( struct pci_dev *pdev )
54 {
55         free_irq( pdev->irq, "pci_drv" );
56         if( ioport )
57                 release_region( ioport, iolen );
58         if( memstart )
59                 release_mem_region( memstart, memlen );
60 }
61
62 static struct file_operations PCIFops;
63
64 static struct pci_device_id pci_drv_tbl[] __devinitdata = {
65         { MY_VENDOR_ID, MY_DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
66         { 0, }
67 };
68
69 static struct pci_driver pci_drv = {
70         .name= "pci_drv",
71         .id_table= pci_drv_tbl,
72         .probe= DeviceInit,
73         .remove= DeviceDeInit,
74 };
75
76 static int __init pci_drv_init(void)
77 {
78         if(register_chrdev(240, "PCI-Driver", &PCIFops)==0) {
79                 if( pci_module_init(&pci_drv) == 0 )
80                         return 0;
81                 unregister_chrdev(240,"PCI-Driver");
82         }
83         return -EIO;
84 }
85
86 static void __exit pci_drv_exit(void)
87 {
88         pci_unregister_driver( &pci_drv );
89         unregister_chrdev(240,"PCI-Driver");
90 }
91
92 module_init(pci_drv_init);
93 module_exit(pci_drv_exit);
```

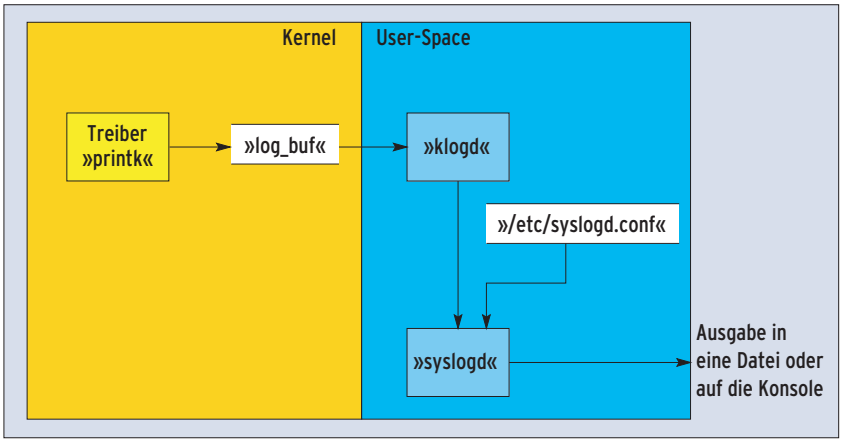


Abbildung 5: Mit »printk« hat Code aus dem Kernel die Möglichkeit, Ausgaben an den User durchzuführen. Der Kernel-Log-Daemon übergibt die Ausgaben dem normalen Syslogd-Mechanismus von Linux.

datei »linux/ioport.h« definiert die zugehörigen Flags »IORESOURCE_IO«, »IORESOURCE_MEM«, »IORESOURCE_IRQ« und »IORESOURCE_DMA«.

Auch bei PCI ist Kenntnis der Hardware unabdingbar

Auch wenn sich mit diesen Mechanismen die Ressourcen eines PCI-Geräts automatisch reservieren ließen, muss der Programmierer dennoch die Ressourcen kennen. Im Normalfall bringt er, zum Beispiel aus der Hardwarebeschreibung, in Erfahrung, an welcher Adresse im Konfigurationsspeicher welche Ressource hinterlegt ist.

Listing 2 zeigt das Gerüst einer einfachen PCI-Geräte-Initialisierung. Das Beispiel nimmt an, dass das Gerät einen Interrupt benötigt sowie im Konfigurationsspeicher »Base Address 0« eine I/O-Adresse und im Konfigurationsspeicher »Base Address 1« Adressen im Speicheradressraum spezifiziert hat. Wer seinen eigenen PCI-Treiber schreiben will, muss auf jeden Fall daran denken, die Identifikation der Ressourcen »MY_VENDOR_ID« und »MY_DEVICE_ID« der eigenen Hardware anzupassen.

Das Modul lässt sich aber auch ohne spezielle Hardware übersetzen und laden. Wer vorher das Speaker-Modul getestet hatte, darf nicht vergessen mit dem Kommando »rmmod« vorher das alte Modul zu entladen, da beide Module die gleiche Majornumber kodiert haben. Während beim Laden des Moduls der komplette Name des Objekts anzugeben ist, reicht beim Entladen der eigentliche Namensteil: »insmod speaker.ko«, aber »rmmod speaker«.

An Listing 2 fallen noch zwei Besonderheiten auf: die Verwendung von »goto« und die Direktive »__devinit«. Das Goto wäre leicht vermeidbar gewesen. Da es aber dem Kernel-Programmierstil entspricht, ist es bewusst eingesetzt (siehe Kasten „Goto OR Not Goto“).

Die Direktive »__devinitdata« reiht sich in die in der ersten Folge [3] beschriebenen Direktiven »__init« und »__exit« ein. Benutzt wird sie bei der Deklaration und Definition globaler Variablen, die wiederum die Initialisierung von Hotpluggable Devices benötigt. Falls der Kernel diese Geräteklasse nicht unterstützt, legt er die so gekennzeichneten Variablen in das Init-Segment. Dieser Speicher wird nach dem Booten wieder freigegeben.

Goto OR Not Goto

Die Gelehrten sind sich einig: Goto ist kein Element des strukturierten Programmierens und daher zu vermeiden. Jedoch widersetzt sich hier Linus Torvalds einer Professorenmeinung (und es ist nicht das erste Mal). Ein sehr lesenswerter Thread in der Kernel-Mailing-Liste [4] diskutiert das Für und Wider. Demnach ist Goto innerhalb des Betriebssystemkerns gern gesehen, bringt es doch zwei wesentliche Vorteile mit sich: Der Code wird erstens übersichtlicher und damit lesbarer und zweitens machen Gotos den Code effizienter. Goto überall verwenden wäre allerdings der falsche Weg. Ihn lehnen auch Torvalds und seine Crew ab.

Der Linux-Kernel benutzt Gotos hauptsächlich beim Aufräumen fehlgeschlagener Initialisierungen. Hier hilft das Schlüsselwort dabei, die Ressourcen umgekehrt zu der Reihenfolge freizugeben, in der sie der Kernel einst reserviert hatte (siehe Listing 2, Funktion »DeviceInit«).

Zum Ausprobieren

Zum Schluss noch etwas zum Ausprobieren: Werden in Listing 2 die Zeilen 25 bis 49 gelöscht und die beiden Defines »MY_DEVICE_ID« und »MY_VENDOR_ID« auf »PCI_ANY_ID« gesetzt (»#define MY_DEVICE_ID PCI_ANY_ID«), dann ruft der Kernel nach dem Laden des neuen Moduls die Probefunktion »DeviceInit« für all jene PCI-Geräte auf, für die er bisher keine passenden Treiber finden konnte.

Die »printk«-Funktion in Zeile 23 erzeugt eine entsprechende Ausgabe in den Syslogs (Abbildung 5). Ob sie auf der Konsole erscheint oder in einer Datei (zum Beispiel »/var/log/messages«), hängt von der Konfiguration des Syslog-Daemons (oft »/etc/syslogd«) ab.

Vorschau

Wer schon immer wissen wollte, wie ein Kernel-Thread programmiert wird und was ein Tasklet ist, sollte die kommende Folge „Kern-Technik“ im Linux-Magazin nicht verpassen. Da Linus Torvalds im Umfeld von Soft-IRQs und Work-Queues beim Kernel 2.6 mächtig aufgeräumt hat, wird das Thema auch für alte Kernel-Hasen spannend. *(jk)* ■

Infos:

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 09/03, S. 86

[2] Assembler Page (TU-Chemnitz): [http://www.tu-chemnitz.de/informatik/RA/educ/mop/assembler_hints/ports.html]

[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“ – Folge 1: Linux-Magazin 08/03, S. 88

[4] Linux-Kernel-Mailingliste, „Verwendung von Gotos im Kernel“: [http://www.lkml.org/archive/2003/1/12/128/index.html]

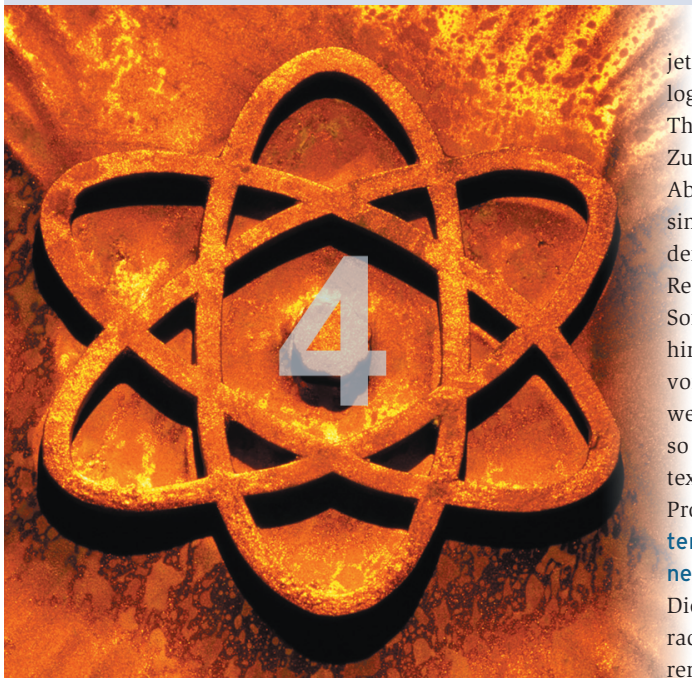
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 4

Kern-Technik

Die Treiber aus den ersten Kern-Technik-Folgen reagieren nur auf Anfragen einer Userspace-Applikation. Andere Vorgänge laufen im Kernel asynchron ab. Sie reagieren auf Hardware- und Software-Interrupts, sind Timer-gesteuert oder arbeiten als eigenständiger Kernel-Thread. Eva-Katharina Kunst, Jürgen Quade



Es spart Zeit, Vorgänge automatisiert im Kernel ablaufen zu lassen. Das bewies beispielsweise der im 2.4er Kernel integrierte Webserver »khttpd«, auch Tux genannt **[1]**. Er ist im Kernel 2.6 allerdings wieder verschwunden. Die Technik, einzelne Funktionen oder ganze Applikationen im Kernel ablaufen zu lassen, wurde aber erweitert und verfeinert. Wer ein bestimmtes Zeitverhalten implementieren oder zyklisch auf Daten zugreifen muss, Statistiken erstellen oder Informationen puffern will, der verwendet asynchrone Funktionen.

Aufgeräumt

Dem im Kernel 2.4 vorherrschenden Wildwuchs hat Linus Torvalds in Version 2.6 Einhalt geboten. Neben den in der dritten Kern-Technik-Folge **[4]** vorgestellten Interrupts (Hard-IRQs) gibt es jetzt die beiden Basistechnologien Soft-IRQ und Kernel-Thread.

Zur Unterscheidung: Beim Abarbeiten eines Hard-IRQ sind weitere Interrupts auf dem gleichen Prozessor im Regelfall nicht zugelassen. Soft-IRQs und Kernel-Threads hingegen können sehr wohl von Interrupts unterbrochen werden. Soft-IRQs laufen im so genannten Interrupt-Kontext ab, Kernel-Threads im Prozess-Kontext (siehe Kasten „Begriffe aus der Kernelwelt“).

Diese Unterschiede sind gerade für Kernel-Programmierer wesentlich, da ihnen einige Kernelfunktionen im Interrupt-Kontext nicht zur Verfügung stehen. Die Basistechnologien Soft-IRQ und Kernel-Thread existieren in unterschiedlichen Ausprägungen. So sind Soft-IRQs feiner differenziert in Soft-IRQ, Tasklet und Timer. Bei Kernel-Threads unterscheidet man Kernel-Thread, Workqueue und Event-Workqueue.

Kernel-Programmierer sollten die Ausprägung Soft-IRQ nach Möglichkeit meiden und dafür auf Tasklets und Timer zurückgreifen. Ein Tasklet übernimmt länger dauernde Aufgaben von einer ISR (Interrupt Service Routine). Ein Timer ist dann nützlich, wenn zu einem bestimmten Zeitpunkt oder periodisch Jobs auszuführen sind.

Wenn es um komplexe Aufgaben geht, greift der Entwickler gern zu Kernel-Threads. In der Ausprägung Workqueue kann er Funktionen durch den Betriebssystemkern aufrufen lassen. Noch weniger Aufwand fällt bei der vordefinierten Event-Workqueue an; allerdings ist bei ihr am wenigsten genau bestimmbar, wann der Auftrag an der Reihe ist.

Weiche Interrupts und Tasklets

Soft-IRQs sind als Erste an der Reihe, wenn alle Hardware-ISRs abgearbeitet sind. Von den 32 möglichen sind in den Kernelquellen (»linux/interrupt.h«) die sechs bereits vordefinierten Soft-IRQs aufgelistet. Für den Treiberentwickler sind hiervon die beiden Ausprägungen Tasklet sowie Timer interessant.

Tasklets übernehmen komplexe Aufgaben von Hardware-ISRs. Lange Hardware-ISRs führen zu ungünstigen Inter-

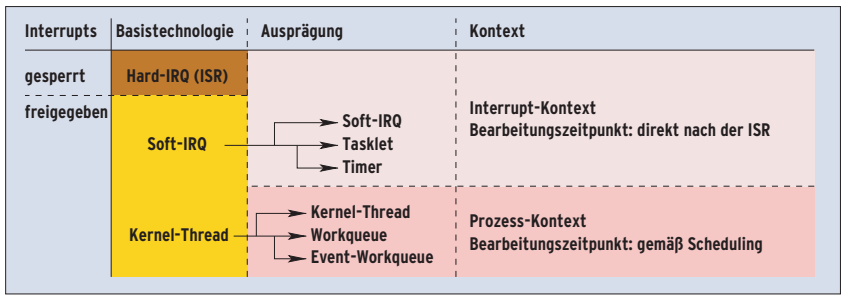


Abbildung 1: Asynchrone Verarbeitung im Kernel ist mit mehreren Technologien möglich, die in unterschiedlichen Kontexten ablaufen. Bei Soft-IRQs und Kernel-Threads gibt es je drei Ausprägungen.

rupt-Latenzzeiten und sind daher zu vermeiden. Sinnvoller ist es, solche Routinen in zwei Teile zu trennen. Der erste führt die (zeit-)kritischen Aktionen aus, während die Interrupts gesperrt sind. Die übrigen Berechnungen finden im zweiten Teil statt, bei freigegebenen Interrupts. Für diesen früher Bottom Half genannten Teil (siehe Kasten „Portierungshilfe“) eignen sich Tasklets.

Der Kernel sorgt dafür, dass ein Tasklet zu einem Zeitpunkt maximal einmal abläuft. Das gilt auch für Mehrprozessorsysteme. Unterschiedliche Tasklets zur gleichen Zeit einzusetzen ist allerdings möglich. Da Tasklets zur Gruppe der Soft-IRQs gehören, genießen sie im System die zweithöchste Priorität, direkt nach den Hardware-Interrupts. Allerdings kommen Tasklets in zwei Prioritätsstufen vor.

Immer der Reihe nach: Priorität der IRQs

Die hoch priorisierten Tasklets sind als erste Soft-IRQs an der Reihe, unmittelbar nach den Hardware-Interrupts. Niedrig priore Tasklets zuletzt. Entwickler sind aufgefordert, wenn möglich die niedrig priore Variante zu verwenden.

Um ein Tasklet zu spezifizieren, muss die ISR zwei Informationen angeben: die Adresse der Funktion, die der Kernel aufrufen soll, sowie einen Parameter, den der Kernel beim Aufruf an die Funktion weitergibt. Beide Angaben liegen in »struct tasklet_struct«. Das Makro »DECLARE_TASKLET« hilft dabei, diese Datenstruktur mit Inhalt zu füllen.

Dass er das Tasklet ausführen soll, erfährt der Kernel durch »tasklet_schedule()«. Diese Funktion muss zum richtigen Zeitpunkt aufgerufen werden, typischerweise innerhalb einer Hardware-ISR. Um ein übersichtliches und abgeschlossenes Beispiel zur Hand zu haben, ruft der Code in Listing 1 die Tasklet-Schedule-Funktion schon während der Modulinitialisierung in Zeile 18 auf.

Achtung: Race Condition

Bei Tasklets kann sich der Entwickler leicht in eine Race Condition verstricken: Wird das Modul entladen (User-Kommando »rmmod«) und ruft der Kernel danach ein von diesem Modul schedultes Tasklet auf, dann soll der Prozessor Code abarbeiten, der nicht mehr vorhanden ist. Das aber ist unmöglich. Der Kernel reagiert mit einer Oops-Meldung.

Der Treiberentwickler muss dafür sorgen, dass sein Modul vor dem Entladen alle Tasklets entfernt, die der Kernel noch nicht abgearbeitet hat. Die entsprechende Funktion heißt »tasklet_kill()«. Der Treiber kann sie gefahrlos aufrufen, selbst wenn er das Tasklet noch gar nicht schedult hat.

Nach dem Kompilieren und Laden (siehe erste Kern-Technik-Folge [2]) des Moduls in Listing 1 sollte im Syslog [4] die Meldung »Tasklet called...« auftauchen. Die Syslogs lassen sich übrigens sehr gut über »tail -f *Logfile*« überwachen. Also: weiteres Fenster mit einer Shell öffnen und »tail -f /var/log/messages« oder Ähnliches aufrufen.

Die in der Listing-Zeile 13 angelegte Struktur »TlDescr« erklärt die Funktion »TasketFunction()« zum Einstiegspunkt des Tasklets. In Zeile 18 sorgt »tasklet_schedule()« dafür, dass der Kernel das Tasklet als niedrig prioren Soft-IRQ abarbeitet. Wer einen hoch prioren Soft-IRQ verwenden möchte, setzt die Funktion »tasklet_hi_schedule()« ein.

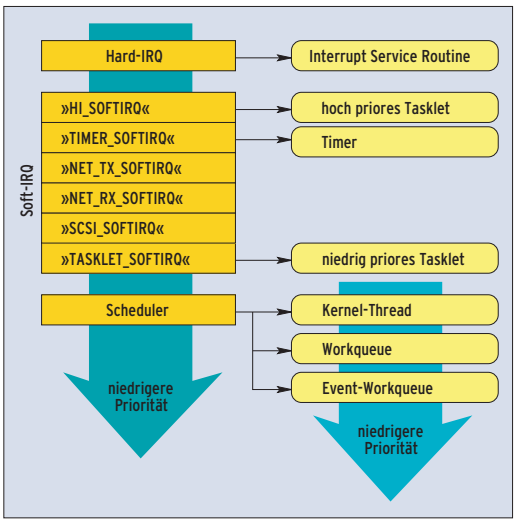


Abbildung 2: Hard- und Software-Interrupts sind hoch prior. Erst wenn alle Interrupts abgearbeitet sind, kommen die Rechenprozesse, zu denen auch die Kernel-Threads zählen, an die Reihe.

Begriffe aus der Kernelwelt

Kontext: Der Kontext (die Umgebung) gibt an, auf welche Dienste und welche Ressourcen ein Codefragment Zugriff hat.

User-Kontext: Der Programmcode einer Applikation läuft im so genannten User-Kontext ab. Ihm stehen die Dienste des Betriebssystems zur Verfügung, wie sie im Systemcall-Interface spezifiziert sind.

Kernel-Kontext: Jeglicher Code des Betriebssystemkerns wird im Kernel-Kontext abgearbeitet. Man unterscheidet Prozess-Kontext und Interrupt-Kontext.

Prozess-Kontext: Die reguläre Umgebung, in der der Kernelcode abläuft, heißt Prozess-Kontext. Hier stehen sämtliche Funktionalitäten des Betriebssystemkerns zur Verfügung, auch das Verarbeiten des Codes für einige Zeit anzuhalten (schlafen zu legen) oder Daten zwischen den Speicherbereichen der Applikationen und des Kernels zu übertragen. Applikations-getriggerte Funktionen eines Gerätetreibers (wie »DriverRead()«), Systemcalls, Kernel-Threads und damit auch Workqueues werden im Prozess-Kontext abgearbeitet.

Interrupt-Kontext: Im Interrupt-Kontext stehen dem Kernelcode nicht alle Dienste des Betriebssystemkerns zur Verfügung. Funktionen in diesem Kontext können sich nicht schlafen legen. Sie dürfen auch keine Funktionen wie »kmalloc()« aufrufen, da diese die Routine unter Umständen schlafen legen wollen. Der Zugriff auf Speicherbereiche der Applikationen (zum Beispiel per »copy_from_user()«) ist ebenfalls tabu. Die Verarbeitung im Interrupt-Kontext ist möglichst kurz zu halten, um reguläre Rechenprozesse nicht unnötig zu verzögern. Routinen, die im Interrupt-Kontext ablaufen, sind ISRs (Interrupt Service Routine), Soft-IRQs, Tasklets und Timer.

Kernelspace: Als Kernelspace bezeichnet man Speicherbereiche, auf die eine Routine innerhalb des Kernels direkt (über Zeiger) zugreifen kann, ohne besondere Funktionen zu bemühen. Nur der Kernel hat Zugriff auf den Kernelspace. Zugang zum Userspace geben ihm die Funktionen »copy_to_user()«, »copy_from_user()«, »put_user()« und »get_user()«, allerdings nur auf den Speicher des gerade aktiven Rechenprozesses.

Userspace: Die Speicherbereiche, auf die eine Applikation direkt, zum Beispiel über Zeiger, zugreifen kann, heißen Userspace. Zum Kernelspace hat eine Applikation keinen Zugang, auch nicht mit Superuser-Rechten.

Ebenso wichtig wie Tasklets sind Timer, die zweite Variante der Soft-IRQs. Mit ihrer Hilfe beauftragt der Entwickler den Kernel damit, eine Funktion zu einem bestimmten Zeitpunkt aufzurufen. Der Kernel misst Zeitpunkte nicht absolut, sondern relativ zum Einschaltzeitpunkt. Als Basis dient die Anzahl der seit dem Einschalten ausgelösten periodischen Timer-Interrupts: Der Kernel zählt sie mit der Variablen »jiffies«.

Augenblicke zählen: Jiffies

Um eine Funktion zu einem bestimmten Zeitpunkt aufrufen zu lassen, muss das Modul eine Datenstruktur vom Typ »struct timer_list« definieren und mit »init_timer()« initialisieren. Danach sind die Adresse der abzuarbeitenden Funktion (Feld »function«), der Abarbeitungszeitpunkt (»expires«) und Eingabedaten für die Funktion (»data«) zu spezifizieren. Der Zeitpunkt ist absolut in »jiffies« anzugeben. Soll ein Modul eine Funktion in einem bestimmte Zeitabstand auslösen, muss es nur eine einfache Rechenaufgabe lösen: aktuellen Zeitpunkt und Relativwert addieren.

Zeitgesteuerte Abläufe

Ist der Timer in Form der Datenstruktur definiert und initialisiert, übergibt die Funktion »add_timer()« die Aufgabe an den Kernel. Sobald der im Expires-Feld angegebene Zeitpunkt erreicht oder überschritten ist, ruft der Kernel die darin spezifizierte Funktion genau ein Mal auf. Soll er eine Funktion periodisch aufrufen, muss diese selbst ihre »timer_list« mit dem nächsten Aufrufzeitpunkt initialisieren und dann dem Kernel erneut übergeben.

In Listing 2 sorgt Zeile 24 dafür, dass der Kernel die Funktion »IncCount()« zwei Sekunden nach dem Laden des Moduls aufruft. Zeile 15 wiederholt dies, sodass die Funktion periodisch alle zwei Sekunden abläuft. Beleg für den Funktionsaufruf ist eine Ausgabe im Syslog. Achtung: Wer den Kernel damit beauftragt, zu einem Zeitpunkt ein einzelnes Timer-Objekt mehrfach abzuarbeiten, löst eventuell einen Crash aus.

Erfahrene Entwickler werden sicherlich bemerken, dass auch bei den Timern ein kritischer Abschnitt vorliegt. Das Modul darf nur dann entladen werden, wenn keine Timer-Funktion mehr im System anhängig ist. Die Lösung: »del_timer_sync()« entfernt eine als Timer eingehängte Funktion wieder.

Listing 1: Tasklet-Modul

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/interrupt.h>
05
06 MODULE_LICENSE("GPL");
07
08 static void TaskletFunction( unsigned long
   data )
09 {
10     printk("Tasklet called...\n");
11 }
12
13 DECLARE_TASKLET( TlDescr, TaskletFunction, 0L
   );
14
15 static int __init ModInit(void)
16 {
17     printk("ModInit called\n");
18     tasklet_schedule( &TlDescr ); // Tasklet
   so bald als möglich ausführen
19     return 0;
20 }
21
22 static void __exit ModExit(void)
23 {
24     printk("ModExit called\n");
25     tasklet_kill( &TlDescr );
26 }
27
28 module_init( ModInit );
29 module_exit( ModExit );
```

Listing 2: Timer-Funktionen

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/timer.h>
04 #include <linux/sched.h>
05 #include <linux/init.h>
06
07 MODULE_LICENSE("GPL");
08
09 static struct timer_list MyTimer;
10
11 static void IncCount(unsigned long arg)
12 {
13     printk("IncCount called (%ld)...\n",
14         MyTimer.expires);
15     MyTimer.expires = jiffies + (2*HZ); // zwei
   Sekunden
16     add_timer( &MyTimer );
17 }
18
19 static int __init ktimerInit(void)
20 {
21     init_timer( &MyTimer );
22     MyTimer.function = IncCount;
23     MyTimer.data = 0;
24     MyTimer.expires = jiffies + (2*HZ); // zwei
   Sekunden
25     add_timer( &MyTimer );
26     return 0;
27 }
28
29 static void __exit ktimerExit(void)
30 {
31     if( del_timer_sync(&MyTimer) )
32         printk("Activ timer deactivated.\n");
33     else
34         printk("No timer activ.\n");
35 }
36
37 module_init( ktimerInit );
38 module_exit( ktimerExit );
```

Prozess im System: Der Kernel-Thread

Anders als Tasklets und Timer können und müssen sich Kernel-Threads während ihrer Laufzeit schlafen legen. Ein Kernel-Thread entspricht einem Thread auf Userebene, aber mit dem Unterschied, dass er komplett im Kernelspace abläuft. Er ist wie jeder andere Rechenprozess durch eine Taskstruktur repräsentiert und erscheint beim Kommando »ps auxw« auch in der Taskliste (siehe Abbildung 4). Der Abarbeitungszeitpunkt eines Kernel-Thread wird vom Scheduler festgelegt.

Diese Threads sind einfach zu erzeugen, ein Aufruf von »kernel_thread()« genügt. Die Funktion benötigt drei Parameter:

- Die Adresse der Funktion, die der Scheduler beim ersten Aktivieren aufrufen soll.
- Ein Argument für die aufzurufende Funktion.
- Ein Bitfeld, das steuert, wie der Kernel den neuen Rechenprozesses erzeugt.

Im Regelfall genügt im Bitfeld der Eintrag »CLONE_KERNEL«, damit läuft das Erzeugen des Thread sehr effizient ab. Der Kernel muss keine Filesystem-Informationen (»CLONE_FS«), keine Filedeskriptoren (»CLONE_FILES«) und Signal-

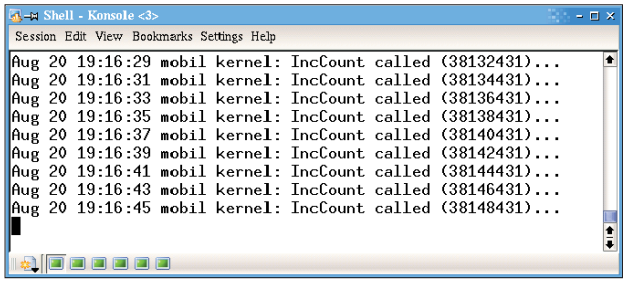


Abbildung 3: Das Timer-Modul schreibt alle zwei Sekunden eine Ausgabe in den Syslog. Am Wert in Klammern ist erkennbar, dass im Kernel jeweils 2000 Timer-Ticks verstrichen sind.

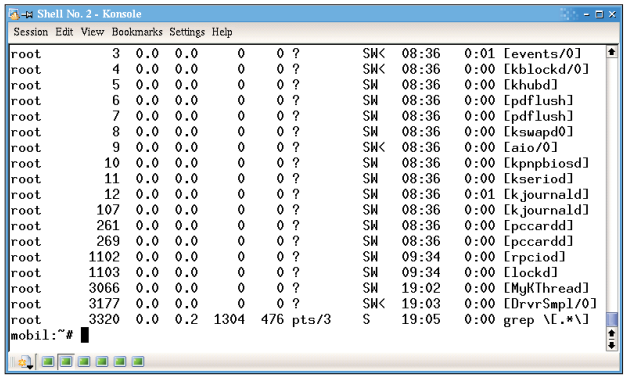


Abbildung 4: Das Kommando »ps auxw« zeigt neben den Userprozessen auch die Kernel-Threads. Letztere sind in der Taskliste durch eckige Klammern markiert.

Handler (»CLONE_SIGHAND«) kopieren oder neu anlegen.

Der Rückgabewert der Funktion ist die Prozess-Identifikation (PID) des neuen Rechenprozesses. Sie ist später nötig, um den Kernel-Thread wieder zu entfernen. Der Rückgabewert »0« bedeutet, dass das Erzeugen schief gegangen ist:

```
ThreadID = kernel_thread( ThreadCode, NULL,
    CLONE_KERNEL );
```

Sobald er erzeugt ist, muss der neue Thread einige Initialisierungen durchführen. Zunächst gibt er per »daemonize()« alle Ressourcen im Userbereich frei, die ihm der Kernel standardmäßig mitgegeben hat.

Zum Daemon mutieren

Die Daemonize-Funktion erwartet als ersten Parameter den Namen des neuen Thread. Sie interpretiert ihn – ähnlich wie »printf()« – als Format-String, die zugehörigen Parameter sind in der Argumentliste ebenfalls enthalten. Sie erzeugt aus dem Format-String den Namen des Kernel-Thread. Er darf maximal 16 Zeichen lang sein, den Rest schneidet »daemonize()« ab.

Ein Kernel-Thread muss sich entweder rechtzeitig beenden oder selbst schlafen legen. Andernfalls würde er den anderen Rechenprozessen, insbesondere den Benutzerapplikationen, alle Rechenzeit wegnehmen. Wie in der zweiten Kern-Technik-Folge [3] gezeigt, hilft eine Waitqueue weiter.

Schlafender Thread

In Listing 3, Zeile 22 legt sich der Kernel-Thread mit Hilfe der Waitqueue eine Sekunde lang schlafen: »wait_event_interruptible_timeout()«. In der Praxis gibt es bessere Kriterien für das Schlafen und Aufwecken des Kernel-Thread: zum Beispiel auf Daten warten, die über eine TCP/IP-Verbindung kommen.

Problematisch ist aber das Beenden des Thread. Alle Rechenprozesse, sowohl die im Userbereich als auch die Kernel-Threads, kommen in der Regel selbstständig zu ihrem Ende. Will ein Prozess einen anderen beenden, sendet er ihm ein Signal. Allerdings hat die Funktion »daemonize()« alle Signale blockiert. Der Kernel-Thread muss sie daher erst wieder freigeben, er ruft dazu die Funktion »allow_signal()« auf.

Schwieriges Ende

Ob während des Schlafens ein Signal angekommen ist, zeigt der Rückgabewert des Makros »wait_event_interruptible_timeout()«. Zeile 25 prüft, ob der Wert »-ERESTARTSYS« entspricht, er bedeutet in diesem Zusammenhang „Thread beenden“. An beliebiger Stelle im Code stellt die Funktion »signal_pending()« fest, ob ein Signal angekommen ist oder

Listing 3: Kernel-Thread

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/completion.h>
05
06 MODULE_LICENSE("GPL");
07
08 static int ThreadID=0;
09 static wait_queue_head_t wq;
10 static DECLARE_COMPLETION( OnExit );
11
12 static int ThreadCode( void *data )
13 {
14     unsigned long timeout;
15     int i;
16
17     daemonize("MyKThread");
18     allow_signal( SIGTERM ); // Funktion ab
       2.5.61 nutzbar
19     printk("ThreadCode startet ...\n");
20     for( i=0; i<10; i++ ) {
21         timeout=HZ; // Eine Sekunde
22         timeout=wait_event_interruptible_
             timeout(
23             wq, (timeout==0), timeout);
24         printk("ThreadCode: woke up ...\n");
25         if( timeout==-ERESTARTSYS ) {
26             printk("got signal, break\n");
27             ThreadID = 0;
28             break;
29         }
30     }
31     complete_and_exit( &OnExit, 0 );
32 }
33
34 static int __init kthreadInit(void)
35 {
36     init_waitqueue_head(&wq);
37     ThreadID=kernel_thread(ThreadCode, NULL,
     CLONE_KERNEL );
38     if( ThreadID==0 )
39         return -EIO;
40     return 0;
41 }
42
43 static void __exit kthreadExit(void)
44 {
45     kill_proc( ThreadID, SIGTERM, 1 );
46     wait_for_completion( &OnExit );
47 }
48
49 module_init( kthreadInit );
50 module_exit( kthreadExit );
```

nicht. Die globale Variable »current« zeigt auf den aktiven Thread:

```
if( signal_pending(current) ) {
    // Signal eingetroffen
} else {
    // Kein Signal
}
```

Der Kernel-Thread beendet sich, indem er die Thread-Funktion per »return« verlässt. Das Signal kann man zum Beispiel in der Shell mit »kill« absenden oder innerhalb des Kernels mit der Funktion »kill_proc()«. In beiden Fällen wird der Thread über die PID ausgewählt.

Completion-Objekt

Erneut droht die Gefahr eines ungeschützten kritischen Abschnitts. Wie bei den übrigen Methoden muss der Entwickler auch hier sicherstellen, dass sich der Kernel-Thread beendet hat, bevor das Modul entladen wird. Die Aufforderung, sich zu beenden, ist über »kill_proc()« einfach realisiert.

Da sich der Rechenprozess selbst beendet, muss das Modul jetzt noch auf das wirkliche Ende warten. Der Schutz dieses kritischen Abschnitts wird dadurch kompliziert, besonders wenn man bedenkt, dass bei einem Mehrprozessorsystem der Code zum Entladen des Moduls eventuell auf einem anderen Prozessor läuft als der Kernel-Thread.

Aus diesem Grund hat die Torvaldsche Entwicklergemeinde das Completion-Objekt entwickelt. Das Kernelmodul muss es global per »DECLARE_COMPLETION«-Makro definieren (Zeile 10 in Listing 3). Sobald sich der Thread beendet, ruft er die Funktion »complete_and_exit()« auf (Zeile 31). Deren Parameter sind die Adresse des Completion-Objekts und der Exitcode des Thread. Die Modul-Entladefunktion wählt »wait_for_completion()« (Zeile 46), die als Parameter ebenfalls die Adresse des Completion-Objekts erhält. Sie kehrt erst zurück, wenn ein passendes »complete_and_exit()« aufgerufen wurde.

Selbst wenn ein Modul mehrere Kernel-Threads startet, genügt ein einzelnes Completion-Objekt. Jeder Aufruf von »wait_for_completion()« im Modul muss sich mit einem Aufruf von »complete_and_exit()« in einem Thread decken. Wartet das Modul auf zwei Kernel-Threads, ruft es in der Entladefunktion zweimal »wait_for_completion()« auf, jeder der beiden Threads muss »complete_and_exit()« einmal aufrufen.

Listing 4: Workqueue

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/workqueue.h>
05
06 MODULE_LICENSE("GPL");
07
08 static struct workqueue_struct *wq;
09
10 static void WorkQueueFunction( void *data )
11 {
12     printk("WorkQueueFunction\n");
13 }
14
15 static DECLARE_WORK( work, WorkQueueFunction, NULL );
16
17 static int __init ModInit(void)
18 {
19     wq = create_workqueue( "DrvrSmpl" );
20     if( queue_work( wq, &work ) )
21         printk("work has been queued ...\n");
22     else
23         printk("queue_work failed ...\n");
24     return 0;
25 }
26
27 static void __exit ModExit(void)
28 {
29     pr_debug("ModExit called\n");
30     if( wq ) {
31         destroy_workqueue( wq );
32         printk("workqueue destroyed\n");
33     }
34 }
35
36 module_init( ModInit );
37 module_exit( ModExit );
```

Workqueue statt Schlafstörung

Einen Thread schlafen legen ist meist mit einigem Aufwand verbunden. Wer mit niedriger Priorität Funktionen im Prozess-Kontext aufrufen will, ist mit der Workqueue oft besser bedient. Dieser Kernel-Thread nimmt die Adressen von Funktionen entgegen, die er dann abarbeitet. Zwei Objekte sind nötig: »struct work_queue_struct« beschreibt die Workqueue und »struct work_struct« referenziert die Funktion.

Der Speicher für das Workqueue-Objekt wird von der Kernelfunktion »create_workqueue()« angelegt (siehe Listing 4, Zeile 19). Ihr einziger Parameter ist der Name der Workqueue oder des Kernel-Thread. Achtung: Der Name darf maximal zehn Zeichen umfassen.

Den Speicher für das Work-Objekt stellt das Modul selbst zur Verfügung. Zum Definieren und Initialisieren nutzt es das Makro »DECLARE_WORK« (Zeile 15). Dessen drei Parameter sind der Name des Work-Objekts, die Adresse der aufzurufenden Funktion und der Wert des Parameters, den die Workqueue an die Funktion übergibt.

Die Funktion »queue_work()« (Zeile 20) trägt das Work-Objekt in die Workqueue ein. Ihre beiden Parameter kennzeichnen die Adresse der Workqueue und die Adresse des Work-Objekts. Bei nächster Gelegenheit ruft die Workqueue die im Work-Objekt spezifizierte Funktion auf. Es ist sogar möglich, beliebig viele Funktionen (Work-Objekte) in einer Workqueue unterzubringen. Sie ruft jedes Objekt genau einmal auf und entfernt es anschließend wieder.

Ein Work-Objekt darf sich nicht selbst in die Workqueue einhängen, denn damit würde es sofort wieder aufgerufen. Die Folge: Das System ist dann nur noch mit dem Abarbeiten dieser einen Funktion beschäftigt. Die im Work-Objekt angegebene Funktion sollte sich auch nicht schlafen legen: Der gesamte Workqueue-Thread würde einschlummern und auf diese Weise alle übrigen eingehängten Funktionen verzögern.

Verzögerungstaktik: Arbeit später erledigen

Muss eine Funktion nicht sofort, sondern erst zeitverzögert laufen, sind Workqueues ebenfalls ein geeignetes Mittel. Hierfür ist die Funktion »queue_delayed_work()« an Stelle von »queue_work()« zuständig. Sie nimmt als zusätzlichen Parameter einen Timeout-Wert (in Jiffies gemessen) auf.

Auch Workqueues sind vor dem Entladen des Treibers wieder zu entfernen: »destroy_workqueue()«. Diese Funktion wartet so lange, bis alle Work-Objekte der Workqueue abgearbeitet sind. Daher muss der Entwickler dafür sorgen, dass keine Instanz seines Treibers das Work-Objekt ständig wieder einhängt.

Wer den Code in Listing 4 kompiliert und das generierte Modul lädt, kann in der Prozesstabelle den Kernel-Thread der Workqueue entdecken. Die Workqueue heißt »DrvrSmpl« (Zeile 19), Abbildung 4 listet sie als »[DrvrSmpl/0]«. Das vom System angefügte »/0« bedeutet, dass es die Workqueue auf der ersten CPU abarbeitet. Achtung: Workqueues lassen sich nur von Kernelerweiterungen nutzen, die selbst der GPL oder einer BSD-Lizenz unterliegen. Für proprietäre Erweiterungen ist diese Funktionalität tabu.

Ereignisreich

Zum Abschluss sei noch die Event-Workqueue erwähnt (siehe Listing 5). Der Kernel erzeugt per Default für jeden Prozessor eine solche Workqueue. In der Prozessliste (»ps auxw«) erscheinen sie unter den Namen »[events/0]«, »[events/1]« und so fort. Die Event-Workqueue tritt an die Stelle des aus früheren Versionen bekannten »keventd«. Ihr Vorteil ist, dass das lästige Aufsetzen einer Workqueue entfällt.

Dank Event-Workqueue kann der Programmierer mit »schedule_work()« (Zeile 19) ohne Umschweife Funktionen zur Abarbeitung an den Kernel übergeben. »schedule_work()« erhält als einzigen Parameter die Adresse des Work-Objekts. »schedule_delayed_work()« startet die Funktion zeitversetzt. Die Aufräumarbeiten übernimmt die Funktion »flush_scheduled_work()« (Zeile 28). Sie stößt die Abarbeitung der Event-Workqueue an und wartet dann auf deren Ende.

Vorschau

Alle Kern-Technik-Folgen weisen auf eine besondere Herausforderung für den Entwickler hin: Race Conditions vermeiden und kritische Abschnitte absichern. Für die Stabilität von Linux ist dies entscheidend. Die nächste Folge wird daher einen Blick auf die entsprechenden Methoden im Kernel 2.6 werfen. *(fjl)*

Infos:

[1] Lothar Wieske, „Turbo-Pinguin – Tux, ein schneller Webserver als Kernelmodul“: Linux-Magazin 03/02, S. 61
[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 1: Linux-Magazin 08/03, S. 88
[3] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 09/03, S. 86
[4] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 3: Linux-Magazin 10/03, S. 81

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux auch Fans von Open Source.

Portierungshilfe

Wichtige Veränderungen zwischen Kernel 2.4 und Kernel 2.6:

Bottom Half: Bottom Halves gibt es im Kernel 2.6 nicht mehr. Stattdessen kommen Tasklets zum Einsatz, der Entwickler hat die Wahl zwischen der hoch prioren und der niedrig prioren Variante. Er kann sicher sein, dass ein Tasklet zu einem Zeitpunkt nur ein einziges Mal aktiv ist, egal wie oft er es eingehängt hat. Tasklets laufen – wie Bottom Halves – im Interrupt-Kontext.

Task-Queue: Die Zeit der Task-Queues ist vorbei. Programmierer nutzten in der Regel die drei Ausprägungen »tq_immediate«, »tq_timer« und »tq_scheduler«. Die Task-Queue »tq_immediate« lässt sich durch ein Tasklet ersetzen. Statt »tq_timer« kommt ein Timer-Objekt zum Einsatz; als Zeitpunkt ist jetzt auf einer x86-Plattform der zehnfache Jiffies-Wert einzutragen. Als Ersatz für »tq_scheduler« eignet sich die Event-Workqueue.

Keventd: Der Kernel-Event-Daemon »keventd« existiert in Kernel 2.6 nicht mehr. Für ihn ist die Event-Workqueue zu verwenden.

Jiffy: Die globale Variable »jiffies« war in früheren Kernelversionen ein 32-Bit-Wert. Die Folge: Auf einer x86-Plattform kam es normalerweise nach 407 Tagen zum Zählerüberlauf. Theoretisch waren dabei inkonsistente Systemzustände möglich – für Entwickler eingebetteter Systeme ein Albtraum.
Kernel 2.6 enthält drei Änderungen. Erstens gibt es mit »jiffies_64« einen auf 64 Bit erweiterten Zähler. Zweitens wurde die Frequenz des Timer-Ticks, der den Zählerwert erhöht, von 100 Hz auf 1000 Hz erhöht. Drittens ist der Zähler so vorinitialisiert, dass er bereits nach fünf Minuten überläuft. Entwickler spüren damit die Konsequenzen wesentlich schneller und können den Code für diesen Fall debuggen.

Daemonize: Die Funktion »daemonize()« war im Kernel 2.4 ohne Parameter. Der Kernel-Thread, von dem diese Funktion aufgerufen wurde, setzte anschließend den Namen des Thread selbst. In Kernel 2.6 erhält Daemonize den Namen als Parameter und übernimmt das Setzen des Thread-Namens.

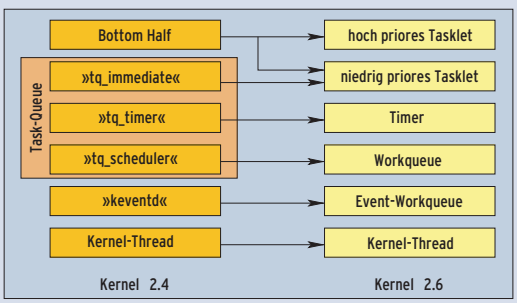


Abbildung 5: Die Techniken zur asynchronen Verarbeitung im alten und im neuen Kernel sind recht unterschiedlich. Für jede Aufgabe gibt es aber passenden Ersatz.

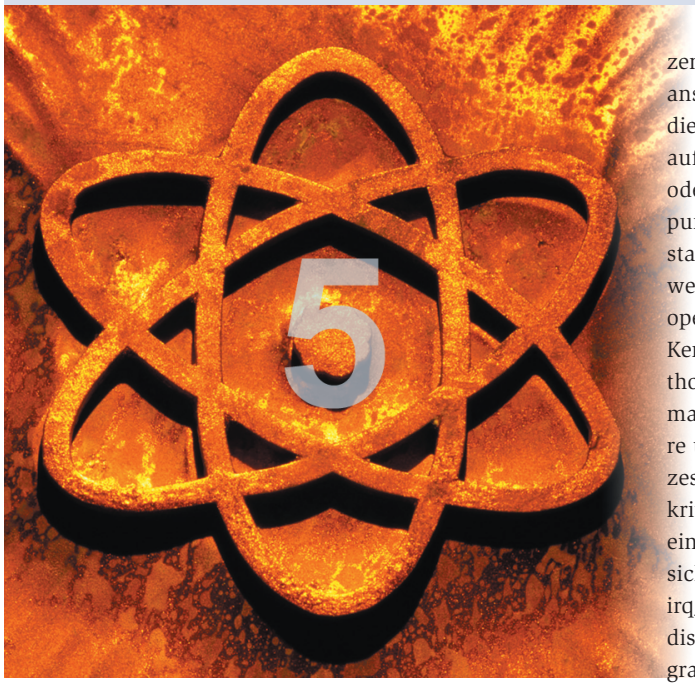
Listing 5: Die Event-Workqueue

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/workqueue.h>
05
06 MODULE_LICENSE("GPL");
07
08 static struct work_struct work;
09
10 static void WorkQueueFunction( void *data )
11 {
12     printk("WorkQueueFunction\n");
13 }
14
15 static DECLARE_WORK( work, WorkQueueFunction, NULL );
16
17 static int __init ModInit(void)
18 {
19     if( schedule_work( &work )==0 )
20         printk("schedule_work not successful ...\n");
21     else
22         printk("schedule_work successful ...\n");
23     return 0;
24 }
25
26 static void __exit ModExit(void)
27 {
28     flush_scheduled_work();
29 }
30
31 module_init( ModInit );
32 module_exit( ModExit );
```

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 5

Kern-Technik

Der Schutz kritischer Abschnitte gehört zu den wichtigsten Aspekten paralleler Programmierung. Das Locking dieser Bereiche soll zuverlässig funktionieren, aber andere Programmteile nicht unnötig bremsen. Linux 2.6 bietet fein abgestuftes Locking für Kernel-Programmierer. Eva-Katharina Kunst, Jürgen Quade



Chaos ist programmiert, wenn mehrere Prozesse parallel auf dieselben Daten zugreifen. Dazu braucht es nicht einmal ein Multiprozessorsystem. Auch auf Einprozessormaschinen enden Zugriffskonflikte oft in einem Systemabsturz. Dagegen hilft die Synchronisation paralleler Abläufe im Kernel durch Locking-Strategien. Richtig eingesetzt verbessern sie die Systemstabilität und steigern zudem die Performance.

Korrekte Locking-Mechanismen verhindern, dass mehrere Prozesse gleichzeitig einen kritischen Abschnitt betreten und gegenseitig ihre Daten durcheinander bringen. Jeder Zugriff auf ein globales Objekt kann zu einem kritischen Abschnitt führen, egal ob es sich um eine gemeinsam genutzte Schnittstelle oder eine Variable handelt.

Das Zauberwort heißt gegenseitiger Ausschluss. Konkurrieren mehrere Instanzen, kommt erst die eine und anschließend die zweite an die Reihe. Denn der Zugriff auf Daten – Parallelität hin oder her – ist zu einem Zeitpunkt x immer nur einer gestattet. Das funktioniert nur, wenn sich alle Beteiligten kooperativ verhalten.

Kernel 2.6 stellt drei Basismethoden zur Verfügung: atomare Operationen, Semaphore und Spinlocks. Auf Einprozessormaschinen lassen sich kritische Abschnitte durch eine einfache Interruptsperre sichern (Funktionen »local_irq_enable()« und »local_irq_disable()«). Viele Kernelprogrammierer kombinieren lokale Interruptsperren in ihrem Code mit den Basismethoden. Eine globale Interruptsperre, wie sie noch in Kernel 2.4 vorkam, existiert in 2.6 allerdings nicht mehr.

Geschickt kombiniert

Die Basismethoden lassen sich nicht nur manuell mit Interruptsperren kombinieren, es gibt sie auch mit mehr Funktionalität. Für die einfachen Anwendungen bietet Linux aber auch beim Locking eine einfache Lösung. Im Fall der Integervariablen lässt sich der kritische Abschnitt relativ einfach schützen. Linux stellt hierzu den Datentyp »atomic_t« zur Verfügung. Das Makro »ATOMIC_INIT()« initialisiert ein Objekt dieses Typs, die Funktionen aus Tabelle 1 lesen, schreiben beziehungsweise ändern dessen Wert.

Die drei »..._and_test()«-Varianten geben True (also den Wert »1«) zurück, falls die Variable, auf die »v« zeigt, nach der Operation den Wert »0« hat. In allen anderen Fällen ist der Rückgabewert »0«. Die Funktion »int atomic_add_negative(int i, atomic_t *v)« addiert den Wert »i« auf und gibt True (»1«) zurück, falls das Ergebnis negativ ausfällt. Ist das Ergebnis größer oder gleich null, gibt die Funktion False (»0«) zurück. Listing 1 zeigt wie die »atomic_...()«-Funktionen abgesichert eine Integervariable dekrementieren und testen.

Atomare Operationen

Der Betriebssystemkern arbeitet die Atomic-Operationen unteilbar (atomic) ab, also in einem Stück. Sie werden nicht durch Interrupts unterbrochen und auch nicht auf mehreren Prozessoren parallel ausgeführt. Die Datentypen sind in der Headerdatei »asm/atomic.h« definiert.

Tabelle 1: Schutz für Integer

Funktion	Aufgabe
int atomic_read(atomic_t *v)	Lesen
void atomic_set(atomic_t *v, int i)	Schreiben
void atomic_add(int i, atomic_t *v)	Addieren
void atomic_sub(int i, atomic_t *v)	Subtrahieren
void atomic_inc(atomic_t *v)	Inkrementieren
void atomic_dec(atomic_t *v)	Dekrementieren
int atomic_sub_and_test(int i, atomic_t *v)	Subtrahieren und Test
int atomic_inc_and_test(atomic_t *v)	Inkrementieren und Test
int atomic_dec_and_test(atomic_t *v)	Dekrementieren und Test

Wer auf andere Linux-Architekturen portierbare Software schreiben möchte, sollte von Atomic-Variablen nur ein Bitfeld von 24 Bit verwenden. Zwar stehen prinzipiell 32 Bit zur Verfügung, doch zur Implementierung des unteilbaren Zugriffs selbst verwendet der Kernel bei einigen Prozessoren 8 Bit.

Reicht gar 1 Bit, stellt Linux in der Headerdatei »asm/bitops.h« passende Inlinefunktionen zur Verfügung. Dazu ist zunächst ein Bitfeld vom Typ »u32« nötig, auf dessen 32 Bit die Funktionen »set_bit()«, »clear_bit()«, »test_and_set_bit()«, »test_and_clear_bit()« und »test_and_change_bit()« operieren. Alle Funktionen besitzen zwei Parameter: die Nummer des Bits (»int nr«, von 0 bis 31) und die Adresse der Variablen (»volatile unsigned long *addr«).

Die drei »test_...()«-Funktionen geben jeweils den Wert des ausgewählten Bits vor der Operation zurück. Das folgende Beispiel setzt mit »set_bit()« das sechste Bit (Nummer 5) und löscht per »clear_bit()« das dritte Bit (Nummer 2). In der If-Abfrage testet und setzt es Bit 0:

```
u32 sw;  // statuswort

set_bit( 5, &sw );
clear_bit( 2, &sw );
if( !test_and_set_bit(0, &sw) ) {
    ... // Bit 0 _war_ nicht gesetzt.
}
```

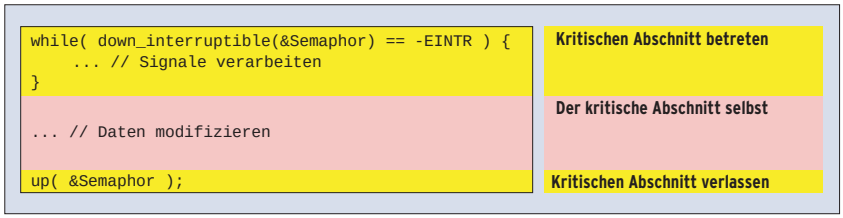


Abbildung 1: Ein Semaphor schützt den kritischen Abschnitt und stellt sicher, dass nur ein Prozess zu einem Zeitpunkt zugreift. Am Anfang und Ende des Abschnitts verändern Funktionen den Zustand des Semaphors.

Komplexe Datenstrukturen

Ist das zu schützende Objekt mehr als 24 Bit breit oder handelt es sich gar um eine komplexe Datenstruktur, dann helfen andere Methoden weiter. Applikationsprogrammierer denken hier vielleicht an die von dem Niederländer Dijkstra konzipierten Semaphore. Ein Semaphor ist zunächst nichts anderes als eine Variable, für die genau zwei Operationen definiert sind. Ein Prozess führt vor dem Betreten des kritischen Abschnitts die P-Operation aus, nach dessen Verlassen die V-Operation.

P steht für das niederländische Verb passeren (durchmarschieren). Diese Operation dekrementiert den Semaphor und legt, falls er einen Wert kleiner als null eingenommen hat, den zugehörigen Prozess schlafen. Die V-Operation (vom niederländischen vrijgeven, freigeben) inkrementiert die Variable. Sie weckt, falls die Variable danach noch negativ oder gleich null ist, einen schlafend auf das gesperrte Objekt wartenden Prozess wieder auf.

Semaphore

Auch wenn es genau genommen innerhalb des Linux-Kernels keine Tasks gibt, stehen Semaphore im Betriebssystemkern zur Verfügung. Denn die letzte Kern-Technik-Folge [1] hat gezeigt: Es gibt Kernel-Threads und darüber hinaus bei den Gerätetreibern die klassischen Treiberinstanzen. Letztere führt der Kernel im Auftrag eines Prozesses aus. Konkurrieren die Codestücke um Betriebsmittel, können zu deren Schutz Kernel-Semaphore verwendet werden.

Bevor ein Programm einen Semaphor verwenden kann, muss es ihn definieren

Kritische Abschnitte und Race Conditions

Ein kritischer Abschnitt ist ein Codestück, das auf mehrfach genutzte Betriebsmittel (Speicher, E/A) zugreift, die andere Instanzen zur selben Zeit verwenden wollen. Wenn zum Beispiel ein Interrupt den Ablauf einer Funktion unterbricht, arbeitet die zugehörige Interrupt-Serviceroutine (ISR) quasi-parallel zur Funktion. Sobald beide auf dieselbe Variable zugreifen, liegt bereits ein kritischer Abschnitt vor. Dann hängt das Ergebnis einer Operation vom Zufall ab, man spricht von einer Race Condition. Die Folgen sind nicht kalkulierbar und reichen von Rechenfehlern bis zum völligen Absturz. Um eine Race Condition zu vermeiden, muss der kritische Abschnitt so geschützt werden, dass ihn zu jedem Zeitpunkt nur ein Prozess durchläuft.

Kritische Abschnitte gibt es nicht nur beim Zugriff auf gemeinsame Daten. Wie in der Kern-Technik-Folge 2 bereits gezeigt, kann es bei der Synchronisation zwischen Codestücken ebenfalls zu einer Race Condition kommen.

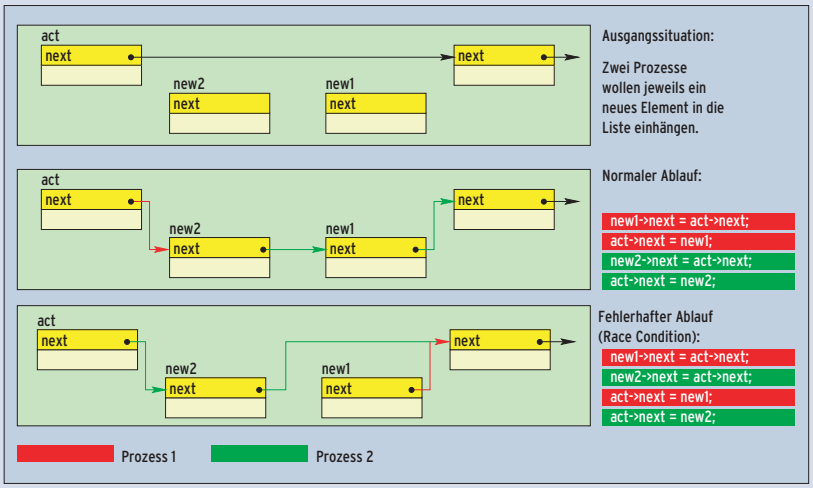


Abbildung 2: Verändern zwei Prozesse gleichzeitig eine verkettete Liste, dann geraten die Zeiger durcheinander. Solche Fehler lassen sich mit Locking-Techniken vermeiden.

Listing 1: »atomic_...()«-Funktionen

```
01 #include <asm/atomic.h>
02 ...
03 static atomic_t Status = ATOMIC_INIT( 4 );
04 int s;
05 ...
06    atomic_add( 2, &Status );
07    s = atomic_read( &Status );
08    while( atomic_sub_and_test( 2, &Status ) ) {
09            ...
10    }
11    atomic_set( &Status, 4 );
```

(»static struct semaphore Liste;«) und initialisieren (»sema_init(&Liste, 2);«). Die Prototypen und Makros dafür befinden sich in der Headerdatei »asm/semaphore.h«. Der Initialwert des Semaphors (im Beispiel der Wert »2«) legt fest, wie viele Prozesse sich gleichzeitig im kritischen Abschnitt aufhalten dürfen. Ist es genau einer, spricht man von gegenseitigem Ausschluss (mutual exclusion). Ein solcher Semaphor heißt daher Mutex.

Mutexe

Da die meisten Semaphore Mutexe sind, gibt es für sie im Kernel eigene Makros. »DECLARE_MUTEX(*Mutexname*)« definiert und initialisiert sie mit dem Namen eines Mutex als Parameter. Die Funktion zum Betreten eines durch einen Semaphor geschützten kritischen Abschnitts heißt »down()« (schließlich wird die Semaphor-Variable erniedrigt) beziehungsweise »down_interruptible()«. Letztere weckt den schlafenden Kernel-Thread oder die schlafende Treiberinstanz, wenn ein Signal eintrifft. In diesem Fall gibt die Funktion einen Wert ungleich null zurück (siehe Listing 2).

Listing 2: Semaphor im Kernel

```
01 #include <asm/semaphore.h>
02 ...
03     if( down_interruptible( &ListenMutex ) ) {
04         return -ERESTART;
05     }
06     ... // Hier ist der kritische Abschnitt
07     up( &ListenMutex );
```

Listing 3: Spinlock im Interruptkontext

```
01 #include <asm/spinlock.h>
02 ...
03 static spinlock_t ListLock = SPIN_LOCK_UNLOCKED;
04 ...
05 {
06     unsigned long irqflags;
07
08     ...
09     spin_lock_irqsave( &ListLock, &irqflags );
10     ... // kritischer Abschnitt
11     spin_unlock_irqrestore( &ListLock, &irqflags );
12     ...
13 }
```

Listing 4: Sequence Lock im Interruptkontext

```
01 #include <linux/seqlock.h>
02 ...
03 static seqlock_t slock;
04 static unsigned long seq;
05 ...
06     do {
07         seq = read_seqbegin( &slock );
08         // Daten lesen
09     } while( read_seqretry( &slock, seq ) );
```

Neben den beiden blockierenden Varianten, einen kritischen Abschnitt zu betreten, gibt es die nicht blockierende Form »down_trylock()«. Diese Funktion liefert sofort, ohne zu warten, einen Rückgabewert. Wenn der kritische Abschnitt betreten werden darf, ist dieser Wert »0«. Die Funktion zum Verlassen des kritischen Abschnitts heißt »up()«.

Häufig – besonders bei der Treiberprogrammierung – sind kritische Abschnitte von ISRs (Interrupt-Service-Routinen) vor konkurrierenden Zugriffen zu schützen. In einem solchen Kontext helfen Semaphore nicht weiter, wollen sie doch die zugehörigen Prozesse schlafen legen. Mit einer ISR geht das nicht, doch Linux bietet für diesen Anwendungsfall andere Locking-Techniken.

Interrupts sperren

Im einfachsten Fall sperrt der Kernel den Interrupt für die Zeit des Zugriffs. Wollen beispielsweise ein Kernel-Thread und eine ISR eine globale Liste bearbeiten, muss der Kernel-Thread, wenn er den kritischen Abschnitt betritt, den Interrupt sperren. Die ISR selbst braucht gar nichts zu tun, da sie nicht durch einen Kernel-Thread unterbrochen werden kann (siehe Abbildung 3). Für Einprozessormaschinen funktioniert das Verfahren.

Auf einem Multiprozessorsystem greift die Interruptsperre nicht. Hier können Kernel-Thread und ISR auf verschiedenen Prozessoren laufen, so dass der Kernel-Thread zunächst das Ende der gerade aktiven ISRs auf sämtlichen Prozessoren abwarten müsste, um dann auf allen Prozessoren die ISRs zu sperren. Für ein auf Effizienz ausgerichtetes Betriebssystem wie Linux ist das keine Alternative. Für Mehrprozessormaschinen bleibt damit nur eine Lösung: Die ISR muss aktiv warten, bis das andere Codestück den Zugriff auf das gemeinsame Betriebsmittel (hier die Liste) abgeschlossen hat.

Warteschleife

Statt zu schlafen und die CPU einem anderen Prozess zur Verfügung zu stellen, durchläuft die ISR so lange eine Schleife, bis der kritische Abschnitt frei ist. Diese Methode heißt Spinlocking, das aktive Warten in der Schleife selbst ist das Spinning. Der Einsatz von Spinlocks hat drei Konsequenzen:

- Spinlocks funktionieren nur auf Mehrprozessorsystemen.
- Damit die Latenzzeiten kurz bleiben, ist der kritische Abschnitt so kurz wie möglich zu halten.
- Während eines durch Spinlocks gesicherten Bereichs darf man den zugreifenden Kernel-Thread nicht schlafen legen.

Der erste Punkt macht deutlich, dass die Entscheidung für eine Locking-Technik tatsächlich von der eingesetzten Hardware und der Konfiguration des Kernels abhängt: Interruptsperre auf Einprozessormaschinen, Spinlocks auf Mehrprozessorsystemen (SMP, Symmetric Multiprocessing). Glücklicherweise muss sich der Programmierer über diese Differenzierung keine Gedanken machen. Er kann auch auf Einprozessormaschinen Spinlock-Makros verwenden, die der Compiler transparent zu Interruptsperren expandiert.

Wie bei den Semaphoren definiert und initialisiert der Programmierer einen

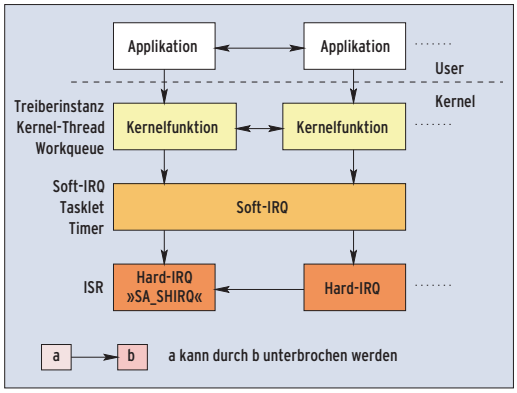


Abbildung 3: In einem Linux-Systems können nur bestimmte Teile einander unterbrechen. Wer die Zusammenhänge kennt, setzt die Schutzmechanismen für kritische Abschnitte gezielt ein.

Tabelle 2: Read-Write-Locks

Sperren	Entsperren
read_lock()	read_unlock()
write_lock()	write_unlock()
read_lock_irq()	read_unlock_irq()
write_lock_irq()	write_unlock_irq()
read_lock_irqsave()	read_unlock_irqsave()
write_lock_irqsave()	write_unlock_irqsave()
read_lock_bh()	read_unlock_bh()
write_lock_bh()	write_unlock_bh()

Tabelle 3: Sequence-Locks

Sperren	Entsperren
write_seqlock()	write_sequnlock()
write_seqlock_irq()	write_sequnlock_irq()
write_seqlock_irqsave()	write_sequnlock_irqrestore()
write_seqlock_bh()	write_sequnlock_bh()
write_seqlock_bh()	write_sequnlock_bh()

Spinlock zunächst als Objekt. In seinen Code bindet er dazu die Kernel-Headerdatei »asm/spinlock.h« ein. Im Regelfall weist er dem Spinlock den Wert »SPIN_LOCK_UNLOCKED« zu, der kritische Abschnitt ist also zunächst frei. Vor dem Betreten des kritischen Abschnitts ruft er die Funktion »spin_lock_irqsave()« auf, beim Verlassen »spin_unlock_irqrestore()« (siehe Listing 3).

Feinarbeit

Semaphore und Spinlocks gibt es noch in weiteren Varianten. Sie wurden eingeführt, um den Kernel besser unterbrechbar und Linux auf diese Weise schneller zu machen. Denn ist der Kernel an mehr Stellen unterbrechbar, müssen andere Programme weniger warten. Die gute Unterbrechbarkeit sorgt für kurze Latenzzeiten und damit wiederum für kurze Antwortzeiten.

Aus der Überlegung heraus, dass oft nur der schreibende Zugriff kritisch ist, sind die Lese-Schreib-Semaphore entstanden. Sie sind vom Typ »struct rw_semaphore«. Die Operationen auf das Objekt sind »down_read()«, »up_read()« (zum Lesen) und »down_write()« und »up_write()« (zum Schreiben). Die Lese-Schreib-Semaphore lassen mehrere Prozesse gleichzeitig lesen. Exklusiv wird der Zugriff erst, wenn ein Prozess die geschützten Daten modifizieren will.

Für Spinlocks sieht die Welt noch komplizierter aus. Denn hier gibt es nicht nur eine Lese-Schreib-Variante, sondern zu der bereits vorgestellten Ausprägung mit Interruptsperre noch eine Abwandlung ohne Interruptsperre, eine mit reiner Soft-IRQ-Sperre und eine, bei der Interrupts gesperrt, die Flags aber nicht gesichert werden.

Die Funktionen für Spinlocks ohne Interruptsperre sind »spin_lock()« und »spin_unlock()«. Sie lassen sich jedoch nur wie Semaphore, also bei einem konkurrierenden Zugriff durch Kernel-Threads oder Treiberinstanzen verwenden. Dafür sind sie aber bei kurzen, kritischen Abschnitten wesentlich effizienter, da sie einen sonst erforderlichen Prozesswechsel vermeiden.

Eine weitere Spinlock-Variante sperrt nur Soft-IRQs. Der kritische Abschnitt kann durch Hard-IRQs weiterhin unterbrochen werden (siehe Abbildung 3). Die zugehörigen Funktionen heißen allerdings nicht, wie zu erwarten wäre, »spin_lock_softirq()« und »spin_unlock_softirq()«, sondern »spin_lock_bh()« und »spin_unlock_bh()«. »bh« steht hier für Bottom Half, eine Art Soft-IRQ der Kernel-Reihe 2.2. Falls ein Kernel-Thread mit einem Timer oder einem Tasklet um ein Betriebsmittel konkurriert (aber nicht mit einer ISR), dann ist diese Technik die erste Wahl.

Es gibt noch eine weitere Variante: Spinlocks, die zwar Interrupts sperren, aber nicht die Interrupt-Flags sichern. Diese genügen, wenn der Zustand der Interrupt-Flags bekannt ist. Innerhalb einer ISR beispielsweise sind Interrupts im Regelfall gesperrt, hier brauchen die Flags nicht explizit abgespeichert zu werden. Die Kernel-Welt ist aber komplizierter, denn Linux bietet noch weitere Möglichkeiten: Jeden der vorgestellten Spinlocks gibt es zusätzlich in einer Lese-Schreib-Variante. Die zugehörigen Objekte heißen Read-Write-Lock (»struct rw_lock«, siehe Tabelle 2).

Sequence-Locks sind neu

Kernel 2.6 führt nun als Neuerung die Sequence-Locks ein. Bei dieser Abwandlung der Read-Write-Locks können nur schreibende Zugriffe einander blockieren. Ein lesender Zugriff blockiert dage-

Die Jahres-CD 2003

Alle Artikel von 2003 im praktischen HTML-Format!



Für 14,95 € *

- alle Ausgaben 01-12/2003 im HTML-Format
- Volltext-indizierte Suchmaschine
- Liste der deutschen Linux User Groups (LUGs)
- Der komplette Abzug von ftp.linux-magazin.de mit allen Listings des Jahrgangs 2003

Jetzt schnell bestellen:

■ Fax: 089 / 99 34 11 99
■ www.linux-magazin.de/produkte
■ e-mail: sales@linux-magazin.de

* Linux-Magazin-Abonnenten: Abo-Upgrade für nur € 6,70 jederzeit möglich!

Ihr Widerrufsrecht: Sie können Ihre Bestellung innerhalb von 14 Tagen beim Linux-Magazin Leser-Service, Stefan-George-Ring 24, D-81929 München, widerrufen. Zur Wahrung der Frist genügt die rechtzeitige Absendung des Widerrufs. Alle Preise gelten innerhalb Deutschlands.

gen nicht. Für den Schreiber ändert sich am Ablauf nichts, er ruft einfach nur vor und nach dem kritischen Abschnitt andere Funktionen auf. Doch beim lesend zugreifenden Prozess sieht die Programmstruktur dann anders aus. Er arbeitet nicht, wie der schreibende, die Folge Sperren, Zugreifen, Freigeben ab, sondern liest in einer Schleife so oft, bis er feststellt, dass sein Lesevorgang nicht durch einen Schreibvorgang unterbrochen wurde.

Wiederholt lesen

Technisch ist der Test, ob ein Schreibvorgang erfolgte oder nicht, durch einen einfachen Zähler realisiert. Stimmen die Werte des Zählers direkt vor und nach dem Lesevorgang überein, gelten die gelesenen Daten als nicht von anderen Prozessen verändert. Weichen die Werte dagegen voneinander ab, springt der Prozess zum Beginn der Schleife.

Auch den Sequence-Lock gilt es, vor Gebrauch zu definieren und zu initialisieren. Hier gibt es sowohl eine statische als auch eine dynamische Methode. Prototypen und Makros befinden sich in der Headerdatei »linux/seqlock.h«. Die statische Methode ist:

```
seqlock_t slock = SEQLOCK_UNLOCKED;
```

Die dynamische Methode:

```
seqlock_t slock;
...
    seqlock_init( &slock );
    ...
```

Der Lesezugriff benötigt zudem eine Variable vom Typ »long«, um darin den Zustand des Sequence-Locks abzulegen. Dieser Zustand wird mit Hilfe der Funktion »read_seqretry()« überprüft. Für den lesenden Zugriff ergibt sich damit eine Struktur wie in Listing 4. Auch die Sequence-Locks gibt es in den Varianten mit und ohne Interruptsperre beziehungsweise mit und ohne Sperre der Soft-IRQs (Tabelle 3).

Der große Kernel-Lock

Zu erwähnen ist noch der große Kernel-Lock. Dieser Lock ist nichts anderes als ein globaler Spinlock. Die Funktion »lock_kernel()« fordert den Spinlock für den ganzen Kernel an und gibt ihn mit »unlock_kernel()« wieder frei. Der globale Kernel-Lock eignet sich nur für Anwendungen, die im Prozesskontext des Kernels abgearbeitet werden; Interrupts sind dabei nicht gesperrt.

Der parallele Zugriff auf globale Daten ist allerdings nur die einfachste Form eines kritischen Abschnitts. Entsprechend leicht ist es, die Gefahr zu erkennen und mit den vorgestellten Mechanismen zu vermeiden. Es ist dagegen nicht so einfach, kritische Abschnitte zu erkennen, die aufgrund der Synchronisationen zwischen unabhängigen Verarbeitungssträngen entstehen.

Die Qual der Wahl

Dass bei bisherigen Kernelversionen allein der Programmierer für die Synchronisation verantwortlich war, führte und führt zu vielen Fehlern, insbesondere in Treibern. Besserung schaffen hier das Completion-Objekt und die Funktion »wait_event_interruptible()« von Kernel 2.6. Beide Mechanismen basieren auf den hier vorgestellten Methoden und sind in ihrer Verwendung bereits in [1] vorgestellt worden.

Race Condition durch Reordering

Eine unerwartete Quelle für Race Conditions ist das Reordering moderner Compiler und Prozessoren, sie optimieren schon mal die Reihenfolge der Befehlsabarbeitung. Das ist durchaus in Ordnung, wenn das Endergebnis der Operation gleich bleibt. Abbildung 4 zeigt ein solches Reordering. Problematisch ist der gezeigte Fall dann, wenn nach Schritt 2a über den Pointer »act« zugegriffen wird. Das kann beispielsweise durch Code geschehen, der auf einer zweiten CPU läuft. Greift dieser Code dabei auf den nicht initialisierten Zeiger des Elementes »new« zu, steht einer Null-Pointer-Exception nichts mehr im Weg.

Memory Barriers statt Spinlocks

Ein Spinlock könnte zwar leicht Abhilfe schaffen und den kritischen Abschnitt sichern. Wenn aber bekannt ist, dass es sich nur um parallel stattfindende lesende Zugriffe handelt, geht es auch einfacher. In diesem Fall genügt eine so genannte Memory Barrier. Diese Speicherbarriere sorgt dafür, dass die Befehle in der definierten Reihenfolge ablaufen, also kein Reordering stattfindet.

Memory Barriers gibt es in Read-, Write- und Read/Write-Varianten. Bei Ersterer (»rmb()«) werden vor dem nächsten Lesevorgang alle zuvor angestoßenen Lesevorgänge abgeschlossen. Die Write-Barriere »wmb()« sorgt dafür, dass vor dem nächsten Schreibauftrag sämtliche bereits angestoßenen Schreibaufträge durchgeführt worden sind, und die Read/Write-Variante »mb()« kombiniert beide Vorgänge miteinander. Die Prototypen für diese Makros befinden sich in der Headerdatei »asm-i386/system.h«. Darüber hinaus gibt es noch das Makro »barrier()«. Es unterbindet aber nur das Reordering durch den Compiler und greift nicht in die Optimierungstechniken der CPU ein.

Auch beim E/A-Zugriff auf Hardware tritt gelegentlich Reordering auf. Falls also die Reihenfolge relevant ist, in der ein Treiber die Hardwareregister beschreibt, bringt auch hier eine Memory-Barriere die Lösung [2].

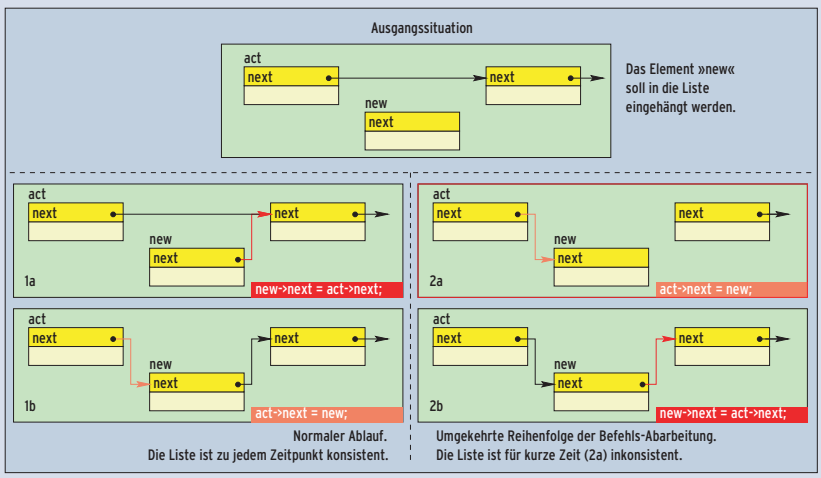


Abbildung 4: Drehen Compiler oder CPU die Folge von Anweisungen um, dann werden Daten kurzfristig inkonsistent. Fehler treten auf, wenn ein anderer Prozess zu diesem Zeitpunkt die Daten liest.

Schutzmechanismen für kritische Abschnitte gehören zu den schwierigsten Kapiteln der Kernelprogrammierung. Zum einen gilt es, kritische Abschnitte überhaupt zu erkennen. Zum anderen müssen Kernelprogrammierer die geeignete Lösungsmethode finden und korrekt anwenden. Die Codefragmente erscheinen auf den ersten Blick einfach, durch die Parallelität sind die Abläufe aber sehr komplex.

Instanzen identifizieren

Wer solche Probleme zu lösen hat, sollte zunächst die beteiligten Instanzen identifizieren und mit diesem Wissen die passende Schutzmethode auswählen: Bei kritischen Abschnitten, beispielsweise zwischen Kernel-Threads und Tasklets, sind Spinlocks sinnvoll (siehe Tabelle 4). Semaphore scheiden aus.

Semaphore bieten sich an, wenn sämtliche Instanzen, die auf den kritischen Abschnitt zugreifen wollen, im Prozess-Kontext ablaufen. Das gilt für Treiberinstanzen (Funktionen des Kernels, die durch die Applikation getriggert werden) und für Kernel-Threads. Davon ausgenommen sind Workqueues, insbesondere die Event-Workqueue. Da diese von mehreren Subsystemen des Kernels und von vielen Treibern verwendet werden, sollte man ihre Codeteile nicht mit Semaphoren schlafen schicken.

Ob sich normale Semaphore oder Lese-Schreib-Semaphore besser eignen, hängt davon ab, ob nur der schreibende Zugriff innerhalb des kritischen Abschnitts vor dem konkurrierenden Zugriff zu sichern ist oder auch der lesende. Da die Lese-Schreib-Semaphoren wesentlich effizienter sind, sollten sie den Vorzug erhalten, wann immer es geht.

Spinlocks sind grundsätzlich überall verwendbar. Da Spinlocks – im Gegensatz zu den Semaphoren – aktiv warten (busy loop, spinning), sollte man damit nur schnell abzuarbeitende kritische Abschnitte schützen.

Kernel 2.6 führt ein ganz neues Gerätemodell ein. Was es damit auf sich hat, wozu es gebraucht wird und wie sich Gerätetreiber in diese Komponente einklinken, das ist Thema der kommenden Kern-Technik-Folge. *(ofr/fjl)* ■

Infos:

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 4: Linux-Magazin 11/03, S. 96

[2] Alesandro Rubini und Jonathan Corbet, „Linux Device Drivers", 2. Auflage, O'Reilly, 2001

[3] Rusty Russell, „Unreliable Guide to Locking": [http://kernelnewbies.org/documents/kdoc/kernel-locking/lklockingguide.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Tabelle 4: Locking-Matrix

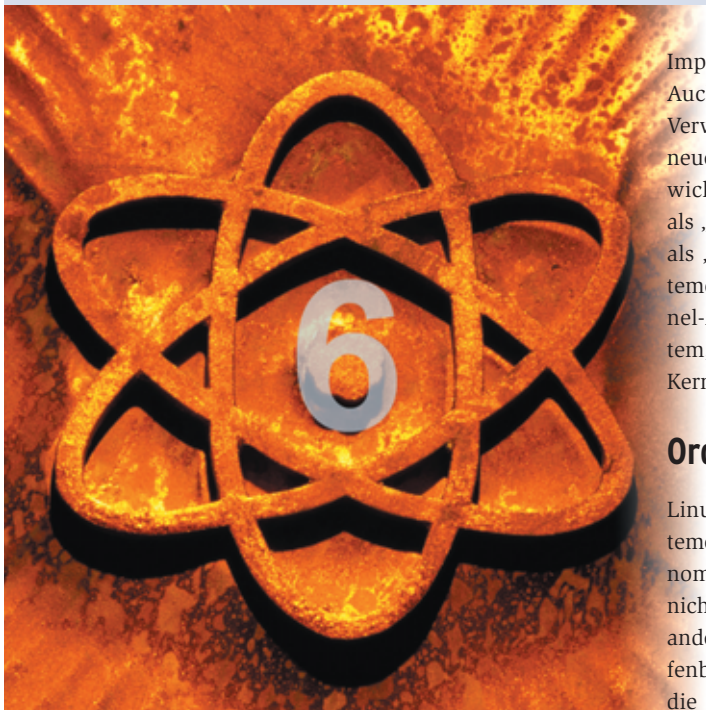
	Treiberinstanz	Kernel-Thread	Workqueue, Event-Workqueue	Soft-IRQ	Tasklet, Timer	Hard-IRQ
Treiberinstanz	Semaphor RW-Semaphor Spinlock RW-Lock Seqlock	Semaphor RW-Semaphor Spinlock RW-Lock Seqlock	Spinlock RW-Lock Seqlock	Spinlock-BH RW-Lock-BH Spinlock-Irqsave RW-Lock-Irqsave Seqlock-Irqsave	Spinlock-BH RW-Lock-BH Spinlock-Irqsave RW-Lock-Irqsave Seqlock-Irqsave	Spinlock-Irqsave RW-Lock-Irqsave Seqlock-Irqsave
Kernel-Thread	Semaphor RW-Semaphor Spinlock RW-Lock Seqlock	Semaphor RW-Semaphor Spinlock RW-Lock	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
Soft-IRQ	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 (siehe Anmerkung 1)	Spinlock-Irqsave RW-Lock-Irqsave Seqlock-Irqsave
Hard-IRQ	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 (siehe Anmerkung 2)

(1) Nur bei unterschiedlichen Tasklets. Dasselbe Tasklet läuft zu einem Zeitpunkt nur einmal.
(2) Nur bei unterschiedlichen ISRs. Dieselbe ISR läuft zu einem Zeitpunkt nur einmal.

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 6

Kern-Technik

Das neue Gerätemodell ordnet Hardware nach Kategorien und bildet die Struktur im Sys-Filesystem ab. Hält sich der Entwickler daran, unterstützt sein Treiber automatisch Powermanagement. Das Gerätemodell besteht aus einem Kernel-API und einem Laufzeitsystem. Eva-Katharina Kunst, Jürgen Quade



Treiber für Hardware machen den größten Teil des Kernels aus. Linux 2.6 führt neue Schnittstellen ein, mit deren Hilfe sich Treiber besser in das Gesamtsystem integrieren: das Gerätemodell und das Sys-Filesystem. Die Informationen darüber sind allerdings spärlich. Kein Wunder, die Komponenten sind brandneu und stecken mitten in der Entwicklung.

API und Laufzeitsystem

Wer in eigenen Gerätetreibern das neue Modell unterstützen möchte, muss im Linux-Quellcode gelegentlich Variablen, Interfaces und Aufrufsemantik recherchieren. Im Kernelbaum finden sich einige Texte, die das Gerätemodell beschreiben (»Documentation/driver-model«), aber in manchen Details von der Implementierung abweichen. Auch beim Namen herrscht Verwirrung. So erscheint das neue Gerätemodell in der Entwicklerdiskussion teilweise als „device model“, aber auch als „driver model“. Das Gerätemodell ist zugleich ein Kernel-API und ein Laufzeitsystem, das Gerätestrukturen im Kernel verwaltet.

Ordnung durch Sysfs

Linus Torvalds hat das Gerätemodell in den Kernel aufgenommen, obwohl es noch nicht ausgereift ist. Wie auch andere Entwickler setzt er offenbar hohe Erwartungen in die neue Komponente. Sie bringt Ordnung in die Gerätelandschaft, hilft beim Powermanagement und verwaltet künftig – als Ersatz für das Device-Filesystem (Devfs) – die Gerätedateien [1].

Das Gerätemodell bildet ab, wie die Prozessoren eines Systems mit den Controllerbausteinen und diese wiederum mit den Peripheriekarten und mit sonstiger Hardware zusammenhängen. Außerdem verleiht das Modell den zugehörigen Softwarekomponenten eine Struktur, beispielsweise den Gerätetreibern. Anhand der im Gerätemodell gesammelten Information ist der Kernel in der Lage, gezielt Powermanagement zu betreiben. So gibt es die Reihenfolge vor, in der das Betriebssystem die Hardware abschaltet: Zunächst müssen die Geräte an einem Bus heruntergefahren werden, bevor der Bus selbst und schließlich der Prozessor in einen Stromsparzustand übergehen.

Das Sys-Filesystem (Sysfs) ist wie das Proc-Filesystem virtuell: Der Kernel erzeugt die Verzeichnisse und Dateien dynamisch, keine Festplatte muss sie speichern. Der durch das Sys-Filesystem aufgespannte Verzeichnisbaum spiegelt die Struktur der Hardware und der zugehörigen Software des jeweiligen Linux-2.6-Systems wider. Über das Sysfs kommt auch der Anwender mit dem Gerätemodell in Berührung. Er muss es nur noch in den Verzeichnisbaum einhängen:

```
mount -t sysfs sysfs /sys
```

Um in Userspace-Programmen Informationen aus dem Sys-Filesystem zu lesen, gibt es bereits die Bibliothek »libsysfs« [4]. Abbildung 1 zeigt, in welche Kategorien sich Geräte und Treiber aufteilen und welche zusätzlichen Schnittstellen, zum Beispiel für den Download von Firmware [2], zur Verfügung stehen.

Mehrfacheinträge

Entsprechend den Kategorien erscheinen einzelne Geräte und Treiber im Sys-Filesystem an mehreren Stellen. Eine PCI-Netzwerkkarte sortiert das Gerätemodell sowohl in die Kategorie Bus (PCI) als auch in die Kategorie Geräteklasse/Netzwerk ein. Mehrfacheinträge realisiert es mit Hilfe symbolischer Links.

Das Sys-Filesystem zeigt nicht nur die Gerätestruktur. Die Blätter des Baums sind lesbare und beschreibbare Pseudodateien. Wie beim Proc-Filesystem lassen sich hierüber Attribute der Hard- und Software auslesen und setzen. Ob Herstellername, Versionsnummer, Gerätezustand oder die Übertragungsstatistik einer Netzwerkkarte – die Nutzungsmöglichkeiten sind fast unbegrenzt.

Das An- und Abmelden von Geräten und Treibern beim Gerätemodell vollzieht sich oft implizit. PCI-, USB- oder Netzwerksubsystem übernehmen diese Aufgabe. Schon wenn ein Treiber die in [3] vorgestellte Funktion »register_chrdev()« aufruft, erstellt der Kernel einen Eintrag. Dennoch sollte der Programmierer das Gerätemodell explizit unterstützen (siehe Abbildung 2).

Will er les- und schreibbare Attribute erzeugen, sollte er dafür das Gerätemodell verwenden. In anderen Fällen ist es obligatorisch, nämlich wenn

- Geräte nicht über USB, PCI oder über ein anderes, standardisiertes Bussystem angeschlossen werden,
- neue Bussysteme eingefügt oder
- wenn neue Geräteklassen definiert werden sollen.

Der Entwickler muss also in seinem Code den Treiber und das Gerät beim Gerätemodell anmelden und sinnvolle Attribute freischalten.

Treiber anmelden

Wenn sich der Treiber beim PCI-, USB- oder IDE-Subsystem anmeldet, hängt es ihn im Sys-Filesystem automatisch unterhalb des zugehörigen Busobjekts ein. Das Gleiche gilt für I2C- (zwischen Hardware-Chips), EISA- und Microchannel-Busse. In allen anderen Fällen aber muss der Treiber ein neues Bussystem definieren oder sich selbst unterhalb des so genannten Plattformbusses einklinken. Weil innerhalb des Gerätemodells jeder Treiber sich einem Bussystem zuordnen muss, dient der Plattformbus als Container für alle Treiber, die nicht zu anderen Bussen gehören.

Das Einhängen des Treibers unterhalb des Plattformbusses folgt dem Standard: Zunächst definiert und initialisiert der Programmierer ein Objekt, in der Sprache C also eine Datenstruktur. Dann übergibt er dieses Objekt dem Kernel, der den Rest erledigt. Wird der Treiber nicht mehr gebraucht, muss er sich beim Kernel abmelden. Der Treiber in Listing 1 beispielsweise definiert die Datenstruktur »struct device_driver« und belegt initial die Elemente »name« und »bus« (Zeilen 29 bis 32). Der Datentyp »struct device_driver« und das Symbol »platform_bus_type« sind in der Headerdatei »linux/device.h« definiert.

Bei der Initialisierung übergibt der Treiber durch den Aufruf der Funktion »dri-

Listing 1: Treiber für ein virtuelles Gerät

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/device.h>
06 #include <linux/completion.h>
07
08 #define DRIVER_MAJOR 240
09
10 MODULE_LICENSE("GPL");
11
12 static struct file_operations Fops;
13 static DECLARE_COMPLETION( DevObjectIsFree );
14 static int Frequenz; // Zustandsvariable des Gerätes
15
16 static void mydevice_release( struct device *dev )
17 {
18     complete( &DevObjectIsFree );
19 }
20
21 struct platform_device mydevice = {
22     .name   = "MyDevice",
23     .id     = 0,
24     .dev = {
25         .release = mydevice_release,
26     }
27 };
28
29 static struct device_driver mydriver = {
30     .name = "MyDevDrv",
31     .bus = &platform_bus_type,
32 };
33
34 static ssize_t ReadFreq( struct device *dev, char *buf )
35 {
36     sprintf(buf, "Frequenz: %d", Frequenz );
37     return strlen(buf)+1;
38 }
39
40 static ssize_t WriteFreq( struct device *dev, const char *buf, size_t
   count )
41 {
42     Frequenz = simple_strtoul( buf, NULL, 0 );
43     return strlen(buf)+1;
44 }
45
46 static DEVICE_ATTR( freq ,S_IRUGO|S_IWUGO, ReadFreq, WriteFreq );
47
48
49 static int __init DrvInit(void)
50 {
51     if(register_chrdev(DRIVER_MAJOR, "MyDevice", &Fops) == 0) {
52         driver_register(&mydriver);           // register the driver
53         platform_device_register( &mydevice );// register the device
54         mydevice.dev.driver = &mydriver;      // now tie them together
55         device_bind_driver( &mydevice.dev );  // links the driver to
                                                    the device
56         device_create_file( &mydevice.dev, &dev_attr_freq ); //
   attributes
57         return 0;
58     }
59     return -EIO;
60 }
61
62 static void __exit DrvExit(void)
63 {
64     device_remove_file( &mydevice.dev, &dev_attr_freq );
65     device_release_driver( &mydevice.dev );
66     platform_device_unregister( &mydevice );
67     driver_unregister(&mydriver);
68     unregister_chrdev(DRIVER_MAJOR,"MyDevice");
69     wait_for_completion( &DevObjectIsFree );
70 }
71
72 module_init( DrvInit );
73 module_exit( DrvExit );
```

ver_register()« das Objekt an den Kernel (Zeile 52). Sobald »driver_register()« abgearbeitet ist, erscheint in einem gemounteten Sys-Filesystem das neue Verzeichnis »/sys/bus/platform/drivers/MyDevDrv/«. Die Funktion »driver_unregister()« entfernt diesen Eintrag wieder (Zeile 67).

Geräte anmelden

Module müssen nicht nur den Treiber beim Laufzeitsystem des Gerätemodells anmelden, sondern auch das Gerät selbst, wenn es durch den Treiber und nicht durch ein PCI-, USB- oder sonstiges Subsystem automatisch erkannt wird. Solche Geräte werden im Modell als Plattform- oder Systemgeräte bezeichnet. Systemgeräte sind im Wesentlichen die Bausteine des Rechnerkerns selbst, also die CPU, der Interrupt-Controller oder auch der Timer. Einträge hierfür finden sich unter dem Ordner »/sys/devices/system/«. Alle übrigen Geräte sind Plattformgeräte. Für sie werden die Einträge im Verzeichnis »/sys/devices/legacy/« erstellt.

Ein Modul definiert ein Gerät als Objekt und initialisiert es bei dessen Anmeldung, genauso wie eben bei den Treibern gezeigt (siehe Listing 1, Zeilen 21 bis 27). Allerdings muss der Code für ein Gerät zudem eine »release()«-Funktion enthalten (Zeile 16). Diese ruft der Kernel auf, sobald das Gerätemodell das Geräteobjekt nicht mehr benötigt. Das Modul wiederum muss sicherstellen, dass der zum Objekt gehörige Speicher erst nach Aufruf der »release()«-Funktion freigegeben wird. Das lässt sich am einfachsten mit einem Completion-Objekt [3] bewerkstelligen (Zeilen 13, 18 und 69), auf das bei der Deinitialisierung des Treibers gewartet wird.

Die Funktion »platform_device_register()« übergibt das Objekt dem Gerätemodell (Zeile 53). Daraufhin erstellt der Kernel das Verzeichnis »/sys/devices/legacy/MyDevice/«. Die Routine »platform_device_unregister()« lässt ihn den Verzeichniseintrag anschließend wieder entfernen (Zeile 66).

So einfach ist es, Treiber- und Geräteobjekte im Gerätemodell zu registrieren – allerdings stehen beide Objekte noch in keiner Beziehung zueinander. Das Gerätemodell weiß also noch nicht, dass das Gerät durch den Beispieltreiber bedient wird, und umgekehrt, dass dieser Treiber für das soeben definierte Gerät verantwortlich ist.

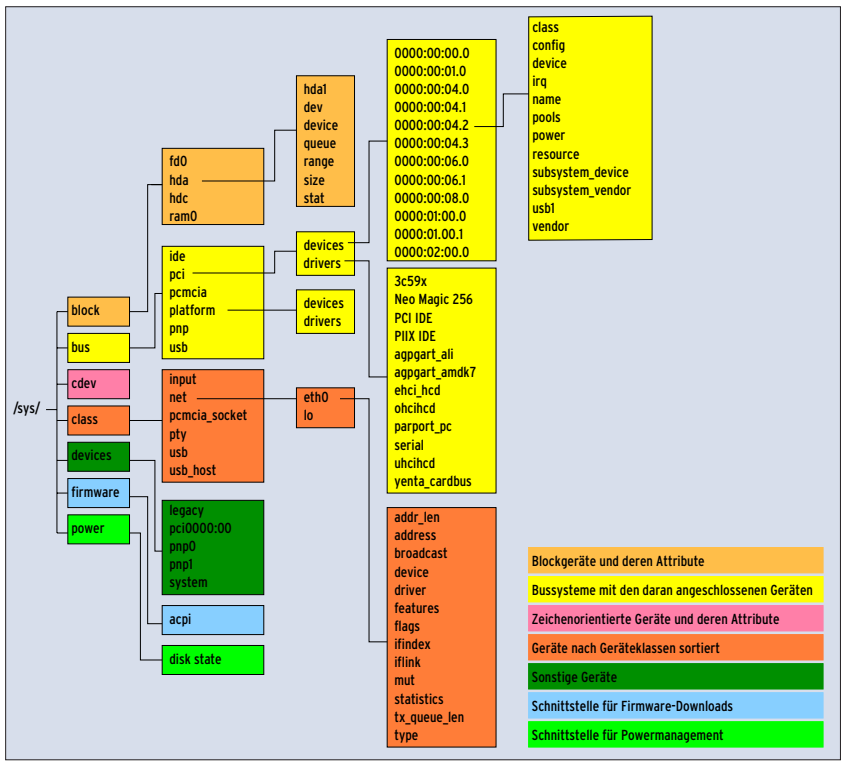


Abbildung 1: Das Sys-Filesystem kategorisiert die Informationen des Gerätemodells und stellt sie in einem Verzeichnisbaum dar. Für jedes Objekt, etwa einen Treiber, existiert ein eigenes Verzeichnis. Die Objektattribute, zum Beispiel Zustandsdaten des Treibers, sind als Dateien repräsentiert.

Verknüpfungspunkte zum Geräteobjekt schaffen

Die Funktion »device_bind_driver()« stellt diese Verbindung her. Parameter der Funktion ist die Adresse des Geräteobjekts. Vor ihrem Aufruf muss allerdings die Adresse des Treiberobjekts »mydriver« im Geräteobjekt »mydevice« eingetragen sein:

```
mydevice.dev.driver = &mydriver;
device_bind_driver( &mydevice.dev );
```

Das Gerätemodell erzeugt einen Link vom Verzeichnis »/sys/bus/platform/drivers/MyDriver« hin zu »/sys/devices/legacy/MyDevice«.

Bisher hat das Gerätemodell Verzeichniseinträge im Sys-Filesystem verändert. Richtig spannend wird es, wenn sich Parameter und Zustände von Gerät und Treiber über Attributdateien einfach lesen und setzen lassen. Dazu ist zweierlei notwendig: Erstens sind die Funktionen zu definieren, die der Kernel aufruft, wenn der Anwender ein Attribut liest oder schreibt. Zweitens müssen die Attributdateien als solche im Sys-Filesystem angelegt werden (siehe Kasten „Attribute programmieren“).

Lesefunktionen für Attributdateien besitzen immer zwei Parameter. Der erste spezifiziert das zugehörige Objekt, beispielsweise das Geräteobjekt oder das Treiberobjekt. Der zweite Parameter ist die Adresse eines Speicherbereichs, in dem die Attributdaten abgelegt werden sollen. Da normalerweise über das Sys-Filesystem keine großen Datenmengen ausgetauscht werden, ist dieser Puffer in seiner Länge auf eine Speicherseite limitiert. Durch diese feste Grenze ist es einfacher, in eigenen Treibern das Sys-Filesystem zu unterstützen.

Daten, die Treiber und Userspace austauschen, sollten in Ascii kodiert sein. Auf diese Weise kann man mit Systemprogrammen einfach auf die Attribute zugreifen. So schreibt »echo "1" > /sys/legacy/MyDevice/attr« den Wert »1« in das Attribut »attr«, während »cat /sys/legacy/MyDevice/attr« den entsprechen-

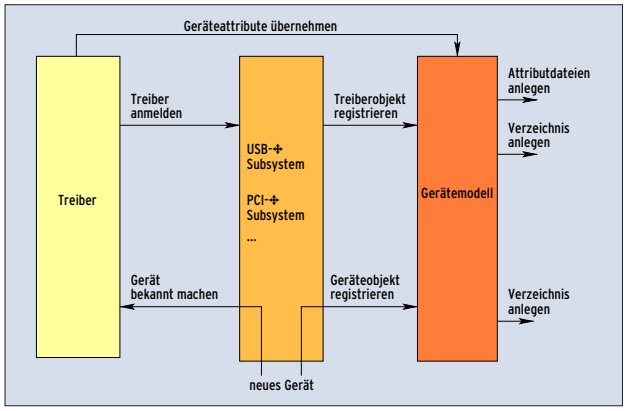


Abbildung 2: Wenn ein Treiber eigene Attribute unterstützen möchte oder wenn er sich nicht bei einem Treiber-Subsystem (wie PCI, USB) anmeldet, muss der Programmierer das Gerätemodell explizit ansprechen.

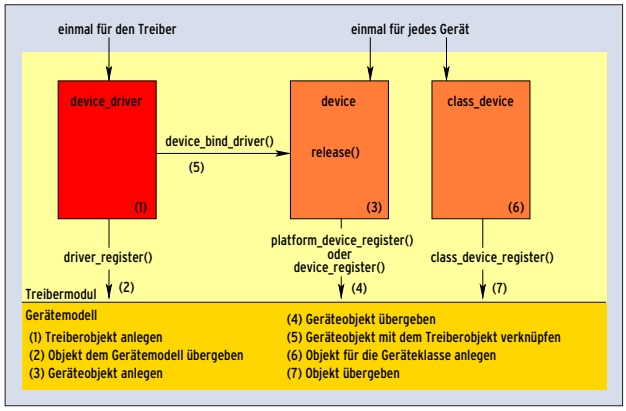


Abbildung 3: Ein Treibermodul muss Objekte für den Treiber selbst und für jedes Gerät definieren und dem Gerätemodell übergeben. »device_bind_driver()« ordnet beide Objekte einander zu.

den Wert liest und ausgibt. Schreibfunktionen besitzen neben den beiden erwähnten noch einen dritten Parameter, der die Anzahl der zu schreibenden Bytes angibt. Übrigens transferieren die Lese- und die Schreibfunktion Daten nur innerhalb des Kernelspace. Für den Transfer zum Userspace ist das Gerätemodell selbst verantwortlich.

Sind die Zugriffsfunktionen auf die Attributdateien definiert, müssen die Dateien im Sysfs erstellt werden. Die beiden Makros »DRIVER_ATTR()« und »DEVICE_ATTR()« definieren und initialisieren die Objekte, die die Funktionen »driver_create_file()« und »device_create_file()« dann dem Gerätemodell übergeben.

Attribute im Sys-Filesystem

Die Namen der über die Makros »DRIVER_ATTR()« und »DEVICE_ATTR()« erzeugten Objekte ergeben sich aus dem ersten Parameter mit dem Vorsatz »driver_attr_« beziehungsweise »device_attr_«. So definiert die folgende Zeile das Objekt »driver_attr_version«:

```
static DRIVER_ATTR( version, S_IRUGO,
   ShowVersion, NULL );
```

Der zweite Parameter spezifiziert die Zugriffsrechte auf die Attributdatei. Die Headerdatei »linux/stat.h« deklariert die möglichen Werte (siehe Tabelle 1). Da dieses Beispiel nur die Lesefunktion »ShowVersion()« verwendet, setzt das Makro auch nur Leserechte (hier für den User, die Gruppe und für die Übrigen). Der letzte Parameter ist »NULL« und verweist damit nicht auf eine Schreibfunktion, die sonst hier erscheinen würde. Die Bedeutung der Parameter gilt für »DEVICE_ATTR()« entsprechend (siehe Kasten „Attribute programmieren" und Listing 1).

Die Funktionen »driver_remove_file()« und »device_remove_file()« entfernen die angelegten Dateien wieder, wenn der Treiber beziehungsweise das Gerät nicht mehr gebraucht wird.

Attributdateien für PCI

Bei PCI-Geräten enthalten die Datenstrukturen »struct pci_driver« und »struct pci_dev« jeweils ein »struct device_driver« und ein »struct device«. Kein Problem also, auch in diesem Fall Attributdateien zu erzeugen. Bei der Treiberinitialisierung legt »driver_create_file()« die Attributdateien des Treibers an, bei der Geräte-Initialisierung »device_create_file()« die des Geräts.

```
driver_create_file( &pcidrv.driver,
    &driver_attr_mytext );
device_create_file( &pcidev->dev,
    &dev_attr_mytext );
```

Das Vorgehen, um ein Gerät bei einer Geräteklasse anzumelden, ist vertraut:

Attribute programmieren

Das Makro »DEVICE_ATTR()« (Listing 2, Zeile 11) definiert mit »freq« die Datenstruktur »dev_attr_freq« (»dev_attr_« wird dem Variablennamen vorangestellt). Es erlaubt lesenden und schreibender Zugriff und legt die Zugriffsfunktionen fest. Während der Geräte-Initialisierung in »DeviceProbe()« legt »device_create_file()« (Zeile 16) die Attributdatei an.

Der Kernel ruft »WriteFreq()« auf, wenn eine Applikation in die Attributdatei »freq« schreibt. Die zu schreibenden Daten befinden sich bereits im Kernelspace. Liest eine Applikation die Attributdatei »freq«, ruft der Kernel »ReadFreq()« auf. Die angeforderten Daten können direkt in den Puffer »buf« geschrieben werden. Die Funktion gibt die Anzahl der gelesenen Bytes zurück.

Bei der Geräte-Deinitialisierung in »DeviceRemove()« entfernt »device_remove_file()« (Zeile 23) die Attributdatei wieder aus dem Sys-Filesystem.

Listing 2: Attribute programmieren

```
01 static ssize_t ReadFreq( struct device *dev, char *buf )
02 {
03     ...
04 }
05
06 static ssize_t WriteFreq( struct device *dev, const char
   *buf, size_t count )
07 {
08     ...
09 }
10
11 static DEVICE_ATTR( freq, S_IRUGO|S_IWUGO, ReadFreq,
   WriteFreq );
12
13 static int DeviceProbe( ... )
14 {
15     ...
16     device_create_file( &mydevice.dev, &dev_attr_freq );
17     ...
18 }
19
20 static int DeviceRemove( ... )
21 {
22     ...
23     device_remove_file(&mydevice.dev, &dev_attr_freq );
24     ...
25 }
```

Meldet sich ein Treiber bei einem Subsystem – zum Beispiel beim Netzwerksubsystem – an, sorgt dieses für die Registrierung beim Gerätemodell.

Geräteklassen

Gibt es dagegen kein zugehöriges Subsystem, muss der Treiberentwickler selbst Hand anlegen – etwa bei Geräten, die zu den Klassen »input« oder »pcmcia _socket« gehören. Dazu definiert er ein Objekt, initialisiert es und übergibt es dem Gerätemodell. Die Übergabe erfolgt während der Geräte-Initialisierung. Dabei geht der Kernelprogrammierer folgendermaßen vor:

- Er definiert ein Objekt von Typ »struct class_device«, zum Beispiel mit Namen »myclassdev«. Es dient als Container, um ein Geräte- und ein Treiberobjekt an das Gerätemodell zu übergeben.
- Um Geräte- und Treiberobjekt miteinander zu verknüpfen, trägt er die Adresse des Treiberobjekts »mydriver« (Typ »struct device_driver«) im Geräteobjekt »mydevice« ein (»struct device«).
- Jetzt trägt er die Adresse von »mydevice« in »myclassdev.dev« ein.
- Nun muss er noch festlegen, unter welcher Klasse (also in welchem Sysfs-Verzeichnis) der neue Eintrag steht. Für Input-Klassen schreibt er die Adresse der globalen Variablen »input_class« (definiert in »linux/input.h«) nach »myclassdev.class«.
- Er kopiert den Verzeichnisnamen, unter dem die Einträge später auftauchen, in »myclassdev.class_id«.
- Er übergibt mit »class_device_register()« das Objekt an den Kernel:

```
static struct class_device myclassdev;
...
    mydevice->driver = &mydriver;
    myclassdev.dev = &mydevice;
    myclassdev.class = &input_class;
    strlcpy( (void *)&myclassdev.class_id,
     "MyDev", 6 );
    class_device_register( &myclassdev );
```

Das Gerätemodell legt daraufhin das Verzeichnis »MyDev« unterhalb von »/sys/class/input« an und setzt symbolische Links auf die entsprechenden Geräte- und Treiberobjekte:

```
device -> ../../../devices/pci0000:00/
0000:00:08.0
driver -> ../../../bus/pci/drivers/pci_drv
```

Auch ein Class-Device kann Attribute besitzen. Die Programmierung unterscheidet sich allein im Namensvorsatz für Makros und Funktionen von dem bereits dargestellten Ansatz (»CLASS« statt »DRIVER« oder »DEVICE«).

Interessanter ist es allerdings, eine eigene Klasse zu definieren. Hierfür sind eine Hotplug- sowie eine Release-Funktion zu implementieren und beide über ein Klassen-Objekt dem Linux-Gerätemodell zu übergeben. Die Hotplug-Funktion wird aufgerufen, sobald sich ein Treiber bei der Klasse anmeldet, die Release-Funktion, wenn sich der Treiber wieder abmeldet.

Tabelle 1: Zugriffsrechte

Symbol	Bedeutung
S_IRWXU	Lesen, Schreiben und Ausführen für den User
S_IRUSR	Lesen für den User
S_IWUSR	Schreiben für den User
S_IXUSR	Ausführen für den User
S_IRWXG	Lesen, Schreiben und Ausführen für die Gruppe
S_IRGRP	Lesen für die Gruppe
S_IWGRP	Schreiben für die Gruppe
S_IXGRP	Ausführen für die Gruppe
S_IRWXO	Lesen, Schreiben und Ausführen für Übrige
S_IROTH	Lesen für Übrige
S_IWOTH	Schreiben für Übrige
S_IXOTH	Ausführen für Übrige
S_IRWXUGO	Kombination von »S_IRWXU|S_IRWXG|S_IRWXO«
S_IALLUGO	Kombination von »S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO«
S_IRUGO	Kombination von »S_IRUSR|S_IRGRP|S_IROTH«
S_IWUGO	Kombination von »S_IWUSR|S_IWGRP|S_IWOTH«
S_IXUGO	Kombination von »S_IXUSR|S_IXGRP|S_IXOTH«

Auf dem FTP-Server [5] ist als Beispiel ein kompletter Treiber zu finden, der die eigene Geräteklasse »GameClass« implementiert. Innerhalb der Klasse erzeugt er ein Klassengerät »GameDevice« und ein virtuelles Gerät »GameClassDevice«.

Peripheriebusse definieren

Eigene Busse definieren ist noch unkomplizierter: Der Programmierer legt ein Busobjekt an, initialisiert es mit dem Namen und registriert es mit der Funktion »bus_register()«:

```
struct bus_type can = {
    .name = "CAN",
};
static int __init MyModulInit(void)
{   ...
    bus_register( &can );
```

Das Konzept des Gerätemodells ist klar strukturiert, aber da der zugehörige Kernelcode noch nicht ganz ausgereift ist, führen kleine eigene Programmierfehler schnell zum Absturz des Systems – dann hilft nur noch ein Reboot.

Fazit und Vorschau

Sich ins neue Gerätemodell einarbeiten lohnt für jeden Kernelprogrammierer. Das Modell ist zukunftsfähig: Es erlaubt ordentliches Powermanagement, zudem wird das Sys-Filesystem viele Aufgaben des überforderten Proc-Filesystems übernehmen. So wird die nächste Kern-Technik-Folge zeigen, wie das Proc-Filesystem in seinem angestammten Bereich funktioniert und wie man es in eigenen Kernelmodulen einsetzt. *(ofr)*

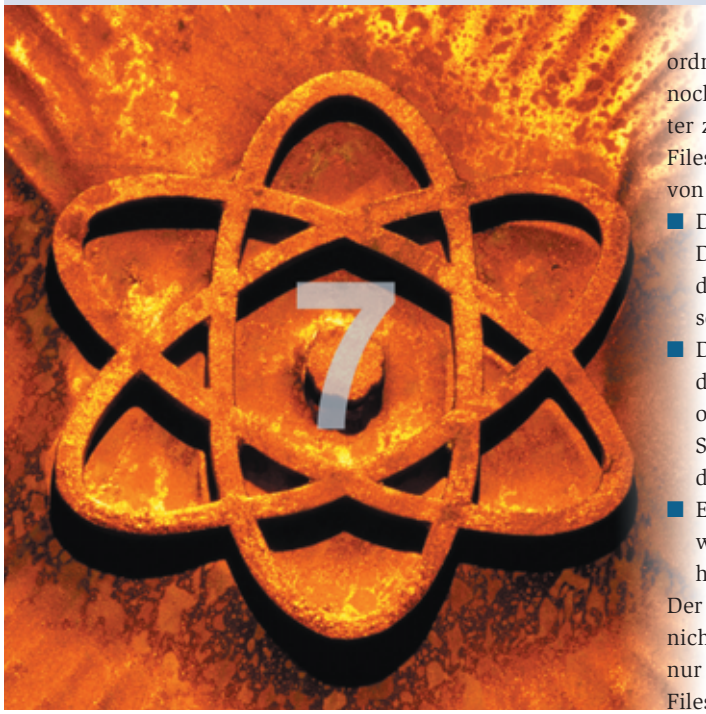
Infos:

[1] Greg Kroah-Hartmann, „udev – A Userspace Implementation of devfs“: [http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Kroah-Hartman-OLS2003.pdf]
[2] Manual Estrada Sainz, „ Firmware Class“, Dokumentation in den Kernelquellen: »Documentation/firmware_class«.
[3] Eva-Katharina Kunst und Jürgen Quade: „Kern-Technik“, Folge 4, Linux-Magazin 10/03, S. 81
[4] Libsysfs: [http://linux-diag.sf.net].
[5] Listings: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/01/Kern-Technik/]

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 7

Kern-Technik

Laufzeit-Informationen des Kernels lassen sich über das Proc-Filesystem auslesen und verändern. Die siebente Kern-Technik-Folge zeigt, wie man in seinem eigenen Code dieses virtuelle Dateisystem unterstützt. Im Kernel 2.6 vereinfachen Sequence Files die Programmierung. Eva-Katharina Kunst, Jürgen Quade



Das Proc-Filesystem (ProcFS) ist ungewöhnlich, denn es existiert nur virtuell. Es besteht aus Verzeichnissen und Dateien, die sich nicht auf einer Festplatte befinden. Der Kernel erzeugt deren Inhalte stattdessen dynamisch zur Laufzeit. Der Vorteil dabei: Statt alter Kamellen liefert es neueste Informationen.

Simple Schnittstelle

Das Proc-Filesystem ist nicht nur eines der ersten, sondern auch das bekannteste virtuelle Linux-Dateisystem. Anfänglich dafür gedacht, Informationen zu Rechenprozessen zu liefern, ist es mittlerweile zu einem regelrechten Sammelsurium von Verzeichnissen und Dateien geworden. Das in der letzten Kern-Technik-Folge vorgestellte Sys-Filesystem soll zwar in Zukunft mit dieser Unordnung aufräumen, aber noch ist das ProcFS das Fenster zum Kernel. Das virtuelle Filesystem weist eine Reihe von Vorteilen auf:

- Der Benutzer kann über Dateien die Informationen des Kernels lesen und schreiben.
- Dank Ascii-Ausgabe ist diese Schnittstelle auch ohne spezielle Software in Shellskripten vom Anwender nutzbar.
- Es ist für die Kernelentwickler recht einfach zu handhaben.

Der letzte Aspekt gilt aber nicht uneingeschränkt. Es ist nur dann einfach, das Proc-Filesystems zu verwenden, wenn Userspace und Kernel geringe Datenmengen austauschen. Mit steigender Informationsmenge sowie bei sich ändernden Daten wird es schwieriger, das System in eigenen Kernelmodulen zu verwenden. Muss eine Kernelkomponente nur wenige Daten ausgeben, legt der Entwickler ein virtuelles Verzeichnis einfach dadurch an, dass er eine einzige Funktion aufruft:

```
struct proc_dir_entry *mk_procdir( const char
   *name, struct proc_dir_entry *parent )
```

Analog erzeugt folgende Funktion eine Proc-Datei:

```
struct proc_dir_entry *create_proc_entry(
   char *name, mode_t mode, struct proc_dir_
   entry *parent )
```

Der Parameter »name« stellt den Namen des Verzeichnisses respektive der Datei dar. »parent« kennzeichnet das übergeordnete Verzeichnis, unter dem Verzeichnis oder Datei erscheinen. Ein Null-Pointer an dieser Position steht für das »/proc«-Verzeichnis selbst. Die beiden Funktionen geben einen Zeiger auf eine Struktur »struct proc_dir_entry« zurück, die den neuen Eintrag repräsentiert (siehe Listing 1, Zeilen 24 und 25).

Rechte setzen

Mit dem zusätzlichen Parameter »mode« legt man die Zugriffsrechte für die Proc-Datei fest. Die Bit-Kodierung der Rechte befindet sich in der Headerdatei »linux/stat.h«. Wird die Funktion mit dem Wert »0« oder dem Symbol »S_IRUGO« als Parameter aufgerufen, können alle User die Proc-Datei lesen. Eine wesentliche Komponente fehlt bislang – eine Funktion, die die eigentliche Information aus dem Kernel aufbereitet und dem User übergibt. Da sie vom jeweiligen Einsatzzweck abhängt, muss der Kernelprogrammierer sie selbst schreiben (im Folgenden »ProcRead()«).
Um dem Kernel die Adresse dieser Funktion bekannt zu geben, trägt der Programmierer sie in die Struktur »struct proc_dir_entry« ein, die der Kernel beim Aufruf von »create_proc_entry()« zurückgegeben hat. Diesen Vorgang zeigt Zeile 27 in Listing 1. Wer schreibfaul ist, kann statt »create_proc_entry()« auch die Inline-Funktion »create_proc_read_entry()« aufrufen:

```
create_proc_read_entry( "ProcExample",
   S_IRUGO, ProcDir, ProcRead, NULL );
```

Die Funktion schreibt die Adresse von »ProcRead()« direkt in die entsprechende Datenstruktur und ersetzt damit die Zeilen 26 bis 29 von Listing 1.

Ähnlich wie »DriverRead()« (siehe [1]) ruft der Kernel »ProcRead()« auf, sobald eine Applikation über den Systemaufruf »read()« die zugehörige Proc-Datei liest. Um der Applikation die internen Informationen lesbar – also in Ascii kodiert – zur Verfügung zu stellen, sind die in Abbildung 1 dargestellten Schritte erforderlich: Speicher reservieren, Daten aufbereiten, Daten kopieren und den Speicher wieder freigeben, sobald die Applikation bedient ist.

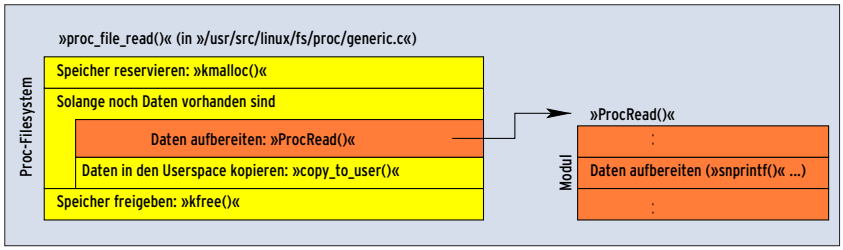


Abbildung 1: Um ein Proc-Filesystem einzurichten, muss der Kernelprogrammierer nur eine »ProcRead()«-Funktion schreiben. Diese wird vom Kernel aufgerufen und schreibt die angefragten Daten in einen Puffer.

Der Kernel hilft

Das Reservieren des Speichers sowie die Kopieraktion zum Userspace und das spätere Freigeben des Speichers übernimmt das Proc-Subsystem automatisch. Dem Programmierer bleibt einzig die Aufgabe, die Daten in dem allozierten Speicher aufzubereiten. Aus diesem Grund besitzen die Funktionen »DriverRead()« und »ProcRead()« – obgleich funktional sehr verwandt – unterschiedliche Aufrufparameter. Ein Funktionskopf sieht folgendermaßen aus:

```
static int ProcRead( char *mempage,
   char **start, off_t offset,
   int size, int *eof, void *data )
```

Der Parameter »mempage« bezeichnet die Adresse der Speicherseite, die das Proc-Subsystem bereits reserviert hat, während »size« die Größe dieses Speicherbereichs nennt. Die Variable »offset« gibt an, ab welcher Stelle in der virtuellen Datei der Userprozess lesen will (Abbildung 5). Die beiden übrigen Parameter »start« und »eof« sind Zeiger auf Variablen, die Rückgabewerte für das Proc-Subsystem aufnehmen.

Über »eof« teilt »ProcRead()« mit, dass keine weiteren Daten zur Ausgabe vorhanden sind. »start« nimmt üblicherweise die Adresse innerhalb der übergebenen Speicherseite auf, ab der »Proc Read()« die Daten abgelegt hat. Falls die auszugebenden Daten am Beginn der Speicherseite stehen, trägt »ProcRead()« hier »NULL« ein. Dann wertet der Kernel den Parameter »offset« aus und findet selbst die Daten.

Eine »ProcRead()«-Funktion ist also leicht zu programmieren, wenn alle Daten in die Speicherseite passen. Die Daten selbst kopiert man am einfachsten mit Hilfe der Funktionen »snprintf()« und »strncat()« in die Speicherseite. Die Zeilen 15 und 16 in Listing 1 zeigen, wie man den Rückgabewert dieser Funktionen nutzt, um sicherzustellen, dass »ProcRead()« nicht mehr Bytes schreibt, als Speicher verfügbar ist. Außerdem gibt die Funktion die Gesamtlänge der geschriebenen Daten zurück.

Vorsicht: Buffer Overflow

Theoretisch stehen auch die Funktionen »sprintf()« und »strcat()« bereit. Man sollte sie aber auch im Kernel vermeiden, denn sie könnten unter Umständen mehr Bytes kopieren, als überhaupt in der Speicherseite Platz haben. Etwas komplizierter wird es, wenn die gut 3000 Bytes einer Speicherseite nicht ausreichen, um alle Daten aufzunehmen. Dann nämlich ruft der Kernel die Funktion »ProcRead()« mehrfach auf. Die wiederum erkennt anhand des Parameters »offset«, welche Daten sie als nächste in der Speicherseite ablegen muss (siehe Abbildung 2).

In diesem Fall kodiert »ProcRead()« einen Datensatz nach Ascii und überprüft, ob dabei der Offset im Ausgabestrom erreicht ist. Wenn nicht, verwirft die Funktion den ersten Datensatz und bereitet den darauf folgenden auf. Diesen Vorgang wiederholt sie so lange, bis der Offset schließlich überschritten ist. Nun muss die Funktion nur noch den Wert von »start« auf den Beginn der aktuellen Ausgabe setzen.

Listing 2 zeigt eine »ProcRead()«-Funktion, die immer wieder den String »Hallo Welt\n« zurückgibt. Wenn etwa 4000 Zeichen ausgegeben sind, setzt »Proc Read()« den Parameter »eof« auf »1« und liefert den Wert »NULL« zurück. Reale

Listing 1: Implementierung einer Proc-Datei

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/proc_fs.h>
04 #include <linux/init.h>
05 #include <linux/stat.h>
06
07 MODULE_LICENSE("GPL");
08
09 static struct proc_dir_entry *ProcDir,
   *ProcFile;
10
11 static int ProcRead( char *buf, char **start,
   off_t offset, int size, int *eof, void *data)
12 {
13    int BytesWritten=0;
14
15    BytesWritten += snprintf( buf, size,
   "ProcRead wurde\n" );
16    BytesWritten += snprintf( buf+BytesWritten,
17         size-BytesWritten, "aufgerufen.\n" );
18    *eof = 1;
19    return BytesWritten;
20 }
21
22 static int __init ProcInit(void)
23 {
24    ProcDir = proc_mkdir( "ExampleDir", NULL );
25    ProcFile = create_proc_entry( "ProcExample",
   S_IRUGO, ProcDir );
26    if ( ProcFile ) {
27         ProcFile->read_proc = ProcRead;
28         ProcFile->data = NULL;
29    }
30    return 0;
31 }
32
33 static void __exit ProcExit(void)
34 {
35    if ( ProcFile ) remove_proc_entry(
   "ProcExample", ProcDir );
36    if ( ProcDir )  remove_proc_entry(
   "ExampleDir", NULL );
37 }
38
39 module_init( ProcInit );
40 module_exit( ProcExit );
```

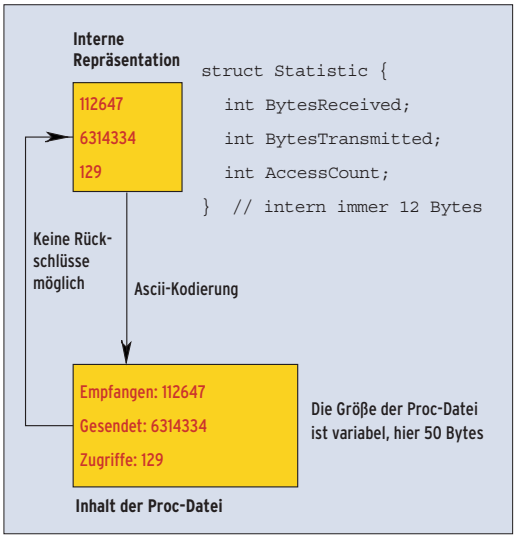


Abbildung 2: Die Größe der Proc-Datei unterscheidet sich vom internen Speicherbedarf der Daten. Auch wenn die Position in den Ausgabedaten (einem Offset entsprechend) bekannt ist, lässt sie keine direkten Rückschlüsse auf den internen Speicherort zu.

»ProcRead()«

»offset« beim Aufruf	Returnwert[1]	»*start« am Ende	neuer Inhalt von »offset«	
0 14	14 23	0 0	14 37	»offset« zählt ausgegebene Ascii-Zeichen
0 1	111 89	1 1	1 2	»offset« zählt in Einheiten, die »ProcRead()« selbst festleg

[1] der Returnwert ist gleich der Anzahl der kopierten Bytes

Abbildung 3: Der Inhalt des Aufrufarguments »start« entscheidet, ob »ProcRead()« den Returnwert oder den Inhalt von »start« selbst zur Berechnung des neuen Offsets verwendet.

Aufgabenstellungen verursachen mehr Kopfzerbrechen. Dass der Ausgabedatenstrom im Regelfall umfangreicher ist als die Repräsentation der internen Daten, stellt an sich noch keine wirkliche Schwierigkeit dar.

Zeitvariante Daten

Dass sich die internen Daten zwischen zwei oder mehr notwendigen Zugriffen auf die »ProcRead()«-Funktion verändern können – also dynamisch sind –, ist problematisch. Soll eine Proc-Datei einen Integerwert ausgeben, der bei einem x86-Kernel grundsätzlich 4 Bytes belegt, benötigt sie für dessen dezimale Ausgabe zwischen 1 (Wert »0«) und 10 Bytes (Wert »4294967296«). Ändert sich der Wert der zugehörigen Variablen zwischen den beiden Aufrufen von »Proc Read()«, kommt es zu einer Verschiebung im Ausgabestrom, also zu einem fehlerhaften Ergebnis. Regel Nummer eins lautet daher, solche Werte nur „ganz oder gar nicht“ auszugeben.

Bei dynamischen Daten – also Daten, die sich mit der Zeit ändern – ist es unmöglich, Rückschlüsse auf den internen Speicherort zu ziehen. Der Parameter »offset« spiegelt nur die bisherige Position im Ausgabedatenstrom wider. Trotzdem muss »ProcRead()« bei wiederholtem Aufruf wissen, wo es beim letzten Mal aufgehört hat zu lesen.

Die Lösung des Problems besteht darin, die aktuelle, interne Position zwischenzuspeichern. Dazu haben die Kernelentwickler auf einen Trick zurückgegriffen. »ProcRead()« darf nämlich in der Variablen »start« auch einen Wert ablegen, der kleiner ist als die Anfangsadresse der Speicherseite. In diesem Fall addiert »ProcRead()« den in »start« abgelegten Wert zum Offset, nicht wie sonst üblich die Anzahl der kopierten Zeichen (siehe Abbildung 3).

In welcher Einheit die interne Position abgelegt wird, bleibt dem Kernelhacker selbst überlassen. Sinnvoll ist es beispielsweise, die Anzahl der bisher ausgegebenen Datensätze mitzuzählen. Wenn also »ProcRead()« beim ersten Aufruf einen Datensatz aufbereitet und diese Daten 10 Bytes belegen, liefert die Funktion zwar – wie bisher auch – »10« zurück. Sie setzt allerdings »start« auf »1«. Beim nachfolgenden Aufruf von »ProcRead()« enthält »offset« nicht den Wert »10«, sondern »1«. »ProcRead()« muss nun den Datensatz mit der Nummer »1« ausgeben, also den zweiten Datensatz. Mit diesem Marker findet die Funktion die Position innerhalb der internen Daten, hier den Index. Listing 3 zeigt, wie man »start« nutzt, um damit die interne Position festzuhalten.

Listing 2: Komplexere Lesefunktion

```
01 static int ProcRead( char *buf, char **start,
   off_t offset,
02                  int size, int *eof, void *data
   )
03 {
04    int ActualWritten=0, BytesWritten=0,
   NewByteCount;
05
06    while( BytesWritten < (long) offset ) { //
   Aktuellen Datensatz suchen.
07       ActualWritten = snprintf( buf, size,
   "Hallo Welt\n" );
08       BytesWritten += ActualWritten;
09    }
10    // In buf befinden sich jetzt alte _und_ neue
   Daten.
11    // Start wird auf die neuen Daten gesetzt.
12    NewByteCount=BytesWritten - offset;
13    *start = buf + ActualWritten -
   NewByteCount;
14    size -= ActualWritten - NewByteCount; // Auf
   neue Startadresse anpassen.
15
16    NewByteCount += snprintf(
   *start+NewByteCount,
17       size - NewByteCount, "Hallo Welt\n");
18    if( offset > 4000 ) {
19       *eof = 1;
20       return 0;
21    }
22    return NewByteCount;
23 }
```

Schon aufgeräumt?

Entfernt der Benutzer das Kernelmodul, sollen die zugehörigen Einträge im Proc-Filesystem natürlich ebenfalls verschwinden. Dazu steht dem Programmierer die Funktion »void remove_proc_entry(const char *name, struct proc_dir_entry *parent)« zur Verfügung.

Eine Proc-Datei kann nicht nur lesen, sondern auch schreiben – ein überaus nützliches Feature. So konfiguriert der Benutzer zur Laufzeit Kernelkomponenten oder Treiber, indem er einfach den entsprechenden Wert in die Proc-Datei schreibt:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Auch für den schreibenden Zugriff steht eine Funktion zur Verfügung, im Folgenden als »ProcWrite()« bezeichnet. Die Adresse dieser Funktion trägt der Kernel-

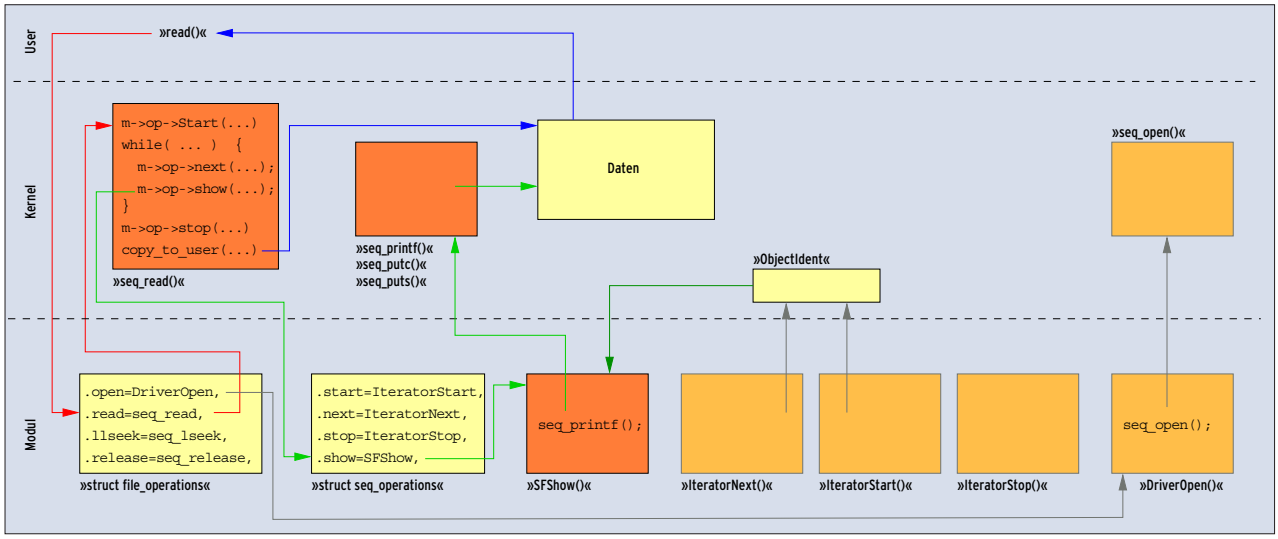


Abbildung 4: Um ein Sequence File zu unterstützen, muss der Programmierer die dargestellten Komponenten realisieren: Drei Iteratoren-Funktionen sowie »SFShow()« benötigt der Kernel, um ein Sequence File anzusprechen. Die Lese- und Schreibfunktionen gleichen dann denen der Standard-C-Bibliothek.

Programmierer in die nach dem Anlegen einer Proc-Datei zurückgegebene Struktur »struct proc_dir_entry« ein. Wichtig ist es dabei, beim Anlegen der Datei die Rechte für den Schreibzugriff (»S_IWUGO«) zu setzen.

Der Prototyp der Funktion (deklariert in »linux/proc_fs.h«) zeigt, dass die Funktion nicht über eine Speicherseite mit der Anwendung kommuniziert, sondern direkt:

```
int ProcWrite( struct file *file, const
   char *buffer, unsigned long count,
   void *data )
```

Einzig der Parameter »data« statt eines Offsets unterscheidet die Funktion von der bekannten »DriverWrite()« (siehe [1]). »ProcWrite()« transferiert die Applikationsdaten selbst aus dem Userspace in den Kernelspace, indem es »copy_from_user()« aufruft. Die meist in Ascii gehaltenen Daten dekodiert man am besten mit Hilfe der Funktionen »strncmp()« oder »strstr()«. Listing 4 zeigt die Implementierung einer Funktion »ProcWrite()«.

Sequence Files

Interne, zeitvariante Informationen Ascii-kodiert ausgeben ist ein Problem aller virtuellen Dateien. Kernel 2.6 bietet mit den so genannten Sequence Files [2] (»linux/seq_file.h«) einen alternativen Lösungsansatz. Ist ein Sequence File (SF) erst mal angelegt, lässt es sich einfach verwenden. Der Entwickler hat im Wesentlichen nur eine Funktion zu schreiben, die einen Datensatz in die Ascii-kodierte Form konvertiert (im Folgenden »SFShow()«):

```
static int SFShow(struct seq_file
   *SFObject, void *ObjectIdent)
```

Der Kernel übergibt dieser Funktion zwei Parameter: erstens die Adresse eines SF-Objekts (»SFObject«), über das der Kernel das Sequence File verwaltet, und zweitens »ObjectIndent«, über den er das auszugebende Element identifiziert. Wie »ObjectIndent« zu interpretieren ist, ist Sache des Programmierers von »SFShow()«. Im Regelfall wird man wohl einen Zeiger auf das auszugebende Element ablegen.

Zur Konvertierung der internen Daten in eine Ascii-kodierte Darstellung bieten sich die folgenden drei Funktionen an: »seq_printf()«, »seq_putc()« und »seq_puts()«. Sie entsprechen den aus dem Applikationsbereich bekannten Funktionen »printf()«, »putc()« und »puts()«. Der einzige Unterschied besteht darin, dass das erste Argument der »seq_«-

Listing 3: Zustandsinformationen in »start«

```
01 static int Beispieldaten[ANZAHL_ELEMENTE];
02 ...
03 static int ProcRead( char *buf, char **start,
   off_t offset,
04         int size, int *eof, void *data )
05 {
06    int count;
07
08    if( offset >= ANZAHL_ELEMENTE ) {
09       *eof = 1;
10       return 0;
11    }
12    count = snprintf( buf, size, "Beispieldaten
   %ld: %d\n",
13       offset, Beispieldaten[offset] );
14    *start = 1; // Ein weiteres Element
   verarbeitet.
15    return count;
16 }
```

Listing 4: Proc-Datei schreiben

```
01 static int ProcWrite( struct file *Instanz,
   const char __user *UserBuffer,
02                 unsigned long count, void
   *data )
03 {
04    char *KernelBuffer;
05    int NotCopied;
06
07    KernelBuffer = kmalloc( count, GFP_KERNEL );
08    if( !KernelBuffer ) {
09       return -ENOMEM;
10    }
11    NotCopied = copy_from_user( KernelBuffer,
   UserBuffer, count );
12    if( strncmp( "Konfiguration", KernelBuffer,
   14 - 1 )==0 )
13       configured = 1;
14    kfree( KernelBuffer );
15    return count - NotCopied;
16 }
```

Funktionen ein Zeiger auf das Sequence-File-Objekt ist. Um die im Speicher liegenden Einzelteile des Sequence File nacheinander zu lesen, benötigt der Kernel außer »SFShow()« weitere geeignete Funktionen, die hier »IteratorStart()«, »IteratorNext()« und »IteratorStop()« heißen. Will eine Anwendung Daten lesen, ruft der Kernel zunächst »IteratorStart()« auf und dann mehrfach »IteratorNext()«. »SFShow()« benutzt die Rückgabewerte dieser Funktionen.

Hat der Kernel genügend Daten gesammelt, ruft er »IteratorStop()« auf und übergibt sie der Anwendung. Sobald diese weitere Daten anfordert, starten zunächst »IteratorStart()«, dann wiederholt »IteratorNext()« und schließlich »IteratorStop()« erneut, und zwar so lange, wie noch Daten vorhanden sind:

```
void *IteratorStart( struct seq_file *m,
   loff_t *Index );
void *IteratorNext( struct seq_file *m,
   void *ObjectIdent, loff_t *Index );
void *IteratorStop( struct seq_file *m,
   void *ObjectIdent );
```

»IteratorStart()« erhält als Argument einen Index, der die Anzahl der bisher bearbeiteten Datensätze angibt. Sind die auszugebenden Daten beispielsweise in einer Liste organisiert, wird »IteratorStart()« genau »Index« Listenelemente verwerfen, um das dann folgende Listenelement der aufrufenden Funktion zurückzugeben. »IteratorNext()« benutzt sowohl den Index als auch die Objekt-Identifikation »ObjectIdent«. Diese Funktion gibt ebenfalls das nächste Element zurück. Sind alle Elemente ausgegeben, ist der Rückgabewert »NULL«. Für jedes zurückgegebenen Element inkrementieren die Funktionen darüber hinaus die Variable »*Index«.

Sobald der Kernel genug Elemente gelesen hat, ruft er die Funktion »IteratorStop()« auf. Sie macht die beim Aufruf von »IteratorStart()« durchgeführten Initialisierungen wieder rückgängig. Mit dem Funktionspaar »IteratorStart()« und »IteratorStop()« lassen sich außerdem konkurrierende Zugriffe auf die auszugebenden Daten über einen Semaphor vermeiden.

Funktionen registrieren

Um die Adressen der Funktionen »IteratorStart()«, »IteratorNext()« und »IteratorStop()« dem Kernel bekannt zu geben, trägt der Entwickler sie in eine Instanz der Datenstruktur »struct seq_operations« ein. Die Funktion »seq_open()« wird typischerweise von »DriverOpen()« oder »ProcOpen()« aufgerufen. Statt der im »struct file_operations« vorkommenden Funktionen »DriverRead()«, »DriverLSeek()« und »DriverRelease()« kommen die durch den Kernel bereitgestellten Funktionen »seq_read()«, »seq_lseek()« und »seq_release()« zum Einsatz. Das Sequence File ist damit fertig implementiert. Abbildung 4 zeigt die Modulkomponenten eines Sequence File sowie den Ablauf, wenn eine Applikation das Sequence File liest.

Das komplette Beispiel eines Sequence File (»seqfile.c«) liegt auf dem FTP-Server des Linux-Magazins bereit [4]. Diese Implementierung organisiert die internen Daten in einer verketteten Liste. Liest eine Anwendung »/proc/SeqTestFile«, gibt der Kernel in diesem Beispiel sämtliche Elternprozesse des aufrufenden Prozesses mit ihrer PID aus.

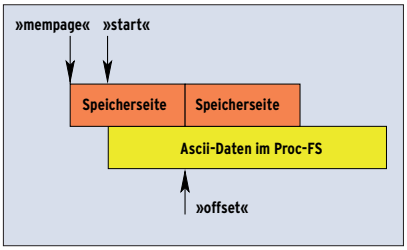


Abbildung 5: »mempage« und »start« verweisen auf die im Speicher liegenden Daten. »offset« bezieht sich dagegen auf die Ausgabe über das Proc-File.

Vorschau

Reale Filesysteme liegen im Gegensatz zu den virtuellen meist auf Festplatten, benutzen also echte Hardware. Der Kernel braucht dafür passende Treiber, um Daten blockweise und im wahlfreien Zugriff zu lesen und zu schreiben. Block-Gerätetreiber implementieren diese Funktionen, sie sind das Thema der nächsten Kern-Technik. *(ofr/fjl)* ■

Single Files – Abwandlung der Sequence Files

Es gibt virtuelle Dateien, denen genau ein Datensatz im Speicher entspricht. Für diesen Fall steht das »Single File«-Interface zur Verfügung. Der Kernel übernimmt das gesamte Handling, insbesondere auch das Seek innerhalb der Daten. Der Entwickler hat letztlich nur eine »Show()«-Funktion zu schreiben. Die »Open()«-Funktion des Moduls (zum Beispiel »DriverOpen()« oder »ProcOpen()«) ruft die Funktion »single_open()« auf. Die Funktion »single_open()« bekommt neben dem Zeiger auf ein »struct file« noch die Adresse der Funktion »SFShow()« übergeben. Da »single_open()« dynamisch Speicher alloziert, kommt statt »DriverRelease()« eine Funktion namens »single_release()« zum Einsatz. Die Lese- und die Seek-Funktionen sind wiederum »seq_read()« und »seq_lseek()«.

Proc-Filesystem aus Benutzersicht

Auch für Anwender bringt der Kernel 2.6 einige Neuerungen mit. Das Format vieler Dateien hat sich geändert und so muss der User die entsprechenden Procps-Programme installieren, um Auskunft über laufende Prozesse oder die Speichernutzung zu bekommen [3]. Wenn, wie schon mehrfach erwähnt, das Sys- viele Aufgaben des Proc-Filesystems übernimmt, ändern sich weitere Files, was für Inkompatibilitäten sorgt. So müssen viele Systemtools angepasst werden, die Kernelinformationen über das Proc-FS lesen und schreiben.

Threads unterstützt das neue ProcFS effizienter, es zeigt erst einmal nur den Original-Thread an und spart damit Suchzeit. Der Kernel schreibt nun über »/proc/config.gz« die Compile- und Modulkonfiguration heraus. Die Benutzer von Multiprozessormaschinen stellen die CPU-Affinität für einzelne Interrupts ein, indem sie eine Bitmaske in »/proc/irq/IRQ#/smp_affinity« schreiben (siehe »Documentation/IRQ-affinity.txt«).

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 9/03, S. 86
[2] Jonathan Corbet, „Driver Porting: The seq_file interface“: [http://lwn.net/Articles/22355/]
[3] Procps: [http://procps.sourceforge.net/]
[4] Listings: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/02/Kern-Technik/]

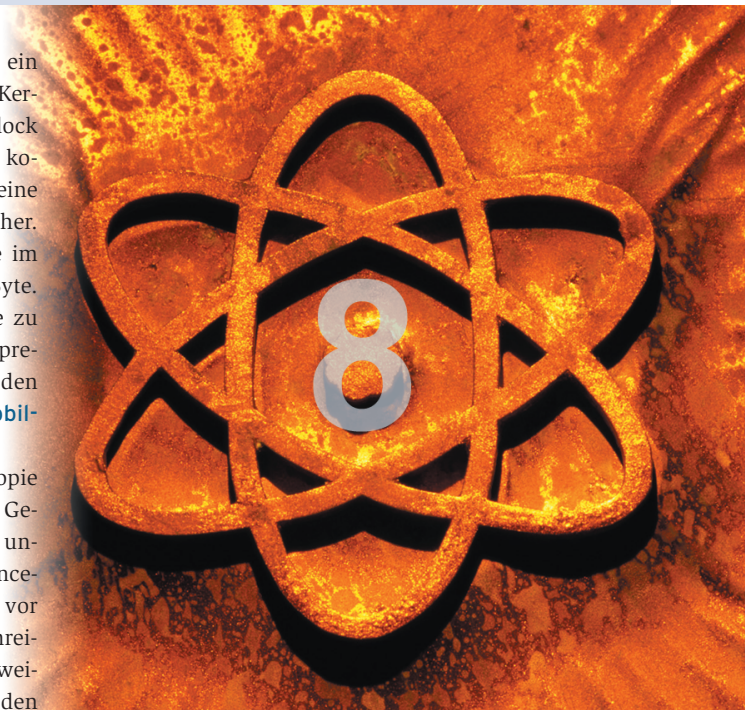
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 8

Kern-Technik

Die Geräteklasse der Blockdevices bildet die Grundlage für Filesysteme. Diese Kern-Technik-Folge erklärt die Treiberkomponenten und zeigt die Implementierung einer RAM-Disk als Beispiel für das neue Blockgeräte-Interface in Kernel 2.6. Eva-Katharina Kunst, Jürgen Quade



Der Name klingt zwar sperrig, doch Blockgeräte haben mit großen Kisten nichts zu tun. Festplatten, DVDs, CD-ROMs, USB-Sticks oder Flashcards verarbeiten Daten blockweise und verlangen deshalb nach einem Blockgeräte-Interface im Kernel. Meist dienen sie dazu, Daten dauerhaft zu speichern, also als Hintergrundspeicher, dessen Verkörperung eben eine Festplatte ist. Programme können mit Funktionen wie »fseek()« oder »lseek()« an beliebige Stellen auf dem Blockgerät springen und dort lesen oder schreiben (Random Access).

Am Ende anfangen

Dabei lässt sich das letzte Byte also durchaus vor dem ersten lesen. Vergleicht man den Transport von Daten mit fließendem Wasser, handelt es sich bei einem Blockgerät nicht um einen Datenstrom, sondern eher um ein stehendes Gewässer. Wer hier Wasser schöpfen will, kann dazu nur einen Eimer benutzen – selbst dann, wenn er lediglich einen einzigen Wassertropfen benötigt. Beim Kernel 2.6 fasst ein solcher Eimer genau 512 Bytes. Wer von einem Blockgerät ein einzelnes Byte lesen möchte, muss also stets 512 Bytes entnehmen. Die restlichen 511 Bytes lässt er dann einfach liegen.

Da von einem Blockgerät ohnehin keine einzelnen Bytes gelesen werden können, verwaltet der Kernel den Hintergrundspeicher über Blocknummern, unabhängig davon, dass auf Hardware-Ebene Köpfe, Zylinder und Sektoren dahinter stecken. Für die ersten 512 Bytes eines Hintergrundspeichers ist Blocknummer 0 maßgeblich, Bytes 512 bis 1023 entsprechen Blocknummer 1 und so weiter.

Ist von einem Blockgerät ein Byte zu lesen, berechnet der Kernel zunächst, in welchem Block sich das Byte befindet, und kopiert diesen Block über seine Nummer in den Hauptspeicher. Dann liest er aus der Kopie im Speicher das gewünschte Byte. Lautet der Auftrag, ein Byte zu schreiben, liest er den entsprechende Block ebenfalls in den Hauptspeicher ein (siehe Abbildung 1).

Nach der Änderung der Kopie schreibt er den Block auf das Gerät zurück. Allerdings nicht unbedingt sofort: Aus Performancegründen wartet das System vor dem physischen Zurückschreiben erst einmal ab, ob noch weitere Schreibaufträge für den Block eintreffen.

Namen statt Zahlen

Damit der Anwender Daten auf dem Blockgerät ablegen und wiederfinden kann, verwaltet der Kernel die Blöcke eines solchen Geräts über ein Dateisystem, beispielsweise Ext 3 (siehe auch Titelthema-Artikel). Durch selbst vergebene Namen für Verzeichnisse und Dateien ist der Zugriff für den Anwender einfacher, als wenn er Blocknummern verwenden müsste.

Auch dem Treiberprogrammierer bringt die Abstraktion Vorteile. Er muss sich nicht darum kümmern, was eine Anwendung auf der Ebene des Dateisystems macht. Sein Treiber bekommt stattdessen vom Betriebssystem so simple Aufträge wie „Kopiere den Inhalt von Block xyz in die Mempage abc“ oder „Schreibe den Inhalt von Mempage abc in Block xyz“.

Um vernünftige Aufträge zu erteilen, braucht der Kernel einige Informationen über den Hintergrundspeicher, etwa dessen Größe angegeben als Anzahl der Blöcke. Außerdem muss er die Major- und Minor-Nummer kennen, um mit ihrer Hilfe über die Gerätedatei die Zuordnung zwischen Applikation respektive Dateisystem und Treiber herzustellen (siehe Kasten „Gerätenummern“).

Datenstrukturen

Alle Informationen stecken in der Datenstruktur »struct gendisk« (definiert im Header »linux/genhd.h«). Diese Struktur ist für den Blockgerätetreiber von zentraler Bedeutung, denn sie spezifiziert sowohl das Gerät – die Disk – als auch

die Gerätezugriffe über den Treiber. Neben der Sektorgröße und der Major- und Minor-Nummer gehören auch der Gerätename und der auf dem Gerät vorhandene Speicherplatz zu den Elementen, die der Treiberentwickler angibt.

Treiberkomponenten

Die Routine »struct gendisk« ist nur eine von drei elementaren Komponenten eines Blockgerätetreibers. Die zweite ist eine Datenstruktur, die der Kommunikation zwischen Treiber und Blockgeräte-Subsystem dient: die Request-Queue (siehe Abbildung 2). In einer solchen Queue sammelt der Kernel alle Aufträge an den Treiber samt dem Zeiger auf die Daten. Mehr noch, der Kernel sortiert in einer so genannten Make-Request-Funktion die Aufträge (Requests) der »request_queue« nach der Sektornummer. So arbeitet ein Treiber physisch nahe zusammenliegende Sektoren nacheinander ab und spart damit Zeit.

Die dritte Komponente ist schließlich die Funktion, die die Befehle „Kopiere den Inhalt von Block xyz in die Mempage abc“ oder „Schreibe den Inhalt von Mempage abc in Block xyz“ ausführt, also den eigentlichen Datentransfer abwickelt. Sie heißt Request-Funktion und ist vom Typ »request_fn_proc«. Im einfachsten Fall arbeitet Request-Funktion die Aufträge der Request-Queue nacheinander ab. Die verschiedenen Auftragstypen sind in der Headerdatei »linux/blkdev.h« zu finden.

Die Requests selbst sind einen genaueren Blick wert: Ein einziger Auftrag überträgt zugleich mehrere Blöcke in den Hauptspeicher. Dann wird es jedoch komplizierter, denn für mehrere Blöcke im Hintergrundspeicher müssen auch mehrere Blöcke im Hauptspeicher vorhanden sein. Dort liegen sie aber, im Unterschied zum Hintergrundspeicher, nur selten sequenziell hintereinander.

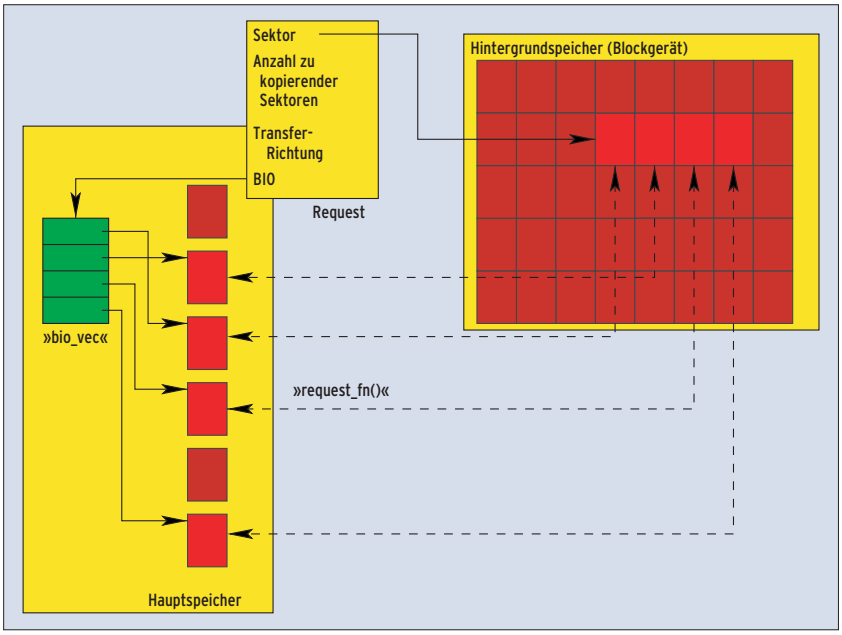


Abbildung 1: Der Kernel verwendet für den Zugriff auf Daten eines Blockgeräts Kopien der Blöcke im Hauptspeicher. Dafür stellt ein Treiber die Funktion »request_fn()« bereit.

Alles BIO

Zur Spezifikation der Blöcke im Hauptspeicher werden daher mit jedem Request eine oder mehrere so genannte BIO-Strukturen (Block I/O) verknüpft, siehe Abbildungen 2 und 3. Jede BIO-Struktur besitzt ein Feld von Zeigern auf Speicherseiten namens »bio_vec«. Liegen die Speicherseiten aber im Userspace oder im High-Memory-Bereich, sind sie nicht direkt adressierbar. Eine Adressierung über die Page ermöglicht in diesen Fällen aber einen Transfer per DMA.

Da bei einem virtuellen Gerät die Optimierung einzelner Requests unnötig ist, kann man die systemeigene Make-Request-Funktion »make_request()« durch eine eigene Funktion ersetzen. Sie wickelt den Datentransfer direkt ab und macht eine (normale) Request-Funktion unnötig. Damit entfällt auch das Iterieren über die einzelnen Requests: Die ersetzte Make-Request-Funktion bekommt die Aufträge – sobald sie anfallen – in einem BIO-Block übergeben.

Ran an die Tastatur

So viel zur Theorie, jetzt zur Praxis. Wenn der Kern eines Blockgerätetreibers nur aus dem Transfer von Blöcken be-

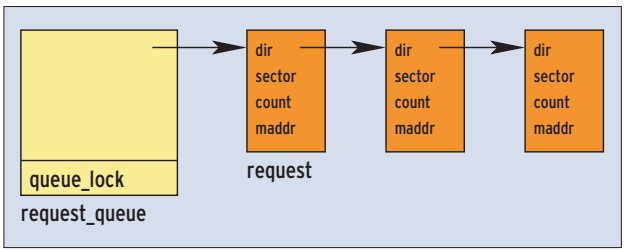


Abbildung 2: Request-Datenstruktur: Eine Request-Queue enthält die unterschiedlichen Aufträge in Form einer verketteten Liste. Der Zugriff auf einzelne Listenelemente erfolgt über das Makro »elv_next_req()«.

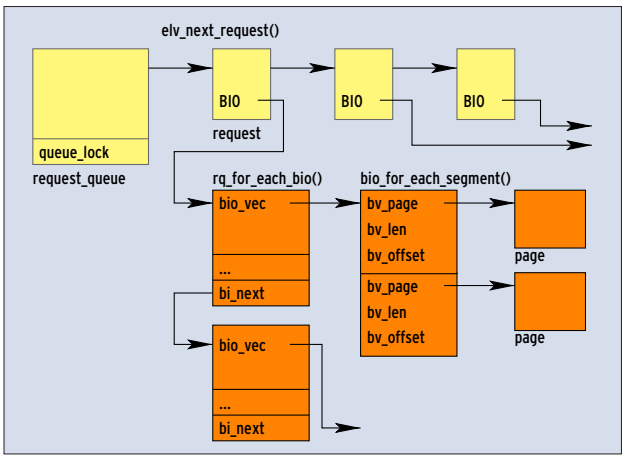


Abbildung 3: Die BIO-Datenstruktur enthält Felder von Zeigern »bio_vec«, die letztlich auf Seiten im Hauptspeicher verweisen.

steht, kann es nicht so schwer sein, einen Treiber für eine virtuelle Festplatte (RAM-Disk) zu schreiben.

Der Speicher der virtuellen Disk soll gleich beim Laden des Treibers reserviert werden. Die Funktion »kmalloc()« eignet sich dafür nicht, weil sie nur maximal 64 KByte zusammenhängenden Speicher reserviert. Stattdessen kommt »vmalloc()« zum Einsatz, das hier 8 MByte alloziert, was für die RAM-Disk genügt. Die spätere Freigabe des Speichers erfolgt mit »vfree()«.

Zugriffsrechte vereinfacht

Die Funktion »alloc_disk()« (Listing 1, Zeile 52) reserviert Speicher für die »struct gendisk«. In diese trägt der Treiber die Kapazität der virtuellen Festplatte und die Tabelle mit den Funktionszeigern ein. Da Zugriffsrechte in diesem Beispiel keine besondere Rolle spielen, reicht es bereits aus, in der Tabelle lediglich das Feld ».owner« zu belegen. Dieses Feld muss der Treiber allerdings in jedem Fall ausfüllen, damit der Kernel das Modul erst dann zum Entladen freigibt, wenn keine Instanz mehr darauf zugreift.

Bevor der Aufruf von »add_disk()« (Zeile 63) das Objekt dem Blockgeräte-Subsystem übergibt, muss die Request-Queue erzeugt werden. Der Treiber für ein virtuelles Gerät reserviert den Speicher für die Request-Queue durch Aufruf von »blk_alloc_queue()« (Zeile 50) und überschreibt die vorgegebene Make-Request-Funktion durch die eigene Variante »bdMakeRequest()«. Dazu ruft der Treiber »blk_queue_make_request()« auf (Zeile 51). Eine gesonderte Request-Funktion ist in diesem Fall nicht nötig. Der noch in Kernel 2.4 notwendige Aufruf der Funktion »register_blkdev()« ist in Kernel 2.6 optional (siehe hierzu Kasten „Portierungshilfe“).

Die Funktion »bdMakeRequest()« bekommt die Request-Queue »q« und den Zeiger auf den BIO-Block des aktuellen Requests »bio« übergeben. Den Parameter »q« verwendet die Funktion nicht, denn der Treiber verarbeitet Anfragen sofort, ohne sie in eine Schlange einzureihen. Bevor der Treiber »bio« auswertet, muss er sicherstellen, dass er auf den Datenbereich des BIO-Blocks zugreifen darf – mit der Funktion »blk_queue_bounce()« (Zeile 25) kein Problem. Um jetzt die Daten zwischen virtueller Festplatte und dem BIO-Block per »memcpy()« zu transferieren, muss der Treiber noch die Quell- und Zieladressen sowie die Anzahl der zu kopierenden Bytes bestimmen.

Adressen berechnen

Über die Startadresse der RAM-Disk und den durch die Sektornummer gegebenen Offset lässt sich die Adresse »maddr« auf der virtuellen Festplatte berechnen. Um an die entsprechenden Adressen im Hauptspeicher zu kommen, durchsucht der Treiber die Tabelle »bio_vec«, in der die Adressangaben des BIO-Blocks stehen. Hierbei hilft das Makro »bio_for_each_segment()«.

Die Makros »bio_data()« und »bio_data_dir()« lesen schließlich die Adresse des zugehörigen Datenbereichs und die

Listing 1: Treiber einer virtuelle Festplatte

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/blkdev.h>
06
07 #define BD_MAJOR 242
08 #define SIZE_IN_KBYTES (8*1024)
09
10 MODULE_LICENSE("GPL");
11
12 static struct gendisk *disk;
13 static struct request_queue *bdqueue;
14 static char *DiscSpace;
15 static struct block_device_operations bdops = {
16     .owner          = THIS_MODULE,
17 };
18
19 static int bdMakeRequest( request_queue_t *q,
   struct bio *bio )
20 {
21    char *kaddr, *maddr;
22    struct bio_vec *bvec;
23    int segnr, BytesTransferred=0;
24
25    blk_queue_bounce( q, &bio );
26    bio_for_each_segment( bvec, bio, segnr ) {
27       kaddr = bio_data(bio);
28       maddr = DiscSpace + (512 * bio-
   >bi_sector);
29       BytesTransferred += bio->bi_size;
30       if( bio_data_dir( bio )==READ ||
   bio_data_dir( bio )==READA ) {
31          memcpy( kaddr, maddr, bio->bi_size );
32       } else {
33          memcpy( maddr, kaddr, bio->bi_size );
34       }
35    }
36    bio_endio( bio, BytesTransferred, 0 );
37    return 0;
38 }
39
40 static int __init ModInit(void)
41 {
42    if( register_blkdev(BD_MAJOR, "bdsample" ))
   {
43       printk("blockdevice: Majornummer %d not
   free.", BD_MAJOR);
44       return -EIO;
45    }
46    if(
   !(DiscSpace=vmalloc(SIZE_IN_KBYTES*1024)) ) {
47       printk("vmalloc failed ...\n");
48       goto out_no_mem;
49    }
50    bdqueue = blk_alloc_queue( GFP_KERNEL );
51    blk_queue_make_request( bdqueue,
   bdMakeRequest );
52    disk = alloc_disk(1);
53    if( !disk ) {
54       printk("alloc_disk failed ...\n");
55       goto out;
56    }
57    disk->major = BD_MAJOR;
58    disk->first_minor = 0;
59    disk->fops = &bdops;
60    disk->queue = bdqueue;
61    sprintf(disk->disk_name, "bd0");
62    set_capacity( disk, (SIZE_IN_KBYTES*1024)>>9
   ); // in 512 Byte Bloecke
63    add_disk( disk );
64    return 0;
65 out:
66    vfree( DiscSpace );
67 out_no_mem:
68    unregister_blkdev(BD_MAJOR, "bdsample" );
69    return -EIO;
70 }
71
72 static void __exit ModExit(void)
73 {
74    unregister_blkdev(BD_MAJOR, "bdsample" );
75    del_gendisk(disk);
76    put_disk( disk );
77    blk_cleanup_queue( bdqueue );
78    vfree( DiscSpace );
79 }
80
81 module_init( ModInit );
82 module_exit( ModExit );
```

Transferrichtung aus. Sie beziehen sich immer auf das aktuelle Segment, also auf den Feldeintrag in »biovec«. Dass der Treiber den Auftrag beendet hat, signalisiert er mit »bio_endio()«. Diese Funktion hat als Parameter einen Zeiger auf den bearbeiteten BIO-Block, die Anzahl der transferierten Bytes und einen möglichen Fehlercode. »BdMakeRequest()« liefert »0« zurück, als Zeichen dafür, dass der Transfer erfolgreich war.

Nach getaner Arbeit muss der Blockgerätetreiber aufräumen. Mit »put_disk()« meldet er die Platte, mit »unregister_blkdev()« sich selbst beim Kernel ab. Dabei gibt er die zu Beginn reservierten Ressourcen für die Platte mit »del_gendisk()« und für die Queue durch »blk_cleanup_queue()« frei. Vor der Freigabe der Queue muss der Treiber »struct gendisk« freigeben.

Mit Hilfe eines passenden Makefile (siehe Listing 2) lässt sich der RAM-Disk-Treiber einfach kompilieren. »insmod« lädt das Modul in den Kernel:

```
insmod virtdisc.ko
```

Um die virtuelle Disk zu testen, ist zunächst eine Gerätedatei anzulegen. Vor dem Mounten muss der Root-User auf der virtuellen Platte ein Filesystem erstellen:

```
mknod virtdiscfile b 242 0
mke2fs virtdiscfile
mount virtdiscfile /mnt
```

Wer versucht das Blockgerät zu partitionieren, wird wenig Erfolg haben. Der Beispieltreiber hat sich nämlich beim Kern nur für ein einziges Gerät registriert. Der Parameter der dafür verwendeten Funktion »alloc_disk()« gibt die Anzahl der unterstützten Geräte an. Er muss also, soll sich das Gerät partitionieren lassen, einen Wert größer als 1 erhalten. Möchten Programme wie »fdisk« die Geometrie der Platte abfragen, ist im Treiber das IO-Control »HDIO_GETGEO()« zu implementieren (siehe [3]).

Listing 2: Makefile für die virtuelle Harddisk

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := virtdisc.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09         $(MAKE) -C $(KDIR)      SUBDIRS=$(PWD) modules
10
11 clean:
12         rm -f *.ko *.o *.mod.c
13 endif
```

Reale Geräte

Ein Blockgerätetreiber für reale Geräte unterscheidet sich von dem vorgestellten Code in zweierlei Hinsicht:

- Ein reales Gerät verwendet die Funktion »blk_init_queue()« , um einer Request-Queue die Adresse der Request-Funktion zu übergeben und sich die Adresse der zugehörigen Request-Queue geben zu lassen. Es gibt kein »blk_alloc_queue()« und kein »blk_queue_make_request()«. Die Funktion erwartet als zweiten Parameter die Adresse eines Spinlocks, der parallele Zugriffe auf die Request-Queue absichert:

```
if( (bdqueue = blk_init_queue(
   &bdRequest, &bdlock))==NULL )
      goto out;
```

- Ein echter Blockgerätetreiber realisiert eine Request-Funktion (siehe Abbildung 4).

Die Request-Funktion entnimmt mit Hilfe des Makros »elv_next_request()« den jeweils nächsten Auftrag aus der Request-Liste:

```
while((req = elv_next_request(q))
   != NULL ) {
```

Das Makro erhält als Parameter die Request-Queue und gibt einen Zeiger auf den nächsten Auftrag »req« zurück (vom Typ »struct request *«, siehe Abbildung 2). Zunächst muss der Treiber aber feststellen, ob es sich um einen normalen Schreib- oder Leseauftrag handelt:

Gerätenummern

Mit Kernel 2.6 hat Linus Torvalds das Ende der klassischen Major- und Minor-Nummern eingeläutet. Anders als geplant sind die neuen Gerätenummern aber nicht 64, sondern nur 32 Bit breit. Abbildung 5 zeigt, wie die alten Major- und Minor-Nummern auf die neuen Gerätenummern abgebildet werden. Das im Artikel vorgestellte Treiberinterface arbeitet weiterhin nach dem alten Schema.

Wer die neuen Gerätenummern nutzen möchte, muss auf zusätzliche Funktionen zurückgreifen. Dazu meldet der Treiber sein Interesse an den benötigten Gerätenummern durch den Aufruf der Funktion »blk_register_region()« an:

```
void blk_register_region(
   dev_t dev,
   unsigned long range,
   struct module *module,
   struct kobject *(*probe)(dev_t,int
*,void *),
   int (*lock)(dev_t,void *),
   void *data);
```

Allerdings sind damit die Gerätenummern nicht sofort exklusiv reserviert. Vielmehr informiert der Kernel den Treiber, sobald er auf eine Disk im entsprechenden Bereich zugreifen möchte, durch Aufruf von »probe()«. Diese Funktion sucht die passende »struct gendisk« und gibt einen Zeiger auf das in der Struktur eingebettete »kobject« zurück. Hat der Programmierer außerdem die Adresse einer Funktion »lock()« übergeben, ruft der Kernel diese zuvor auf.

Der Datentyp »dev_t« repräsentiert die neue Gerätenummer. Das Makro »MK_DEV(major, minor)« generiert aus der Major-Nummer »major« und der Minor-Nummer »minor« die Gerätenummer. Der Parameter »range« von »blk_register_region()« gibt die Anzahl der zu reservierenden Gerätenummern an.

Wird der Treiber deinitialisiert, nimmt er durch Aufruf von »blk_unregister_region(dev_t dev, unsigned long range)« die Reservierung wieder zurück (siehe auch [4]).

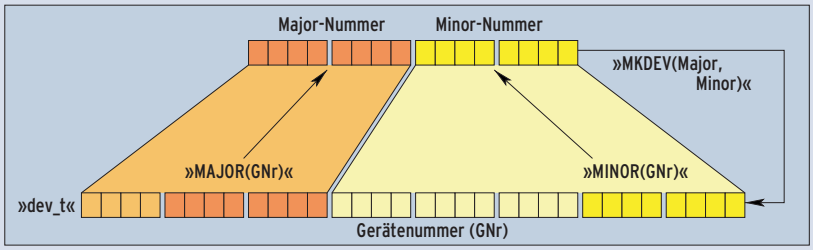


Abbildung 5: Eine eindeutige Vorschrift bildet die alten Major- und Minor-Nummern auf die neuen Gerätenummern ab. »MKDEV()« erzeugt aus Major/Minor eine Gerätenummer, die Makros »MAJOR()« und »MINOR()« sind für die Gegenrichtung zuständig.

- Anzeige -

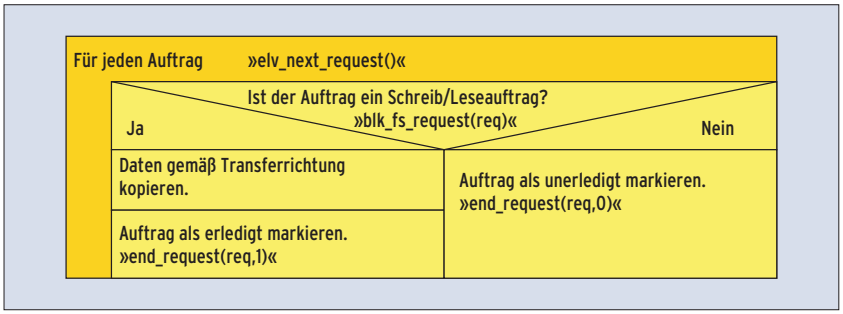


Abbildung 4: Das Struktogramm zeigt die Arbeitsweise der Transferfunktion eines Blockgerätetreibers. Die Routine »elv_next_request()« bildet eine Schleife für alle Aufträge an den Treiber.

```
if( !blk_fs_request(req) ) {
   // kein Schreib-/Leseauftrag
   end_request( req, 0 );
      continue;
}
```

Ist dies nicht der Fall, holt der Treiber mit »continue« den nächsten Auftrag aus der Liste. Handelt es sich um einen Schreib- oder Leseauftrag, arbeitet er ihn ab. Das Makro »rq_for_each_bio()« liefert dazu als Teil einer Schleife den jeweiligen BIO-Block zurück, den der Treiber dann wie beschrieben auswertet:

```
rq_for_each_bio(bio, req ) {
```

Hat er einen Auftrag abgearbeitet, ruft er »end_request()« auf. Der erste Parameter ist dabei der Auftrag selbst. Der zweite Parameter spezifiziert mit einem Wert ungleich null, dass der Auftrag bearbeitet wurde (siehe Abbildung 4).

Rückblick und Vorschau

Das Blockgerätemodell bietet eine einheitliche Programmierschnittstelle für reale Hardware zur Datenspeicherung – von der großen Festplatte bis zum USB-Stick. Um USB-Geräte geht es in der nächsten Folge der Kern-Technik, denn die haben mittlerweile RS-232 als Standardschnittstelle abgelöst. *(ofr)* ■

Infos:

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 1: Linux-Magazin 8/03, S. 88

[2] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur": Linux-Magazin 1/04, S. 28

[3] Jonathan Corbet, „Driver Porting: A simple block driver": [http://lwn.net/Articles/58719/]

[4] Jonathan Corbet, „Driver Porting: the gendisk interface": [http://lwn.net/Articles/25711/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Portierungshilfe

Im Kernel 2.6 repräsentiert nicht mehr die Tabelle der Treiber-Einsprungspunkte »struct block_device_operations« das Blockgerät, sondern die Struktur »struct gendisk«. Die Tabelle mit den Treiber-Einsprungspunkten trägt man nun in die »struct gendisk« ein, die darüber hinaus die Kenndaten des Geräts (der Disk) aufnimmt. Die dafür vorgesehenen globalen Variablen aus Kernel 2.4 existieren nicht mehr.

Die Übergabe des Objekts an das Blockgeräte-Subsystem hat sich ebenfalls geändert. Sie findet nicht mehr in »register_blkdev()«, sondern in der Funktion »add_disk()« statt. Außerdem braucht Linux 2.6 die Funktion »register_blkdev()« nicht mehr. Nur wer seinem Treiber dynamisch eine Minor-Nummer zuteilen lassen und ihn im Verzeichnis »/proc/devices« listen möchte, verwendet die Funktion weiterhin.

Lokale Queues

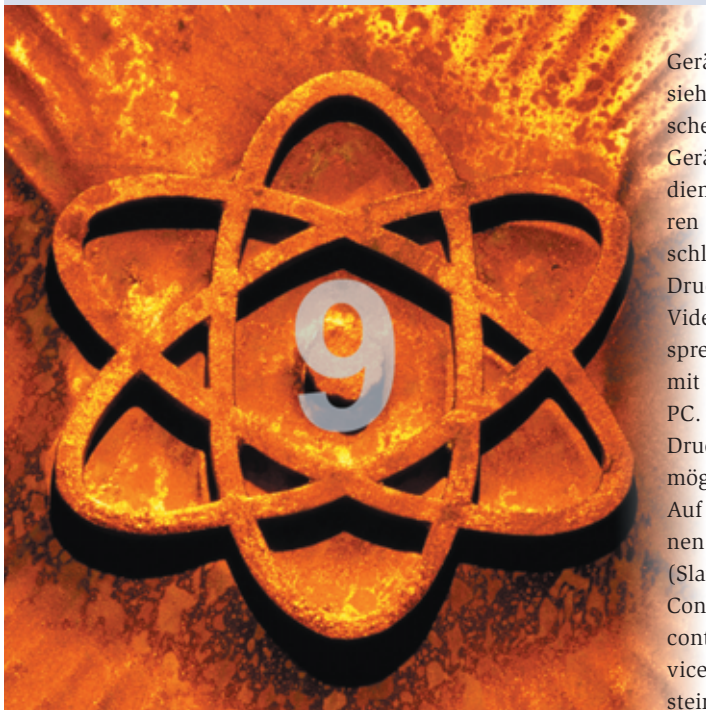
Auch die interne Pufferung hat sich verändert. Statt einer globalen Request-Queue bringt jedes Blockgerät seine eigene Warteschlange und folglich auch seinen eigenen Lock mit. Den Lock »io_request_lock« gibt es nicht mehr.

Der Zugriff auf die Requests erfolgt nicht mehr über das Makro »CURRENT«, sondern über »elv_next_request()«. Die aus Kernel 2.4 bekannten Buffer-Heads ersetzt Version 2.6 durch die BIO-Blöcke mit ihren eigenen Zugriffsfunktionen.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 9

Kern-Technik

USB hat die klassische serielle Schnittstelle als Allround-Interface abgelöst und macht mit Version 2.0 den Highspeed-Bussen Konkurrenz. Kernel 2.6 unterstützt schon ein breites Spektrum an Hardware und hilft unbekannte USB-Geräte direkt zu programmieren. Eva-Katharina Kunst, Jürgen Quade



Version 2.0 des Universal Serial Bus (USB) bringt attraktive Bandbreiten: Erreichte schon USB 1.1 Datenraten bis zu 12 MBit/s, sind jetzt 480 MBit/s das Maß aller Dinge [1]. Neben schneller Übertragung steht USB auch für die große Vielfalt moderner Peripheriegeräte. Grund genug für einen Blick auf die Treiberprogrammierung für USB-Devices, siehe [2]. Für den Zugriff auf USB-Geräte ist nicht unbedingt ein Kernelmodul nötig, denn mit der Libusb steht eine Bibliothek zur Verfügung, die den Zugriff aus dem Userspace ermöglicht [3]. Dieser Artikel zeigt, wie es im Kernelspace geht, sollte es einmal nötig sein.

Der USB verbindet Peripheriegeräte mit dem Rechner. Physisch ist USB als Baum strukturiert, mit dem USB-Controller als Wurzel (Master), den Hubs als Ästen (Abzweigungen) und den einzelnen Geräten als Blättern (Slaves), siehe Abbildung 1. Inzwischen gibt es kaum noch eine Geräteart, die USB nicht bedient – Mäuse oder Tastaturen eignen sich zum USB-Anschluss ebenso wie Scanner, Drucker, Digital- oder auch Videokameras. Die Endgeräte sprechen jedoch immer nur mit einem Master, meist dem PC. Den Scanner direkt am Drucker anschließen ist nicht möglich.

Auf Hardware-Ebene bedienen Master und Endgeräte (Slaves) den Bus über einen Controllerbaustein: den Hostcontroller respektive den Devicecontroller. Dieser Baustein wird durch den USB-Controller-Treiber angesteuert, der wiederum die IO-Ports der Hardware anspricht, siehe [4].

Meister und Gehilfe

Ähnlich wie bei Netzwerkkarten ist auch für Controllerbausteine unterschiedlichen Typs ein eigener Treiber erforderlich, siehe Abbildung 2. Der Controller-Treiber wird im Normalfall durch das USB-Subsystem angesteuert, das das Gerätemanagement übernimmt: So ist es dafür zuständig, Geräte zu finden, zu identifizieren und sie einem Treiber zuzuordnen. Darüber hinaus realisiert das USB-Subsystem eine Controller-unabhängige Schnittstelle für den gerätespezifischen Treiber.

Sowohl für den Host als auch für die Peripherie bringt Linux ein solches USB-Subsystem mit. Für die meisten erhältlichen Controller gibt es bereits Linux-Treiber. Auf dieser Ebene ist also glücklicherweise nicht mehr viel zu tun.

Anders liegt der Fall bei einem gerätespezifischen Treiber. Er sorgt beispielsweise bei einem USB-Scanner dafür, die richtige Gammakurve zu laden oder die Scannerlampe einzuschalten. Ein solcher Treiber, der im Gerät selbst werkelt, hängt stark von der verwendeten Hardware ab. Man nennt ihn Gadget-Treiber. Er ist vor allem für Entwickler eingebetteter Systeme von Interesse, eine Einführung hierzu findet sich in [5]. Der Artikel beschränkt sich den gerätespezifischen Host-Treiber des Masters.

Ab geht die Post

Der Host-Treiber spricht mit seinem USB-Gerät über eine Paketschnittstelle. Ein Paket heißt URB (USB Request Block) und enthält den Adressaten, Transferinformationen und die Daten selbst. Die Transferinformation gibt die Art der Übertragung an, bei USB eine von vier Möglichkeiten:

- Kontroll-Transfer (Control): Kurze Pakete mit Konfigurationsdaten für das Gerät beziehungsweise Statusinformationen vom Gerät.
- Massen-Transfer (Bulk): Pakete zum normalen Datenaustausch zwischen Controller und Gerät.
- Interrupt-Transfer (Int): Periodischer Datentransfer zwischen Controller und Gerät.
- Isochroner Transfer (Iso): Übertragung in Realzeit und mit fester Bandbreite.

Eine Gerätenummer, eine Interfacenummer und ein Endpunkt definieren den Adressaten einer solchen Nachricht.

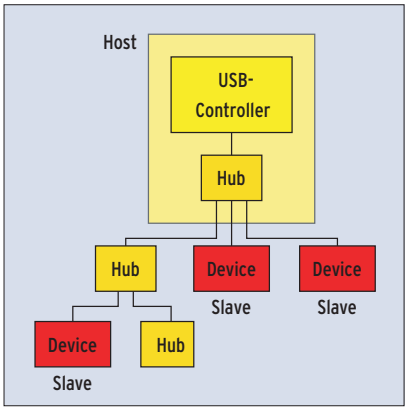


Abbildung 1: USB in Baumform: Geräte hängen über einen oder mehrere Hubs am Host.

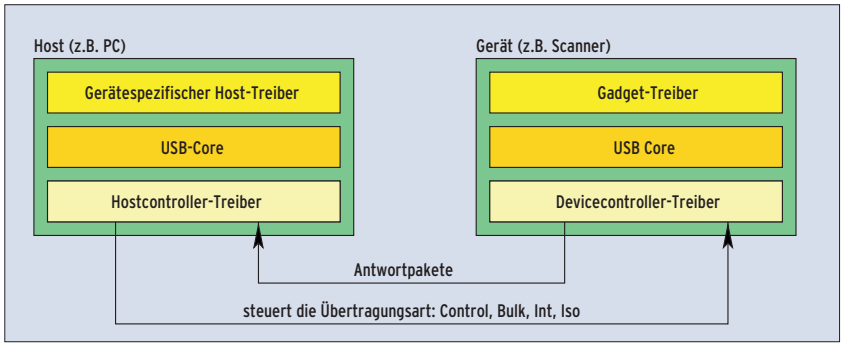


Abbildung 2: Im Host und im Endgerät finden sich jeweils drei Schichten für die USB-Kommunikation.

Der grundlegende Aufbau des USB-Systems unter Linux ähnelt dem von PCI: Da USB-Geräte ebenfalls hotpluggable sind, ist die Treiberinitialisierung von der Geräteinitialisierung getrennt. Laden Anwender oder der Modul-Autoloader den Treiber, ruft der Kernel die Funktion »ModInit()« auf. Sie registriert den Treiber mit »usb_register()« beim USB-Subsystem und zeigt diesem an, für welche Geräte er zuständig ist, siehe Listing 1, Zeilen 86 bis 90.

Dazu dient die Struktur »usb_dev_id«, deren Aufbau und Verwendung der Kasten „USB-Gerätemerkmale“ näher beschreibt. Die Funktion »usb_register()« übernimmt neben den Gerätemerkmalen in der »struct usb_driver« noch weitere Informationen: den Treibernamen, die Adresse der Funktion zur Geräteinitialisierung »probe()« und die Adresse der Deinitialisierungsfunktion »disconnect()«, siehe Zeile 77 bis 82.

Sobald das USB-Subsystem ein Gerät ortet, das der vom Treiber übergebenen Gerätekennung entspricht, initialisiert es das Device. Die »probe()«-Funktion überprüft, ob das Gerät vom Treiber bedient wird. Falls ja, muss der Kernel die Userseite darüber informieren, dass ein neues Gerät vorhanden ist.

Wie die Applikation auf die neue Peripherie zugreift, hängt von der Art des Geräts ab. Handelt es sich beispielsweise um einen USB-Netzwerkadapter, meldet der Treiber ihn beim Netzwerk-Subsystem an. Gehört das Gerät jedoch keiner derartigen Klasse an, kann es als Character-Device angesprochen werden.

Kernel, übernehmen Sie!

Weil Major-Nummern knapp sind, meldet das USB-Subsystem im Kernel 2.6 sich stellvertretend für alle USB-Geräte als zeichenorientierter Treiber mit der Major-Nummer 180 an. Im Kernel 2.4 hatte das USB-Subsystem noch jedem Treiber eine eigene Minor-Nummer zugeteilt. Beim neuen Kernel muss ein Treiber diese Minor-Nummer explizit anfordern. Dazu spezifiziert der Treiber das Gerät während der Initialisierung in der Datenstruktur »struct usb_class_driver«, siehe Listing 1, Zeilen 48 bis 53.

Zu den Feldern der Struktur zählen der Treibername und die Zugriffs- und Geräteart (zum Beispiel zeichenorientiertes Gerät). Zusätzlich muss das Modul die Tabelle mit Funktionen »struct file_operations« übergeben, die der Kernel aufruft, wenn eine Applikation auf die zugehörige Gerätedatei zugreift. Achtung: Von diesen Funktionen ist zumindest die Funktion »DriverOpen()« zu implementieren, selbst wenn sie nur aus der Zeile »return 0« besteht.

Ruft der Treiber »usb_register_dev()« auf, teilt ihm das USB-System dynamisch eine Minor-Nummer zu, die sich dann im Feld »minor« befindet. Für den Zugriff auf das Device muss eine Gerätedatei mit dieser Minor-Nummer existieren. Ist das Device-Filesystem aktiviert, legt es diese Gerätedatei automatisch im Verzeichnis »/dev/« an, ansonsten ist hier Handarbeit gefordert (Aufruf von »mknod«, siehe Kasten „Ärmel hochkrempeln“).

Blockweise Requests

Nun können Anwendungen auf das USB-Gerät zugreifen. Der Treiber schickt dann USB Request Blocks (URBs) an das Gerät und empfängt entsprechende Antworten. In einem URB, definiert in »include/linux/usb.h«, trägt die Anwendung alle Parameter ein, die der USB-Core dazu benötigt, den Datentransfer zwischen Controller und Peripheriegeräten abzuwickeln.

Dazu gehören die Endadresse (Gerät, Interface, Endpunkt), die eigentlichen Daten, die Länge der zu übertragenden Daten, ein Timeout für die Übertragung sowie die Adresse einer Callback-Funktion. Die ist nötig, weil USB asynchron arbeitet. Eine Instanz – etwa die Lese-

USB-Gerätemerkmale

Ein USB-Gerät besitzt bis zu neun Merkmale: Produkt-ID, Versionsnummernbereich (Low und High), Geräteklasse, Geräteunterklasse, Geräteprotokoll, Interfaceklasse, Interfaceunterklasse und Interfaceprotokoll. Ein Treiber muss nicht alle Merkmale belegen. Vielmehr ist es möglich, ein Gerät nur aufgrund einiger Merkmale auszuwählen.

Anders als bei PCI gibt es allerdings für die einzelnen Attribute keine Wildcards. Stattdessen gibt ein Bitfeld in der Struktur »struct usb_device_id« an, ob ein bestimmtes Merkmal zur Identifikation herangezogen werden soll oder nicht (so genannte Match-Flags). Ein gesetztes Bit sorgt dafür, dass das USB-Subsystem das Merkmal auswertet.

Unterstützung durch Makros

Der Programmierer muss die notwendigen Bit-Operationen nicht von Hand durchführen, die Headerdatei »linux/usb.h« enthält dafür Makros. »USB_DEVICE()« bekommt beispielsweise die Parameter Herstellerkennung und Produkt-ID übergeben und erstellt aus diesen Angaben eine gültige Version der »struct usb_device_id«. Ähnlich übernimmt »USB_DEVICE_VER()« zusätzlich zu Herstellerkennung und Produkt-ID den Versionsnummernbereich.

funktion in einem Treiber – übergibt dem USB-Subsystem einen URB. Während der dort spezifizierte Auftrag abgearbeitet wird, kann die Treiberfunktion weiterlaufen, blockiert also nicht. Sobald das Subsystem mit dem Auftrag fertig ist, ruft es die Callback-Funktion auf.

USB ruft zurück

Ein URB wird durch die Funktion »usb _alloc_urb()« erzeugt. Ist die URB-Datenstruktur angelegt, muss das Modul als Nächstes deren Felder ausfüllen (Initialisierung des URB). Die Headerdatei »include/linux/usb.h« definiert für die Transferarten Control, Bulk und Int entsprechende Makros: »usb_fill_control _urb()«, »usb_fill_bulk_urb()« sowie »usb_fill_int_usb()«. Ist der URB fertig gestellt, schickt der Treiber ihn ab. Die Funktion »usb_submit_urb()« startet den Transfervorgang und »usb_unlink_ urb()« bricht ihn notfalls wieder ab. Die Callback-Funktion (Element »complete«) übernimmt gegebenenfalls das Ergebnis und gibt den URB mit Hilfe der Funktion »usb_free_urb()« wieder frei.

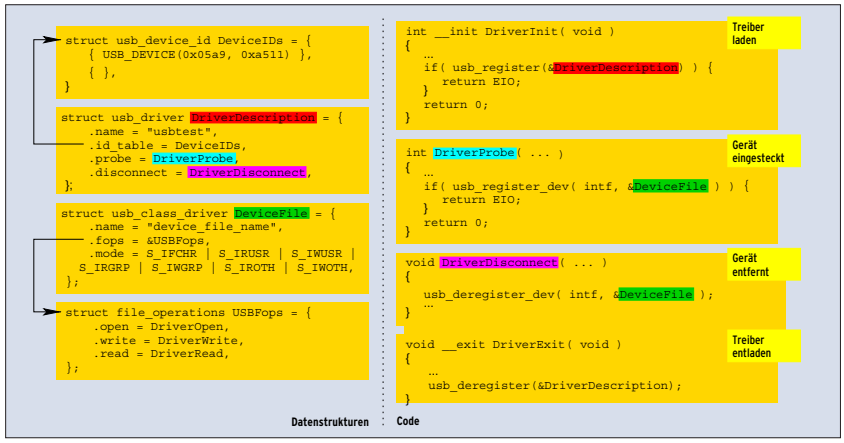


Abbildung 3: Funktionen eines USB-Host-Treibers mit den von ihm benutzten Datenstrukturen.

Listing 1: Treibergerüst für USB

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/usb.h>
06 #include <asm/uaccess.h>
07
08 MODULE_LICENSE( "GPL" );
09
10 #define USB_VENDOR_ID 0x05a9
11 #define USB_DEVICE_ID 0xa511
12
13 struct usb_device *dev;
14
15 static ssize_t usbtest_read( struct file
   *Instanz, char *buffer, size_t count,
16                 loff_t *ofs )
17 {
18    __u16 status;
19    char pbuf[20];
20
21    if( usb_control_msg(dev, usb_rcvctrlpipe(dev,
   0), USB_REQ_GET_STATUS,
22                            USB_DIR_IN |
   USB_TYPE_STANDARD | USB_RECIP_INTERFACE,
23                            0, 0, &status,
   sizeof(status), 5*HZ) < 0 )
24           return -EIO;
25    snprintf( pbuf, sizeof(pbuf), "status=%d\n",
   status );
26    if( strlen(pbuf) < count )
27           count = strlen(pbuf);
28    count -= copy_to_user(buffer,pbuf,count);
29    return count;
30 }
31
32 static int usbtest_open( struct inode
   *devicefile, struct file *Instanz )
33 {
34    return 0;
35 }
36
37 static struct file_operations USBFops = {
38    .owner = THIS_MODULE,
39    .open  = usbtest_open,
40    .read  = usbtest_read,
41 };
42
43 static struct usb_device_id usbid [] = {
44    { USB_DEVICE(USB_VENDOR_ID, USB_DEVICE_ID), },
45    { }                  /* Terminating entry */
46 };
47
48 struct usb_class_driver ClassDescr = {
49    .name = "usbtest",
50    .fops = &USBFops,
51    .mode = S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP
   | S_IWGRP | S_IROTH | S_IWOTH,
52    .minor_base = 16,
53 };
54
55 static int usbtest_probe(struct usb_interface
   *interface,
56         const struct usb_device_id *id)
57 {
58    dev = interface_to_usbdev(interface);
59    printk("USBTEST: 0x%4.4x|0x%4.4x, if=%p\n",
   dev->descriptor.idVendor,
60           dev->descriptor.idProduct, interface
   );
61    if(dev->descriptor.idVendor==USB_VENDOR_ID
62           && dev->descriptor.idProduct ==
   USB_DEVICE_ID) {
63           if( usb_register_dev( interface,
   &ClassDescr ) ) {
64                  return -EIO;
65           }
66           printk("got minor= %d\n", interface-
   >minor );
67           return 0;
68    }
69    return -ENODEV;
70 }
71
72 static void usbtest_disconnect( struct
   usb_interface *iface )
73 {
74    usb_deregister_dev( iface, &ClassDescr );
75 }
76
77 static struct usb_driver usbtest = {
78    .name= "usbtest",
79    .id_table= usbid,
80    .probe= usbtest_probe,
81    .disconnect= usbtest_disconnect,
82 };
83
84 static int __init usbtest_init(void)
85 {
86    if( usb_register(&usbtest) ) {
87           printk("usbtest: unable to register
   usb driver\n");
88           return -EIO;
89    }
90    return 0;
91 }
92
93 static void __exit usbtest_exit(void)
94 {
95    usb_deregister(&usbtest);
96 }
97
98 module_init(usbtest_init);
99 module_exit(usbtest_exit);
```

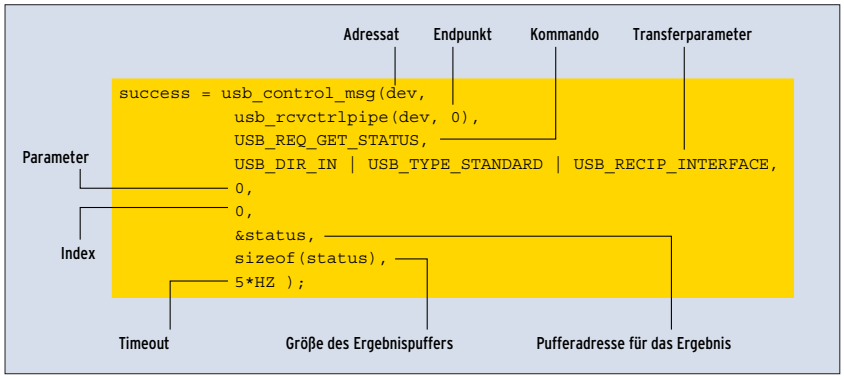


Abbildung 4: High-Level-Funktionen wie »usb_control_msg()« helfen dabei, URBs zu definieren und an Geräte zu versenden. Sie enthalten den Adressaten, die Transferparameter und einen Puffer für den Rückgabewert.

Für die Transferarten Control und Bulk gibt es Hilfsfunktionen, die das Erstellen und Verschicken eines URB übernehmen: »usb_control_msg()« oder für Bulk »usb_bulk_msg()«. Die Funktionen schicken die Daten an einen definierten Endpunkt und warten, bis der Transfer entweder beendet ist oder ein Timeout erfolgt. Sie geben anschließend die Anzahl der übertragenen Bytes zurück, im Fehlerfall einen negativen Wert.

Abbildung 4 zeigt, wie eine Statusanfrage verschickt wird. Von den drei Adressangaben (Geräte, Interface und Endpunkt) erscheinen hier nur zwei, denn das Interface ist bereits per Gerät »dev« ausgewählt. Der Endpunkt ist eine Unix-Pipe, sie wird durch ein Bitfeld repräsentiert, das die Geräteadresse, die Endpunktnummer, die Transferrichtung und die Transferart angibt. Für die Pipe-Funktionalität stehen Makros zur Verfügung, deren Name sich zusammensetzt aus: dem Präfix »usb_«, der Transferrichtung »snd« oder »rcv«, der Transferart »ctrl«, »bulk«, »iso« oder »int« sowie dem Abschluss »pipe«. Bei dem in der Abbildung verwendeten »usb_rcvctrlpipe« handelt es sich also um den Empfang (»rcv«) von Statusinformationen (»ctrl«) des Geräts.

Ein weiterer Parameter gibt den Befehl an, den das Gerät ausführen soll. Das können gerätespezifische Kommandos wie »USB_TYPE_VENDOR« oder Standardkommandos sein. Dafür gibt es folgende USB-Sendefunktionen:

- usb_get_descriptor()
- usb_get_device_descriptor()
- usb_get_status()
- usb_get_string()
- usb_string()
- usb_set_configuration()
- usb_get_interface()

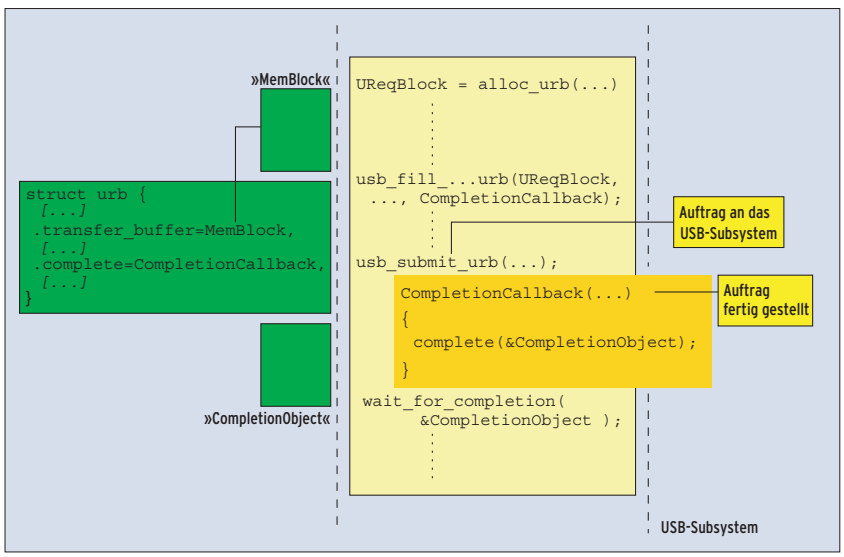


Abbildung 5: Bei asynchroner USB-Kommunikation legt der Treiber den URB an und übergibt eine Callback-Funktion, die das USB-Subsystem aufruft, wenn es den Request abgearbeitet hat.

Der nächste Parameter besteht aus der Transferrichtung aus Sicht des Hosts, der Art des Kommandos und einer Adressatenauswahl. Die Transferrichtung ist »USB_DIR_IN« oder »USB_DIR_OUT«. Als Adressaten stehen zur Auswahl: »USB_RECIP_MASK«, »USB_RECIP_DEVICE«, »USB_RECIP_INTERFACE«, »USB_RECIP_ENDPOINT« oder »USB_RECIP_OTHER«. Dann folgen noch ein optionaler Parameter sowie ein Index. Bei einem Kommando an das Gerät ist der Index »0«. Ein Befehl an das Interface enthält die Interfacenummer, ein Kommando an den Endpunkt und die Endpunktnummer.

Der nächste Parameter spezifiziert den Speicherbereich, in dem der Gerätetreiber das Ergebnis ablegt. Die beiden letzten Parameter bezeichnen die Speicherlänge und den Timeout des Requests.

Massenversand

Beim Bulk-Transfer heißt die Funktion entsprechend »usb_bulk_msg()«. Anders als »usb_control_msg()« besitzt sie nur die folgenden sechs Parameter: das Gerät, die Pipe, den Speicherblock samt Daten, die Länge des Speicherblocks, die Adresse einer Variablen, die das Ergebnis aufnimmt, sowie einen Timeout:

```
ult = usb_bulk_msg(dev, usb_rcvbulkpipe
  (dev, 0), buffer, count, &count, HZ*2 );
```

Wie »usb_control_msg()« versendet die »usb_bulk_msg()« das Paket und wartet auf eine Antwort. Achtung: Da die beiden Funktionen synchron arbeiten, dürfen sie nicht im Interrupt-Kontext eingesetzt werden. Bei asynchroner Kommunikation (Interrupt- und Isochron-Transfer) wartet das Modul nicht auf das Er-

Listing 2: Makefile zum Treibergerüst

```
01 TARGET=usbtest
02
03 ifneq ($(KERNELRELEASE),)
04 obj-m   := ${TARGET}.o
05
06 else
07 KDIR    := /lib/modules/$(shell uname -r)/build
08 PWD     := $(shell pwd)
09
10 default:
11         $(MAKE) -C $(KDIR)      SUBDIRS=$(PWD) modules
12 endif
```

Ärmel hochkrempeln

Besitzen Sie ein USB-Gerät, können Sie die beschriebenen Grundmechanismen eines USB-Treibers testen. Den Code des Treibers finden Sie in Listing 1. Im Quelltext tragen Sie die Hersteller-ID und die Produkt-ID (Zeilen 10 und 11) Ihres spezifischen Geräts ein. Außerdem müssen Sie noch dafür sorgen, dass der für das Gerät ursprünglich vorgesehene Treiber nicht geladen wird. Am besten gehen Sie folgendermaßen vor:

1. Stellen Sie sicher, dass Ihr Kernel USB unterstützt: Dazu muss USB-Support im Kernel konfiguriert sein, siehe auch [7]. Falls der USB-Support über Module erfolgt, stellen Sie durch Aufruf von »lsmod« fest, ob die Hostcontroller-Treiber mit den Namen »ehci_hcd«, »uhci_hcd« oder »ohci_hcd« geladen sind. Falls nicht, laden Sie den passenden mit »modprobe uhci_hcd«, »modprobe ehci_hcd« oder »modprobe ohci_hcd«.

2. Identifizieren Sie die Hersteller- und Geräte-ID Ihrer Peripherie. Dafür ist das Programm »lsusb« hilfreich. Es bezieht seine Informationen aus dem USB-Filesystem, das Sie per »mount -t usbfs none /proc/bus/usb« in den Verzeichnisbaum einhängen (falls nicht ohnehin bereits geschehen).
Stecken Sie das Gerät an den USB-Bus und rufen »lsusb« auf. In der Ausgabe sollten Sie Ihre Peripherie wiederfinden:

```
Bus 004 Device 001: ID 0000:0000
Bus 003 Device 032: ID 05a9:a511
   OmniVision Technologies, Inc. [...]
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 001: ID 0000:0000
```

Das Beispiel verwendet eine Webcam zum Test der Mechanismen innerhalb des USB-Subsystems. Die Ausgabe zeigt, dass die Webcam die Hersteller-ID »0x05a9« und die Produkt-ID »0xa511« hat. In Listing 1 sind diese IDs bereits eingetragen.

3. Damit das Hotplug-System den ursprünglich vorgesehenen Treiber nicht automatisch lädt, tragen Sie ihn in »/etc/hotplug/blacklist« ein. Sie finden den Treiber über das File »/lib/modules/*Kernelversion*/modules.usbmap«.

4. Wenn Sie den Quellcode modifiziert und den Treiber übersetzt haben, laden Sie diesen mit »insmod usbtest.ko«. Falls Sie das Device-Filesystem aktiviert haben, wird vom USB-Subsystem automatisch eine passende Gerätedatei angelegt.
Falls nicht, legen Sie die Gerätedatei selbst an. Bei USB ist die Major-Nummer immer »180«. Die Minor-Nummer hat der Treiber in den Syslogs ausgegeben, innerhalb des Treibers wird die Minor-Nummer »16« angefordert.

```
mknod devicefile c 180 16
```

5. Der Treiber ist so implementiert, dass – wenn Sie von der Gerätedatei mit »cat <devicefile« lesen – er den Status zurückgibt:

```
status = 0
...
```

Falls Sie das Gerät jetzt abstecken, bricht »cat« mit einem Ein-/Ausgabefehler ab.

gebnis. Hier muss der Programmierer selbst URBs anlegen, initialisieren und per »usb_submit_urb()« verschicken (Abbildung 5). Das Modul legt den URB an und initialisiert die Transportart. Dabei übergibt es eine Callback-Funktion, die vom USB-Subsystem aufgerufen wird, sobald es den Auftrag abgearbeitet hat. Zur Synchronisation der Aufträge dient ein Completion-Objekt [6].

Und tschüs!

Wird das Gerät abgeklemmt, informiert das USB-Subsystem den Treiber über die »disconnect()«-Funktion, die der Treiber bereits bei der Anmeldung übergeben hat (im Element ».disconnect« der Datenstruktur »usb_driver«, siehe Abbildung 3). Die Funktion beendet alle aktiven Zugriffe auf das Gerät und vermerkt, dass es entfernt worden ist.
Das Subsystem ruft die Funktion »disconnect()« allerdings auch dann auf, wenn das Treibermodul entladen wird. Im einfachsten Fall meldet sich der Treiber wieder bei jenem USB-Subsystem ab, bei dem er sich bei der Geräteinitialisierung zuvor registriert hatte. Dann reicht zum Abmelden die Funktion »usb_deregister_dev()« aufzurufen, siehe Listing 1, Zeilen 72 bis 74. Für den nun fälligen Test des Treibers beschreibt der Mini-Workshop im Kasten „Ärmel hochkrempeln“ alles Notwendige.

Portierung auf Kernel 2.6

Wie in Kernel 2.4 meldet sich der USB-Treiber mit »usb_register()« beim USB-Subsystem an. Allerdings hat sich die als Parameter übergebene »struct usb_driver()« geändert: Der Treiber nutzt nicht mehr automatisch die Major-Nummer des USB-Subsystems. Soll der Treiber auf die Major-Nummer zurückgreifen, muss er unter Kernel 2.6 die »usb_class_driver« ausfüllen, darunter einen Zeiger auf die »struct file_operations«. Für Treiber, die die Major-Nummer nicht verwenden, entfällt dieser Schritt.
Die Funktion »usb_register_dev()« übergibt die »struct usb_class_driver« dem USB-System. Ist das Device-Filesystem aktiviert, erstellt Kernel 2.6 automatisch eine Gerätedatei. Mit »usb_deregister_dev()« gibt der Treiber die ihm zugewiesene Minor-Nummer wieder frei.
Die Funktion »usb_submit_urb()« ist in Kernel 2.6 um einen Parameter reicher: Nun steuern Flags die Speicherreservierung. So lässt sich mit »GFP_KERNEL« das von der älteren Version bekannte Verhalten wieder herstellen. Auch die »probe()«-Funktion erscheint in neuem Gewand. Der erste Parameter ist nun vom Typ »usb_interface«:

```
void *USBDriverProbe( struct usb_device
   *dev, unsigned intf, const struct
   usb_device_id *id );
```

Vorschau

Die nächste Folge der Kern-Technik erklärt, wie die Parameterübergabe an dynamische Kernelmodule unter Linux 2.6 funktioniert und wie Module untereinander Daten und Funktionen austauschen können. *(ofr)* ■

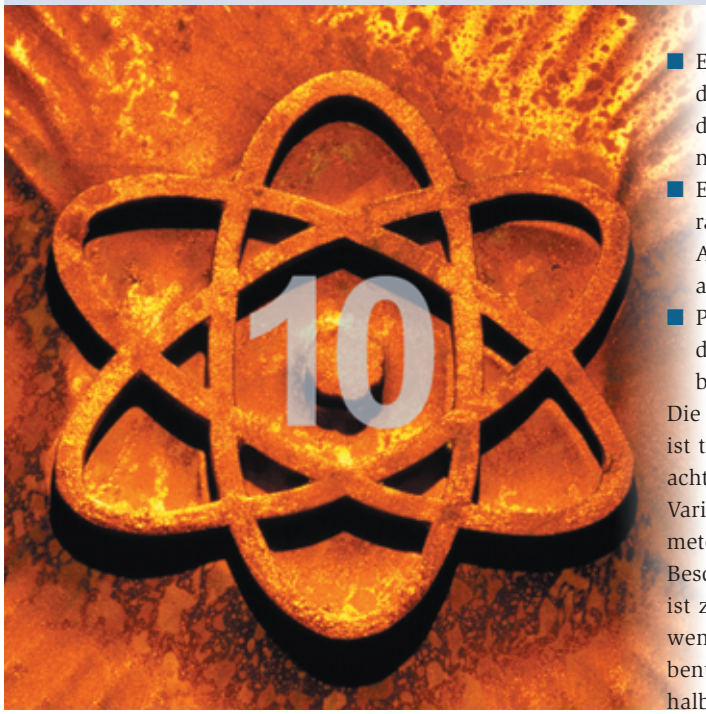
Infos:

[1] USB-Spezifikationen: [http://www.usb.org/developers/docs]
[2] David Brownell, „Linux and USB 2.0“: [http://www.linux-usb.org/usb2.html]
[3] Johannes Erdfelt, „Libusb Developers Guide“: [http://http://libusb.sourceforge.net/]
[4] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 3: Linux-Magazin 10/03, S. 81
[5] David Brownell, „USB Gadget API for Linux“: [http://libusb.sourceforge.net/doc/]
[6] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 4: Linux-Magazin 11/03, S. 96
[7] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur“: Linux-Magazin 2/04, S. 28

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 10

Kern-Technik

Aus Platz- und Übersichtsgründen lagert der Kernel seine Funktionalität größtenteils in Module aus. Dieser Vorgang lässt sich über Parameter fein einstellen. Linux 2.6 bietet Typsicherheit für Modulparameter und erlaubt es dem Kernelprogrammierer, dafür eigene Datentypen zu verwenden. Eva-Katharina Kunst, Jürgen Quade



Auch bei Modulen bringt Linux 2.6 im Vergleich zu 2.4 kleine Veränderungen mit: Kernelprogrammierer müssen Parameter nun anders implementieren, können aber eigene Datentypen definieren. Wenn der neue Kernel dynamische Module lädt und dabei Parameter übergeben bekommt, kann er sie auf ihre Gültigkeit überprüfen. IO-Adressen, Speicherbereiche, Gerätemerkmale und Betriebsarten sind nur einige Anwendungsbeispiele dafür.

Die Parameter eines Moduls bestehen aus zwei Komponenten: dem Namen und dem Wert. Die dafür vordefinierten Datentypen zeigt Tabelle 1. Den Datentyp »byte«, im Kernel 2.4 noch vorhanden, gibt es in 2.6 nicht mehr. Um in einem eigenen Modul einen Parameter zu implementieren, muss der Programmierer Folgendes tun:

- Eine Variable definieren, die im laufenden Modul den Parameterwert aufnimmt.
- Eine Beschreibung des Parameters erstellen, die der Anwender des Moduls auslesen kann.
- Per Makro den Parameter dem Kernel bekannt geben.

Die Definition der Variablen ist trivial. Hier ist nur zu beachten, dass der Datentyp der Variablen mit dem des Parameters übereinstimmt. Die Beschreibung des Parameters ist zwar nicht zwingend notwendig, aber für den Modulbenutzer oft nützlich, deshalb sollte der Kernelprogrammierer diese Zusatzarbeit leisten. Dazu übergibt er dem Makro »MODULE_PARM_DESCR()« den beschreibenden Text. Der Anwender liest diese Beschreibung mit dem Programm »mod_info«.

Makros für alle

Um den Parameter dem Kernel bekannt zu machen, stehen gleich zwei Makros zur Auswahl. Am einfachsten ist es, wenn der Name des Parameters mit dem Namen der Variablen übereinstimmt. In diesem Fall kommt das Makro »module_param()« zum Einsatz. Es bekommt den Variablen- beziehungsweise den Parameternamen übergeben, den Datentyp (siehe Tabelle 1) und einen Dateizugriffsmodus. In einer der kommenden Kernelversionen wird das Gerätemodell [1] nämlich eine Gerätedatei mit dem Namen des Parameters erstellen. Der Zugriffsmodus auf diese Gerätedatei wird dann dem an dieser Stelle angegebenen Zugriffsmodus entsprechen.

Die Funktionen und Makros für Modulparameter sind in »linux/moduleparam.h« definiert. Listing 1 zeigt hierzu ein Beispiel: ein Modul, dem der Benutzer beim Laden den Parameter »myint« vom Typ »int« übergeben kann. Aus »param1.c« (Listing 1) erzeugt der Compiler nach Anleitung des Makefile (Listing 2) das Modul »param1.ko«, das der Befehl »insmod« in den laufenden Kernel lädt (Abbildung 1). Liegt der angegebene Parameter außerhalb des Definitionsbereichs, wird das Modul nicht geladen. So entspricht der Wert »abcd« nicht der Definition eines Integer-Parameters.

Nur mit einem gültigem Parameter wie »myint=1234« lässt sich das Modul laden. Das Syslog (meist »/var/log/messages«) zeigt den Wert des Parameters, den das Modul mit »printk()« ausgibt (Lis-

Tabelle 1: Mögliche Datentypen für Modulparameter

Bezeichnung	Beschreibung
short	Short-Wert mit Vorzeichen
ushort	Short-Wert ohne Vorzeichen
int	Integer-Wert mit Vorzeichen
uint	Integer-Wert ohne Vorzeichen
long	Long-Wert mit Vorzeichen
ulong	Long-Wert ohne Vorzeichen
charp	Zeiger auf einen String
bool	boolescher Wert, mögliche Werte sind »y«, »Y«, »n«, »N«, »0«, »1«
Array	Feld eines der Standard-Datentypen
String	Zeichenfeld

ting 1, Zeile 12). Der Befehl »rmmod param1« entfernt das Modul wieder.

Stimmen die Namen von Variable und Parameter hingegen nicht überein, hilft das Makro »module_param_named()« weiter. Es besitzt als erstes Argument den Namen des Modulparameters, als zweites den Variablennamen im Modul. Die letzten beiden Parameter von »module_param_named()« repräsentieren wieder den zugehörigen Datentyp und die Zugriffsrechte. Listing 3 zeigt die Verwendung des Makros.

Für einen Test lässt sich das Modul wieder wie oben beschrieben laden. Nur lautet jetzt der Name des Modulparameters »maxtime«, während die modulinterne Variable den alten Namen »myint« weiterhin behält.

Felder und Strings

Wie Tabelle 1 zeigt, stehen neben den Standarddatentypen auch Feld- beziehungsweise String-Datentypen zur Auswahl. Sie in einem Kernelmodul einzusetzen ist etwas schwieriger, doch auch für sie existieren jeweils eigene Makros. Außerdem braucht das Modul für Strings und Felder zwei Variablen, denn der Kernel muss über Feld- und Stringlänge Buch führen.

Zusätzlich teilt er dem Treiber mit, wie viele Elemente des Parameters der Anwender tatsächlich beim Aufruf verwendet hat. Leider ist hierbei nicht zu unterscheiden, ob der Benutzer sämtliche Elemente vorbelegt hat oder ob der Parameter gänzlich unbesetzt blieb.

Das Makro zur Übergabe eines Felds als Modulparameter heißt »module_param_array()«. Es besitzt ebenfalls vier Parameter: Variablen- beziehungsweise Parametername, Datentyp des Feldes, Anzahl der Feldelemente und schließlich die Zugriffsrechte. Eine Variante, bei der Variablenname und Parametername nicht übereinstimmen müssen, existiert ebenfalls, und zwar »module_param_array_named()«. Listing 4 zeigt den Einsatz der Makros anhand eines Beispiels. Es wertet die Zählervariable aus und zeigt die Anzahl der vom Anwender übergebenen Variablen an.

Für Strings als Modulparameter stehen die beiden Makros »module_param_string()« und »module_param_string_named()« zur Verfügung. Ihre Parameter entsprechen denen der Feldmakros. Listing 5 zeigt ihre Anwendung.

Typ-Überprüfung

Der Modullader überprüft, ob Parameter und Datentyp übereinstimmen. Ist dies nicht der Fall, entfernt er das Modul gleich wieder aus dem Kernel. Das gilt auch für die selbst definierten Datentypen, doch es ist schwieriger, die Typüberprüfung zu implementieren: Der Programmierer muss dazu eine so genannte Set-Funktion bereitstellen, die überprüft, ob der angegebene Parameterwert in den Wertebereich des Datentyps passt oder nicht. Die Get-Funktion konvertiert dagegen ein übergebenes Datum wieder in die ursprüngliche Darstellung. Ein kurzes Beispiel zeigt, wie man die Funktionen verwendet.

Der selbst definierte Datentyp soll nur gerade Werte und die Zahl 0 zulassen (0, 2, 4, 6 …) und deshalb »even« heißen. Die Funktion, die den übergebenen Parameterwert überprüft, soll »param_set_even()« heißen. Sie bekommt vom Kernel je einen Zeiger auf einen String und auf ein Element vom Typ »struct kernel_param« übergeben. Diese Struktur repräsentiert Kernel-intern die Modulparameter. Sie enthält unter anderem einen Zeiger auf die Variable, in die der Modullader den Wert abgelegt (das Element »void *arg«). Sobald der Modullader ein

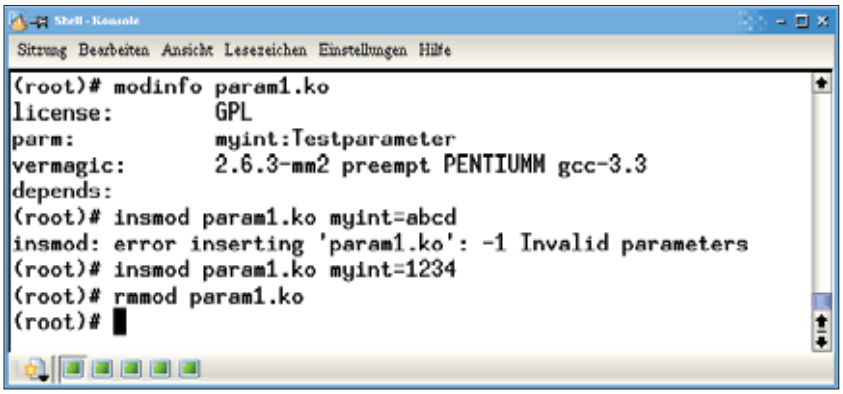


Abbildung 1: Der Versuch, mit »insmod« das Beispiel aus Listing 1 in den laufenden Kernel zu laden, funktioniert nur, wenn der Parameter innerhalb des definierten Wertebereichs liegt.

Listing 1: Parameter mit »module_param()«

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03
04 MODULE_LICENSE("GPL");
05 MODULE_PARM_DESC(myint, "Testparameter");
06
07 static int myint;
08 module_param( myint, int, 666 );
09
10 static int __init mod_init(void)
11 {
12         printk("myint = %d\n", myint);
13         return 0;
14 }
15
16 static void __exit mod_exit(void)
17 {
18 }
19 module_init( mod_init );
20 module_exit( mod_exit );
```

Listing 2: Makefile für »param1.c«

```
01 TARGET=param1
02
03 ifneq ($(KERNELRELEASE),)
04 obj-m   := ${TARGET}.o
05
06 else
07 KDIR    := /lib/modules/$(shell uname -r)/build
08 PWD     := $(shell pwd)
09
10 default:
11         $(MAKE) -C $(KDIR)      SUBDIRS=$(PWD) modules
12 endif
```

Listing 3: Makro »module_param_named()«

```
01 ...
02 static int myint;
03 module_param_named(maxtime, myint, int, 666);
04
05 static int __init mod_init(void)
06 {
07    printk("myint = %d\n", myint);
08    return 0;
09 }
10 ...
```

Modul mit einem selbst definierten Parameter lädt, ruft der Kernel die Funktion »param_set_even()« auf. Sie konvertiert den als String übergebenen Parameter in den internen Datentyp und prüft, ob er innerhalb des Wertebereichs liegt, und speichert einen gültigen Wert ab.

Listing 6 zeigt in den Zeilen 13 bis 17 die Implementierung. Der Wert selbst wird durch »simple_strtoul()« in einen Integerwert konvertiert (Zeile 13). Die folgende Abfrage überprüft mit Hilfe der Modulo-Funktion, ob der Wert gerade oder ungerade ist (Zeile 14). Wenn er ungerade ist, gibt die Testfunktion »-EINVAL« (ungültiger Wert) zurück (Zeile 15), sonst »0«.

Die Get-Funktion hat die Aufgabe, die interne Form des Parameters zurück in eine Ascii-Kodierung zu transformieren. Das geht am einfachsten mit »sprintf()«, wobei nur sicherzustellen ist, dass nicht mehr Bytes geschrieben werden als in einer Speicherseite Platz haben (Zeile 22). Die Adresse, an die »sprintf()« das Ergebnis schreiben soll, ist als Parameter der Get-Funktion übergeben worden: »char *pbuf«.

Neben den beiden Funktionen muss der Programmierer noch ein Makro definieren, das unter Umständen eine Prüfung des Typs durch den Compiler ermöglichen könnte. In fast allen Fällen kann das Makro aber auch einfach leer bleiben.

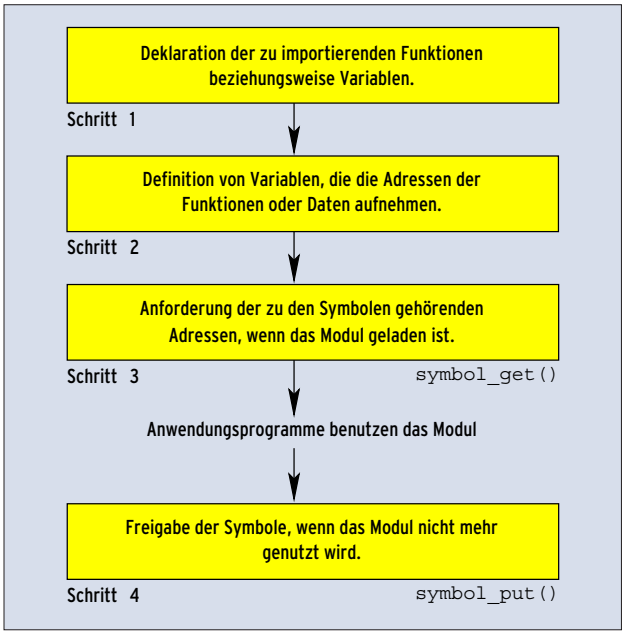


Abbildung 2: Vier Schritte zur Implementierung von Intermodul-Kommunikation.

Listing 4: Übergabe von Feldern

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03
04 MODULE_LICENSE("GPL");
05
06 static int intarray[4];
07 static int intarraycount=4;
08 module_param_array(intarray, int, intarraycount, 666);
09 MODULE_PARM_DESC(intarray, "Ein Feld mit bis zu 4 Int");
10
11 static int __init mod_init(void)
12 {
13    printk("intarraycount=%d\n", intarraycount);
14    for(; intarraycount; intarraycount--)
15        printk("%d: %d\n", intarraycount,
16        intarray[intarraycount - 1]);
17    return 0;
18 }
19
20 static void __exit mod_exit(void)
21 {
22 }
23 module_init(mod_init);
24 module_exit(mod_exit);
```

Listing 5: Definition eines String-Parameters

```
01 ...
02 static char string[10];
03 module_param_string( optionname, string,
   sizeof(string), 666 );
04
05 static int __init mod_init(void)
06 {
07    if( string[0] )
08       printk("string: %s\n", string);
09    ...
```

Namen zusammensetzen

Der Name dieses Makros leitet sich aus dem Präfix »param_check_« und dem selbst erteilten Namen des Parameters ab. Damit ergibt sich das in Listing 6 in Zeile 7 dargestellte Makro »param_check_even()«. Jetzt fehlt nur noch der Parameter, den das Makro »module_param()« nach dem bekanntem Schema definiert (Zeile 26). Der Versuch, das Modul einmal mit dem Parameter »para=5« zu laden, schlägt wie erwartet fehl. Ein Aufruf des Moduls mit »para=2« hingegen ist erfolgreich.

Parameter stehen natürlich nicht nur Modulen zur Verfügung. Auch einem in den Kernel gelinkten Treiber lassen sich beim Booten Parameter übergeben. Die Implementierung erfolgt wie beim Modultreiber, nur der Aufruf gestaltet sich etwas anders: Vor dem Parameternamen steht der Name des Moduls. Beispielsweise steht bei »psmouse.proto=bare« der Name »psmouse« für das Mausmodul, »proto« ist der Parametername und »bare« dessen Wert.

Import und Export

Module bekommen aber nicht nur beim Laden Parameter übergeben, sondern interagieren auch mit anderen Modulen. Auf der einen Seite stellen sie Variablen

Lizenz zum Linken

Ob der Quellcode von Modulen, die der Linux-Kernel einbindet, veröffentlicht werden muss oder nicht, ist schon seit jeher eine Streitfrage. Im Grunde geht es darum, ob ein Modul den Kernel nur nutzt oder ihn erweitert. Linus Torvalds erkennt die rechtliche Grauzone an, auch wenn er klar Stellung bezieht. Seiner Meinung nach muss der Code eines jeden nach außen gegebenen Moduls veröffentlicht werden. Support gibt es ohnehin von der Entwicklergemeinde nicht mehr, wenn der Kernel „tainted" ist, wenn sich also im Kernel ein Modul ohne passende Lizenz befindet.

Darüber hinaus können die Entwickler selbst definieren, ob eine Funktion von einem Binary-only-Treiber (also einem Treiber ohne Sourcecode) genutzt werden darf oder nur von einem Treiber mit einer passenden Lizenz (wie die GNU Public License GPL). Dies geschieht über die im Artikel vorgestellten Makros »EXPORT_SYMBOL()« und »EXPORT_SYMBOL_GPL()«. Im Kernel 2.6.3 gibt es über 300 Funktionen, die nur von einem GPL-konformen Treiber genutzt werden können.

Um diese Einschränkung zu umgehen, tricksen Hersteller von Binary-only-Treibern oft: Sie unterteilen ihren Treiber in einen Open-Source- und einen Binary-only-Teil. Der Open-Source-Teil führt ein Mapping von geschützten Funktionen auf neue Funktionen durch, die dann das Makro »EXPORT_SYMBOL()« auch an Binary-only-Treiber durchreicht.

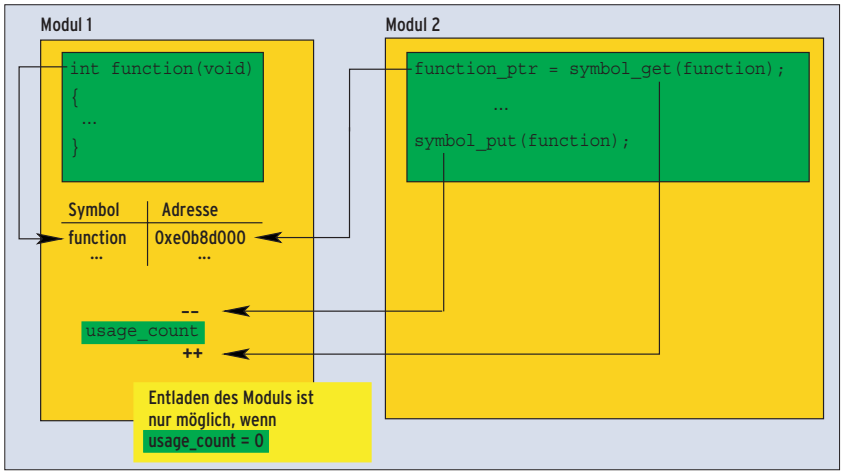


Abbildung 3: Die Makros »symbol_get()« und »symbol_put()« verändern den »usage_count« des exportierenden Moduls.

sowie Funktionen zur Verfügung, auf der anderen nutzen sie die des Kernels und anderer Module.

Das Bereitstellen von Variablen und Funktionen – das Exportieren – gestaltet sich einfach. Eine zu exportierende Variable oder Funktion geben die Makros »EXPORT_SYMBOL()« oder »EXPORT_SYMBOL_GPL()« bekannt. Beim Einsatz von »EXPORT_SYMBOL()« darf jede andere Kernelkomponente das Symbol verwenden. Wird dagegen »EXPORT_SYMBOL_GPL()« gewählt, kann das exportierte Symbol nur von Modulen genutzt werden, die ihrerseits unter einer BSD- oder GPL-Lizenz stehen (siehe Kasten „Lizenz zum Linken“).

Komplexer wird die Intermodul-Kommunikation allerdings, wenn es zwischen dem exportierenden und dem importierenden Modul keine hierarchische Abhängigkeit gibt. Dazu das folgende Szenario: Modul 1 exportiert ein Symbol »good_bye« und möchte zugleich ein Symbol »hello« importieren. Modul 2 wiederum importiert ein Symbol »good_bye« und exportiert »hello«.

Der Versuch, das Modul 1 zu laden, misslingt, da für das Symbol »hello« keine Adresse, also keine Repräsentation im Kernel gefunden werden kann. Modul 2 lässt sich ebenfalls nicht laden, denn bei ihm kann der Kernel das Symbol »good_bye« nicht auflösen – das Modul 1, das die zum Symbol gehörige Funktion oder Variable zur Verfügung stellt, ist ja nicht geladen.

Modul an Modul

Kernel 2.4 besitzt für diesen Fall die Funktionen »inter_module_register()«, »inter_module_unregister()«, »inter_module_get()« und »inter_module_put()«. Bei Linux 2.6 fehlen diese Funktionen, da sie nicht vernünftig gegen Race Conditions abzusichern sind.

An ihre Stelle treten die Makros beziehungsweise Inline-Funktionen »symbol_get()«, »symbol_put()« (siehe Abbildung 2) und »symbol_put_addr()«. Sie sind übrigens vom Kernel mit »EXPORT_SYMBOL_GPL()« exportiert, also können

Listing 6: Einschränkung von Parameterwerten

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/moduleparam.h>
04
05 MODULE_LICENSE("GPL");
06
07 #define param_check_even(name, p)
08
09 int param_set_even(const char *pbuf, struct
   kernel_param *kp)
10 {
11    int number;
12
13    number = simple_strtoul(pbuf, NULL, 0);
14    if(number % 2) // ungerade?
15            return -EINVAL;
16    *(int *)kp->arg = number;
17    return 0;
18 }
19
20 int param_get_even(char *pbuf, struct
   kernel_param *kp)
21 {
22    return sprintf((char *)pbuf, "%d", *(int
   *)kp->arg);
23 }
24
25 static int para;
26 module_param(para, even, 0666);
27
28 static int __init mod_init(void)
29 {
30    printk("para = %d\n", para );
31    return 0;
32 }
33
34 static void __exit mod_exit(void)
35 {
36 }
37 module_init( mod_init );
38 module_exit( mod_exit );
```

Listing 7: »icm.ko«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <asm/io.h>
04
05 MODULE_LICENSE("GPL");
06
07 static char *textbuf = "Hallo Treiber";
08
09 void ton_an(u16 tonwert)
10 {
11    s8 save;
12
13    if( tonwert ) {
14       tonwert = CLOCK_TICK_RATE/tonwert;
15       printk("ton_an(0x%x)\n", tonwert);
16       outb( 0xb6, 0x43 );
17       outb_p(tonwert & 0xff, 0x42);
18       outb((tonwert>>8) & 0xff, 0x42);
19       save = inb( 0x61 );
20       outb(save | 0x03, 0x61);
21    } else {
22       outb(inb_p(0x61) & 0xFC, 0x61);
23    }
24 }
25
26 static int __init buf_init(void)
27 {
28    printk(textbuf);
29    return 0;
30 }
31
32 static void __exit buf_exit(void)
33 {
34    outb(inb_p(0x61) & 0xFC, 0x61);
35 }
36
37 module_init(buf_init);
38 module_exit(buf_exit);
39 EXPORT_SYMBOL_GPL(ton_an);
40 EXPORT_SYMBOL_GPL(textbuf);
```

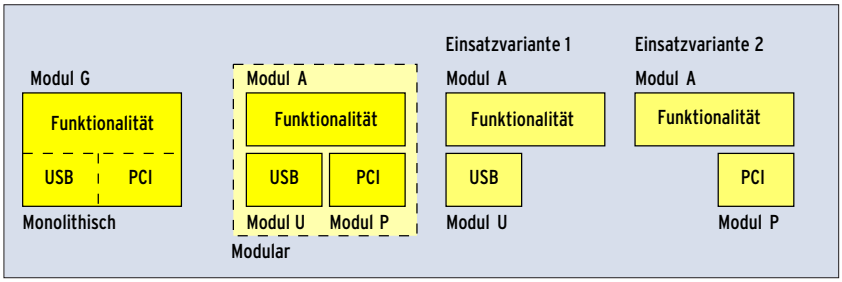


Abbildung 4: Die modulare Lösung trennt USB-, PCI- (Module U und P) und Anwendungsfunktionen (Modul A). Alternativ dazu packt der Programmierer alles auf einmal in Modul G.

nur jene Treiber sie nutzen, die selbst unter einer BSD-Lizenz oder der GNU General Public License stehen.

Symbole finden

Da Symbole nicht beim Laden aufgelöst werden können, geschieht dies zur Laufzeit. Will ein Modul ein Symbol importieren, muss es dazu Speicher (einen Pointer) reservieren, der später die Adresse des Symbols aufnimmt. Diese Adresse fordert das Makro »symbol_get()« an, dessen einziger Parameter der Name des Symbols ist. Das Makro »symbol_get()« baut den Namen des Symbols so zu einem String um, dass innerhalb des Kernels die zugehörige Adresse gesucht werden kann. Ist das Symbol aufgelöst, gibt die Funktion die Adresse zurück, sonst »NULL«.

Darüber hinaus zählt die Funktion »symbol_get()«, wie viele andere Komponenten auf das angeforderte Symbol zugreifen. Solange ein Zählwert größer null vorliegt, lässt sich das exportierende Modul nicht entladen. Ein Aufruf von »symbol_put()« oder alternativ »symbol_put_addr()« dekrementiert den Zähler wieder (Abbildung 3). Zu jedem »symbol_get()« gehört daher ein »symbol_put()« oder ein »symbol_put_addr()«. Das Makro »symbol_put()« besitzt als Parameter den Namen des Symbols, während »symbol_put_addr()« das Symbol über seine Adresse findet.

Die Listing 7 und 8 zeigen die Implementierung zweier Module »icm.ko« und »stacked.ko«. Dabei nutzt »stacked.ko« eine Funktion und eine Variable von »icm.ko«, lässt sich aber auch dann laden, wenn sich das Modul »icm.ko« noch nicht im Speicher befindet. Lädt der Anwender per »insmod« nur das Modul »stacked.ko«, erscheint im Syslog die Meldung, dass der Kernel die beiden Symbole »ton_an« und »textbuf« nicht finden konnte. Lädt er zunächst »icm.ko« und anschließend »stacked.ko«, müsste der PC-Lautsprecher einen Ton von sich geben. Icm basiert nämlich auf dem in [2] vorgestellten Speaker-Treiber. Das Entladen von »stacked.ko« schaltet den Lautsprecher wieder ab. Das Modul »icm.ko« lässt sich erst entladen, wenn auch »stacked.ko« aus dem Speicher entfernt ist.

Listing 8: »stacked.ko«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 extern void ton_an(u16 tonwert);
07 extern char *textbuf;
08
09 static void (*ton_an_function)(u16 var);
10 static char **textbufptr;
11
12 static int __init buf_init(void)
13 {
14    ton_an_function = symbol_get(ton_an);
15    if (ton_an_function)
16       ton_an_function(330);
17    else
18       printk("can't find address of symbol
   \"ton_an\"\n");
19    textbufptr = symbol_get(textbuf);
20    if(textbufptr)
21       printk("content of 0x%p: \"%s\"\n",
   textbufptr, *textbufptr);
22    else
23       printk("can't find address of symbol
   \"textbufptr\"\n");
24    return 0;
25 }
26
27 static void __exit buf_exit(void)
28 {
29    if (ton_an_function) {
30       ton_an_function(0);
31       symbol_put(ton_an);
32       //symbol_put_addr(ton_an_function); //
   alternativ zu symbol_put
33    }
34    if (textbufptr)
35       symbol_put(textbuf);
36 }
37
38 module_init(buf_init);
39 module_exit(buf_exit);
```

Teile und herrsche?

Die hier vorgestellte Intermodul-Kommunikation sollte Entwickler nicht dazu verleiten, jedes Problem in diverse Module zu zerteilen. Das ist nur sinnvoll, wenn damit die Software besser skaliert oder flexibler wird. Lässt sich beispielsweise Hardware über USB oder PCI anschließen, mag es sinnvoll sein, für beide Busse je ein Modul zu erstellen. Ein drittes Modul deckt dann die eigentliche Funktionalität ab und kommuniziert mit Anwendungen (siehe Abbildung 4). Dieses Anwendungsmodul ließe sich mit einem USB- oder einem PCI-Modul betreiben.

Vorschau

Ein Blick in die Verzeichnisse »/lib/« und »/usr/lib/« zeigt, dass es für Applikationsprogrammierer viele Bibliotheken gibt, um ihnen die Entwicklungsarbeit zu erleichtern. Im Kernel gibt es dagegen keine Bibliotheken im klassischen Sinn. Trotzdem bietet der Kernel dem Entwickler eine Reihe hilfreicher Funktionen. Die wichtigsten stellt die nächste Folge der Kern-Technik vor. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 6: Linux-Magazin 1/04, S. 94.

[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 3: Linux-Magazin 10/03, S. 81.

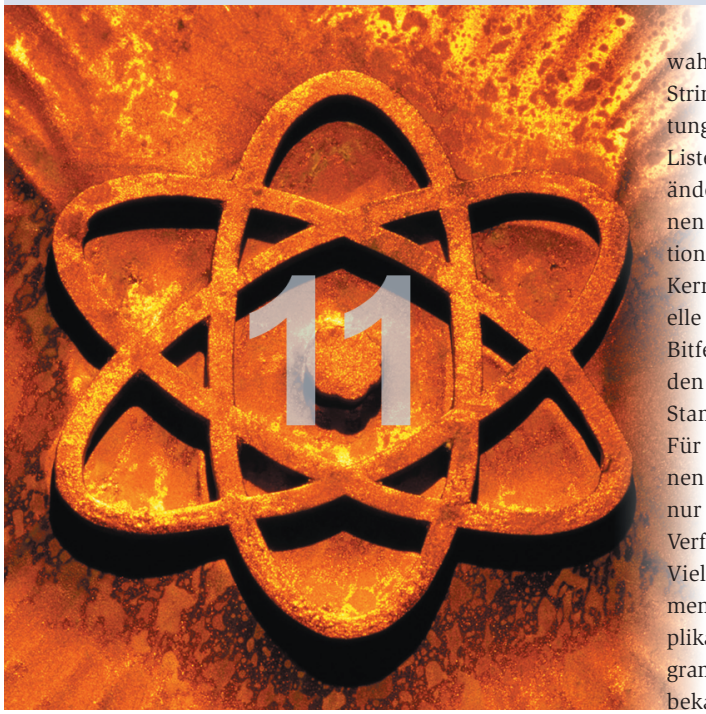
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 11

Kern-Technik

Fertige Bibliotheken nehmen Programmierern Arbeit ab. Auch der Kernel bietet Hilfsfunktionen, obwohl er keinen Zugriff auf Libraries wie die Glibc hat. Von der Stringumwandlung bis zur Listenverwaltung findet sich allerlei Nützliches, das dieser Artikel erklärt und übersichtlich auflistet. Eva-Katharina Kunst, Jürgen Quade



Wer ein Anwendungsprogramm schreibt, muss nicht jede Funktion neu erfinden. Ein- und Ausgabe, Zeichen umwandeln, Strings verarbeiten – solche Aufgaben werden in Funktionen ausgelagert und in Bibliotheken gesammelt. Im Kernel fällt die Unterscheidung zwischen System- und Bibliotheksfunktion schwer (siehe Abbildung 1 und Kasten „Bibliotheks- und Systemfunktionen im Kernel“): Es gibt dort keine Bibliotheken, alles ist Systemcode. Dennoch lassen sich auch im Kernel Systemdienste von Hilfsfunktionen unterscheiden.

Hilfe ohne Bibliotheken

Obwohl der Kernel eine Reihe nützlicher Hilfsfunktionen mitbringt, gibt es keine Übersicht, die beschreibt, wie sie heißen und was sie tun. Reichhaltig ist die Auswahl zum Beispiel für die Stringformatierung, -verarbeitung und -konvertierung. Um Listen anzulegen und zu verändern, gibt es ebenfalls einen umfangreichen Funktionssatz. Daneben bietet der Kernel Funktionen für spezielle Aufgaben, etwa um große Bitfelder zu verarbeiten oder den Taktzyklenzähler (Time Stamp Counter) auszulesen. Für mathematische Operationen stellt der Kernel dagegen nur eine einzige Funktion zur Verfügung.

Viele Hilfsfunktionen stammen aus dem Bereich der Applikationen und sind den Programmierern meist prinzipiell bekannt. Manche Kernelvarianten unterscheiden sich von ihren Verwandten allerdings in Details: zum Beispiel hinsichtlich ihres Namens, aber auch bei den Parametern und ihrer Bedeutung.

Ausgabe über Syslog

Die Funktion »printf()« beispielsweise kommt im Kernel nur in abgewandelter Form vor und heißt dort »printk()«. Wie »printf()« ist sie für die Ausgabe in Ascii zuständig. Der Unterschied besteht aber nicht nur im Namen. So gibt es bei der Kernelversion keine Standardausgabe, wie sie »printf()« von Haus aus verwendet. Die Funktion »printk()« gibt die Daten mit dem Kernel-Log-Daemon »klogd« an den Syslog-Daemon »syslogd«, der sie in eine Logdatei schreibt.

Damit der Syslog-Daemon die Dringlichkeit der Nachricht einstufen kann, bietet »printk()« die Möglichkeit, so genannte Magics im Formatstring unterzubringen. Ein solches Magic besteht aus einer Ziffer von 1 (Emergence-Meldung) bis 7 (Debug-Meldung), die in spitzen Klammern steht. Es muss unbedingt als Erstes im Formatstring auftauchen. Eine Debugmeldung sieht damit beispielsweise so aus:

```
printk("<7>Irq=%d\n", irq);
```

Natürlich muss sich der Programmierer nicht die Nummern merken, denn die Headerdatei »linux/kernel.h« definiert für jede Dringlichkeitsstufe einen symbolischen Namen. Damit lässt sich die Debug-Ausgabe auch folgendermaßen kodieren:

```
printk(KERN_DEBUG "Irq=%d\n", irq);
```

Mit den Makros »pr_debug()« und »pr_info()« geht es noch einfacher (siehe Tabelle 1). Die Routine »pr_debug()« setzt nicht nur »KERN_DEBUG« vor den Formatstring, sondern schaltet die Debug-Meldungen ganz aus, wenn das Symbol »DEBUG« beim Kompilieren nicht definiert ist. Die obige Debug-Ausgabe lautet mit diesem Makro:

```
pr_debug("Irq=%d\n", irq);
```

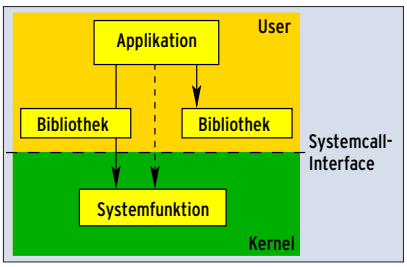


Abbildung 1: Applikationen nutzen Bibliotheksfunktionen und Systemdienste; die Letzteren stellt der Betriebssystemkern zur Verfügung.

- Anzeige -

Bibliotheks- und Systemfunktionen im Kernel

Aus Sicht einer Linux-Anwendung ist die Abgrenzung einfach: Systemfunktionen oder auch Systemcalls arbeitet der Kernel im Auftrag einer Applikation im privilegierten Modus ab. Routinen, die im Kontext der eigenen Applikation ablaufen (User-Kontext), sind Bibliotheks- oder Hilfsfunktionen.

Mathematische Funktionen, beispielsweise »sin()« für Sinus, zählen also zu den Bibliotheksfunktionen. Eine Ausgabe per »write()« ruft dagegen über die Systembibliothek einen Systemcall auf, der im Kernelmodus abläuft, weil er nur so auf die (sonst geschützte) Hardware zugreifen darf (siehe Abbildung 1). Dieser einfachen Definition zufolge kann es im Betriebssystemkern keine Bibliotheksfunktionen geben – schließlich läuft hier jeder Code im Kernel ab.

Dennoch ist eine Differenzierung hinsichtlich des Funktionsablaufs möglich: Routinen, die auf interne Datenstrukturen zurückgreifen und diese gar verändern, sind Systemfunktionen. Routinen, die zustandslos arbeiten, werden hier Hilfsfunktionen genannt. Im Quellcode des Kernels findet sich mit dem Ordner »lib« sogar ein eigenes Verzeichnis dafür.

Die Funktion »pr_info()« unterscheidet sich von »pr_debug()« in der Dringlichkeit: Sie verwendet die nächstdringliche Stufe 6 für »Info«. Außerdem lässt sich ihre Ausgabe nicht durch den Compiler ausschalten.

Sollen Ausgaben nur im Speicher und nicht direkt für den Syslog-Daemon aufbereitet werden, stehen dem Kernelprogrammierer die von der C-Bibliothek bekannten Funktionen »sprintf()«, »snprintf()«, »vsprintf()« und »vsnprintf()« zur Verfügung (siehe Tabelle 2). Deren Parameterliste und Wirkungsweise entsprechen genau den gleichnamigen Funktionen auf Anwendungsebene. Deshalb helfen bei Problemen die Manpages der Bibliotheksfunktionen weiter. Mitt-

Tabelle 1: Ausgabefunktionen in »linux/kernel.h«

Name und Parameter	Aufgabe
int printk (const char *fmt, ...)	Formatierte Ausgabe über Syslog
int pr_debug (const char *fmt, ...)	Formatierte Debugausgabe
int pr_info (const char *fmt, ...)	Formatierte Ausgabe von normalen Infos

Tabelle 2: Stringformatierung in »linux/kernel.h«

Name und Parameter	Aufgabe
int sprintf (char *buf, const char *fmt, ...)	Formatierte Ausgabe in »buf«
int snprintf (char *buf, size_t size, const char *fmt, ...)	Formatierte Ausgabe in »buf«, maximal »size« Zeichen
int vsprintf (char *buf, const char *fmt, va_list args)	Formatierte Ausgabe in »buf«
int vsnprintf (char *buf, size_t size, const char *fmt, va_list args)	Formatierte Ausgabe in »buf«, maximal »size« Zeichen

Tabelle 3: Stringdekodierung in »linux/kernel.h«

Name und Parameter	Aufgabe
int sscanf (const char *buf, const char *fmt, ...)	Konvertiert »buf« gemäß »fmt«
int vsscanf (const char *buf, const char *fmt, va_list args)	Konvertiert »buf« gemäß »fmt«
unsigned long simple_strtoul (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »unsigned long«
long simple_strtol (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »long«
unsigned long long simple_strtoull (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »unsigned long long«
long long simple_strtoll (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »long long«

lerweile dürfte es sich herumgesprochen haben, dass die Varianten mit einem zusätzlichen n wie »snprintf()« und »vsnprintf()« den ursprünglichen Versionen aus Sicherheitsgründen vorzuziehen sind, um das Risiko für Buffer Overflows zu verkleinern. Allerdings verleiten auch diese Funktionen den Programmierer zu Fehlern, beispielsweise dann, wenn er mit mehreren Aufrufen von »snprintf()« in ein und denselben Puffer schreibt.

Strings und Zeichen

Kernel 2.6.4 führt die neuen Funktionen »int scnprintf(char *buf, size_t size, const char *fmt, ...)« und »int vscnprintf(char *buf, size_t size, const char *fmt, va_list args)« ein. Sie unterscheiden sich von »snprintf()« respektive »vsnprintf()« dadurch, dass sie die Anzahl der wirklich geschriebenen Bytes zurückgeben – ohne das abschließende »0«-Zeichen mitzuzählen. Falls die Puffergröße (Parameter »size«) bereits beim Aufruf den Wert 0 besaß, liefern die Funktionen ebenfalls 0.

Die String-Funktionen eignen sich vor allem für die Ein- und Ausgabe Ascii-kodierter Daten über das Proc- und das Sys-Filesystem, siehe [1] und [2]. Ein weiteres Einsatzfeld ist die Dekodierung von Parametern beim Kernelstart oder beim Laden eines Moduls.

Während die vorgestellten Funktionen numerische Werte in Strings umwandeln können, sind diese Routinen für den umgekehrten Weg zuständig: »sscanf()«, »vsscanf()«, »simple_strtoul()«, »simple_strtol()«, »simple_strtoull()« und »simple_strtoll()« (Tabelle 3).

Die Funktion »sscanf(const char* buf, const char* fmt, ...)« dekodiert den String »buf« gemäß der Formatanweisung »fmt« und weist die darin enthaltenen Einzelwerte den übergebenen Variablen zu. Damit unterscheidet sie sich nicht von der gleichnamigen Funktion auf Anwendungsebene. Auch »simple_strtoul()«, »simple_strtoull()«, »simple_strtol()« und »simple_strtoll()« entsprechen den Anwendungsfunktionen ohne den Namensvorsatz »simple_...«.

Im Bereich der Stringverarbeitung finden sich weiterhin die so genannten Ctype-Funktionen »isalnum()«, »isalpha()«, »iscntrl()«, »isdigit()«, »isgraph()«, »islower()«, »isprint()«, »ispunct()«, »isspace()«, »isupper()«, »isxdigit()« und »isascii()« (Tabelle 4). Wie die gleichnamigen Applikationsfunktionen dienen sie dazu, einzelne Zeichen zu klassifizieren. Auch die häufig benötigten Funktionen »tolower()« und »toupper()« brauchen nicht selbst programmiert zu werden. Wer die Zeichen eines Strings einheitlich in Kleinbuchstaben oder in Großbuchstaben haben möchte, kann sich ihrer bedienen. Kernel 2.6 besitzt auch Funktionen, die Strings verglei-

Tabelle 4: Zeichenklassifizierung in »linux/ctype.h«

Name und Parameter	Aufgabe
unsigned char isalnum (unsigned char c)	Prüft, ob »c« ein alphanumerisches Zeichen (Buchstabe oder Ziffer) ist
unsigned char isalpha (unsigned char c)	Prüft, ob »c« ein alphabetisches Zeichen (Buchstabe) ist
unsigned char iscntrl (unsigned char c)	Prüft »c« auf Kontrollzeichen
unsigned char isdigit (unsigned char c)	Prüft, ob »c« zur Klasse der Ziffern gehört
unsigned char isgraph (unsigned char c)	Prüft, ob »c« zur Klasse der druckbaren Zeichen (ohne Space) gehört
unsigned char islower (unsigned char c)	Prüft, ob »c« ein Kleinbuchstabe ist
unsigned char isprint (unsigned char c)	Prüft, ob »c« druckbar ist
unsigned char ispunct (unsigned char c)	Prüft, ob »c« kein Buchstabe und keine Ziffer und dennoch druckbar ist
unsigned char isspace (unsigned char c)	Prüft, ob »c« ein Leerzeichen (Space, Tab, Formfeed) ist
unsigned char isupper (unsigned char c)	Prüft, ob »c« ein Großbuchstabe ist
unsigned char isxdigit (unsigned char c)	Prüft, ob »c« eine Hexadezimalziffer ist
unsigned char isascii (unsigned char c)	Prüft, ob »c« ein Ascii-Zeichen ist

Listing 1: Listenverarbeitung

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 static spinlock_t sl;
07
08 struct my_list {
09    int counter;
10    struct list_head link;
11 };
12 struct list_head rootptr;
13
14 static int show_listfunction_usage( void )
15 {
16    int i;
17    struct my_list *akt_element;
18    struct list_head *loopvar, *tmp;
19
20    for(i=0; i<5; i++) { // Liste anlegen
21       // innerhalb der Critical Section darf
             nicht kmalloc()
22       // verwendet werden!
23       akt_element = kmalloc(sizeof(struct
            my_list), GFP_USER);
24       spin_lock(&sl);
25       akt_element->counter = i;
26       printk("adding: %p- %d\n", akt_element,
                i);
27       list_add(&akt_element->link, &rootptr);
28       spin_unlock(&sl);
29    }
30    spin_lock(&sl);
31    list_for_each_safe(loopvar, tmp, &rootptr)
      { // Liste freigeben
32       i = list_entry(loopvar, struct my_list,
             link)->counter;
33       printk("delete: %p - %d\n", loopvar, i);
34       list_del(loopvar);
35       kfree(list_entry(loopvar,struct my_list,
         link));
36    }
37    spin_unlock(&sl);
38    return 0;
39 }
40
41 static int __init modinit(void)
42 {
43    INIT_LIST_HEAD(&rootptr);
44    show_listfunction_usage();
45    return -EIO;
46 }
47
48 static void __exit modexit(void)
49 {
50    return;
51 }
52
53 module_init(modinit);
54 module_exit(modexit);
```

chen. Tabelle 5 listet von »strstr()« über »strlen()« bis »strncat()« alle auf. Mit aufgeführt sind Funktionen, die nicht auf Strings, sondern allgemein auf Speicherbereiche anwendbar sind wie etwa »memcpy()«, »memset()« & Co.

Listen im Kernel

Auch für komplexere Datenstrukturen bietet der Kernel Unterstützung. So gibt es für doppelt verkettete Listen eine Reihe von Hilfsfunktionen. Der Programmierer definiert in der zu verkettenden Datenstruktur ein Element vom Typ »struct list_head«. Fehlt noch der Listenkopf, den er entweder statisch durch »LIST_HEAD_INIT()« oder dynamisch durch Aufruf von »INIT_LIST_HEAD()« initialisiert. Beide Makros sind in der Headerdatei »linux/list.h« definiert. Die Funktion »list_add()« hängt Elemente in die Liste ein, während »list_del()« sie wieder entfernt.

Für den Zugriff auf die einzelnen Listenelemente bringt der Kernel mehrere Makros mit. Allen ist gemeinsam, dass sie eine Schleife darstellen, die über die Listenelemente iteriert. Die einfachste Variante ist »list_for_each()«.

Der Programmierer übergibt diesem Makro eine Variable vom Typ »struct list_head« (im Weiteren »akt_element«) und den Listenkopf. Innerhalb der Schleife zeigt »akt_element« auf das aktuelle Listenelement. Allerdings ist das ein Zeiger auf die Daten zur Listenverwaltung und nicht unbedingt auf die Anfangsadresse des tatsächlichen Datenobjekts – die liefert das Makro »list_entry()« (siehe Abbildung 2 und Listing 1, Zeile 32).

Für die etwas elegantere Iteration direkt über die Objekte stehen Entry-Varianten zur Verfügung, zum Beispiel »list_for_each_entry_safe()«. Abbildung 3 zeigt, wie man das Makro verwendet und was

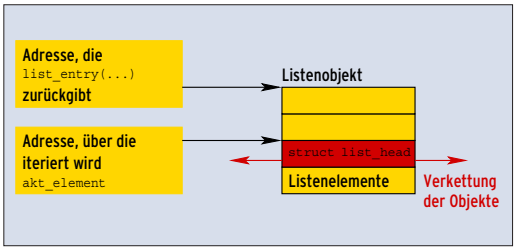


Abbildung 2: Bei einer Iteration über eine Liste zeigt das aktuelle Element »akt_element« häufig auf die Daten zur Listenverwaltung. Das Makro »list_entry()« liefert die Adresse des Anfangs der tatsächlichen Objektdaten.

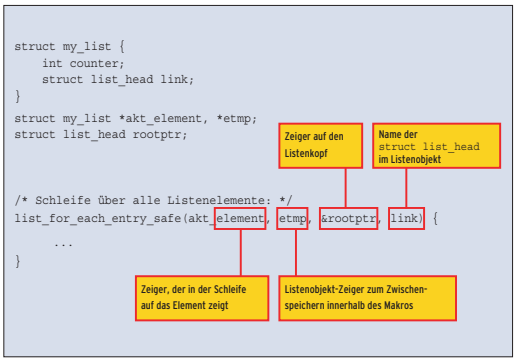


Abbildung 3: Das Makro »list_for_each_entry_safe()« realisiert eine Schleife direkt über die Listenobjekte, statt wie in Abbildung 2 nur über die Verwaltungszeiger.

Tabelle 5: String- und Speicherfunktionen in »linux/string.h«

Name und Parameter	Aufgabe
char *strstr (const char *string, const char *substring)	Sucht »substring« in »string«
int strnicmp (const char *s1, const char *s2, size_t len)	Längen-limitierter Stringvergleich (case insensitive)
char *strncpy (char *dest, const char *src, size_t len)	Kopiert »src« nach »dest«, maximal »len« Zeichen
char *strcpy (char *dest, const char *src)	Kopiert »src« nach »dest«
size_t strlcpy (char *dest, const char *src, size_t size)	Kopiert »src« nach »dest«, maximal »size« Zeichen, immer mit »0« terminiert
char *strncat (char *dest, const char *src, size_t count)	Hängt »src« an »dest« an, kopiert maximal »count« Zeichen, der Ergebnisstring ist Null-terminiert
size_t strlcat (char *dest, const char *src, size_t count)	Wie »strncat()«, gibt aber die neue Stringlänge zurück
size_t strspn (const char *s, const char *accept)	Durchsucht »s« nach den Zeichen aus »accept«
char *strsep (char s, const char *ct)	Zerlegt »s« in Token
char *strpbrk (const char *cs, const char *ct)	Findet das erste Auftreten von »ct« innerhalb von »cs«
size_t strlen (const char *s);	Gibt die Länge von »s« (ohne abschließende Null) zurück
size_t strnlen (const char *s, size_t count)	Berechnet die Länge des mit »count« Zeichen limitierten Strings »s«
int strcmp (const char *cs, const char *ct)	Stringvergleich
int strncmp (const char *cs, const char *ct, size_t count)	Stringvergleich mit maximal »count« Zeichen
char *strchr (const char *s, int c)	Findet das erste Auftreten des Zeichens »c« innerhalb von »s«
char *strrchr (const char *s, int c)	Findet das letzte Auftreten des Zeichens »c« innerhalb von »s«
void *memset (void *s, int c, size_t count)	Füllt den Speicher ab Adresse »s« mit den Zeichen »c«
void bcopy (const char *src, char *dest, int count);	Kopiert »src« nach »dest«
void *memcpy (void *dest, const void *src, size_t count)	Kopiert »src« nach »dest«
void *memmove (void *dest, const void *src, size_t count)	Wie »memcpy()«, allerdings mit überlappenden Speicherbereichen
int memcmp (const void *cs, const void *ct, size_t count)	Vergleicht die Speicherbereiche »cs« und »ct«
void *memscan (void *addr, int c, size_t size)	Sucht das Zeichen »c« innerhalb von »addr« (gibt die Adresse zurück, an der »c« erstmals auftaucht, sonst ein Byte nach dem Bereich)
void *memchr (const void *s, int c, size_t n)	Wie »memscan()«, aber mit anderem Rückgabewert, falls c nicht gefunden wird: »NULL«

die einzelnen Parameter bedeuten. Die Safe-Varianten stellen zudem sicher, dass der aktuelle Listeneintrag nicht aus der Liste entfernt wird, während der Kernel die Schleife durchläuft.

Es gibt noch eine ganze Reihe weiterer Funktionen, um Listen zu bearbeiten. Darunter solche, die zum Beispiel Elemente am Ende der Liste einhängen, zwei Listen zusammenfügen oder eine Liste teilen. Die Init-Funktionsvarianten sorgen zusätzlich noch dafür, dass der Listenkopf richtig, also als leere Liste, initialisiert wird.

Listing 1 demonstriert die Anwendung der Listenfunktionen. Die Funktion »show_listfunction_usage()« durchläuft eine Schleife und hängt mit »list_add()« fünf Elemente in die Liste ein (Zeilen 20 bis 28). Ist die Liste erzeugt, durchläuft das Modul sie mit dem beschriebenen Makro »list_for_each_safe()« (Zeile 31) und löscht die einzelnen Elemente mit »list_del()«. An die Daten kommt das Modul über das Makro »list_entry()« (siehe Abbildung 2).

Diesen Ablauf protokolliert der Syslog-Daemon, dem das Modul jeden Schritt über »printk()« mitteilt (Zeilen 26 und 33). Das dauerhafte Laden des Moduls wird verhindert, indem die Funktion »modinit()« den Fehlercode »-EIO« zurückgibt (Zeile 44). Das vereinfacht den Test, denn das Modul muss so nicht von Hand entladen werden.

Ohne Mathematik

Die Tabellen 1 bis 7 listen die wichtigsten Hilfsfunktionen respektive Makros im Kernel inklusive ihrer Parameter. Darüber hinaus gibt es beispielsweise die »min()«- und »max()«-Makros, die eine umfangreiche Typprüfung der übergebenen Argumente durchführen, oder die bereits erwähnten Funktionen für Bitfelder mit mehr als 32 Bit. Wer allerdings mathematische Funktionen sucht, geht leer aus. Da im Kernel Floating-Point-Operationen verboten sind, fehlen mathematische Funktionen mit Ausnahme der Integer-Wurzel »int_sqrt()«.

Erst kürzlich hat Kernelhacker Andrew Morton auf der Kernel-Mailingliste alle Programmierer dazu aufgefordert, die vorhandenen Hilfsfunktionen zu nutzen. Denn so müssen sie nicht nur selbst weniger Code schreiben, sondern vermeiden auch Fehler – schließlich sind die Hilfsfunktionen im Kernel bereits gründlich getestet.

Vorschau

Ein ladbares Modul produzieren ist die eine Sache, dabei guten Kernelcode zu schreiben eine andere. Selbst wer auf Applikationsseite schon viel programmiert hat, stolpert im Kernel leicht über versteckte Hindernisse. Wer sie kennt, muss nicht alle möglichen Fehler selber machen, die im Kernel oft zum Totalabsturz führen. Deshalb zeigt die Kern-Technik in der nächsten Folge, worauf der Kernelprogrammierer besonders achten sollte. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 6, Sys-Filesystem: Linux-Magazin 01/04, S. 94

[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 7, Proc-Filesystem: Linux-Magazin 02/04, S. 48

[3] Listings zu diesem Artikel: [http://www.linux-magazin.de/Service/Listings/2004/06/Kern-Technik]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Tabelle 6: Zeichenkonvertierung in »linux/ctype.h«

Name und Parameter	Aufgabe
unsigned char tolower (unsigned char c)	Gibt »c« als Kleinbuchstaben zurück
unsigned char toupper (unsigned char c)	Gibt »c« als Großbuchstaben zurück
unsigned char toascii (unsigned char c)	Konvertiert »c« in ein Ascii-Zeichen

Tabelle 7: Listenverarbeitung in »linux/list.h«

Name und Parameter	Aufgabe
void list_add (struct list_head *new, struct list_head *head)	Hängt das Listenelement »new« hinter »head« ein
void list_add_tail (struct list_head *new, struct list_head *head)	Hängt das Listenelement »new« vor »head« ein
void list_del (struct list_head *entry)	Löscht »entry« aus der Liste
void list_del_init (struct list_head *entry)	Löscht »entry« aus der Liste und initialisiert es wieder
void list_move (struct list_head *list, struct list_head *head)	Löscht das Element »list« und hängt es hinter »head« ein
void list_move_tail (struct list_head *list, struct list_head *head)	Löscht das Element »list« und hängt es vor »head« ein
int list_empty (const struct list_head *head)	Gibt »true« zurück, falls kein Element in der Liste »head« eingehängt ist
void list_splice (struct list_head *list, struct list_head *head)	Hängt alle Elemente von »list« in »head« ein
void list_splice_init (struct list_head *list, struct list_head *head)	Hängt alle Elemente von »list« in »head« ein und initialisiert »list«
list_entry (ptr, type, member)	Gibt das zum Listenelement »ptr« gehörige Objekt zurück
list_for_each (pos, head)	Iteriert über die Liste »head«
list_for_each_prev (pos, head)	Iteriert rückwärts über die Liste »head«
list_for_each_safe (pos, n, head)	Iteriert über die Liste »head«, wobei das Element vor dem Löschen geschützt ist
list_for_each_entry (pos, head, member)	Iteriert über die Objekte der Liste »head«
list_for_each_entry_reverse (pos, head, member)	Iteriert rückwärts über die Objekte der Liste »head«
list_for_each_entry_continue (pos, head, member)	Iteriert über die Objekte von »head« ab Listenelement »member«
list_for_each_entry_safe (pos, n, head, member)	Iteriert über »head« und schützt vor dem Löschen des Listeneintrags

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 12

Kern-Technik

Linux ist stabil, weil sich die meisten Programmierer sehr um die Qualität ihres Codes bemühen. Statt hässlicher Quick-and-dirty-Hacks dominiert liebevoll ausgearbeitete Software die zentralen Teile. Im Kernel ist dies besonders wichtig, weil es folgenreiche Kernel-Bugs vermeidet. Eva-Katharina Kunst, Jürgen Quade



Software im Allgemeinen und speziell Systemsoftware muss sicher, effizient, performant, skalierbar, wartbar und anwendbar sein (siehe Tabelle 1). Einige Grundregeln für gutes Design helfen jedem Kernelprogrammierer dabei, in diesem anspruchsvollen Bereich Qualitätssoftware zu entwickeln.

Die erste Regel fordert Vollständigkeit. Ein Treiber muss alle Funktionen seiner Hardware nutzen und das Applikationsinterface komplett unterstützen. Ein Treiber, der nur »read()« und »write()«, aber kein »poll()« oder »select()« kennt, ist nur eingeschränkt tauglich. Gleiches gilt für die Zugriffsmodi. Bis zum 2.4er Kernel gab es nur blockierende und nicht blockierende, Kernel 2.6 kennt zusätzlich asynchrone Zugriffe.

Vollständig sollte auch die Integration neuer Komponenten ausfallen. Im Kernel bedeutet das: Ein Modul muss das neue Gerätemodell unterstützen sowie das Proc-Filesystem, die Hotplug-Funktionen und das Device-Filesystem. Dabei sammelt sich eine Menge Code an, der vollständig dokumentiert sein will.

Ein gutes Stück Code zeichnet sich durch einfache Handhabung aus. Falls es sich parametrisieren lässt, sollte ein Modul die Parameter nach Möglichkeit automatisch bestimmen oder zumindest mit sinnvollen Defaultwerten besetzen. Solche Komfortfunktionen tragen wesentlich zur Akzeptanz einer Softwarekomponente bei.

Beim Entwurf der Interfaces muss der Entwickler dafür sorgen, dass die Benutzer das Interface intuitiv anwenden können. Gut ist es, wenn sie einen Treiber mit »cat« und »echo« testen können. Schlechter wäre, falls sie dafür immer eine spezielle Software benötigten oder viele IO-Controls (per »ioctl()« in einem C-Programm) setzen müssten.

Einfache Handhabung

Um IO-Controls zu vermeiden, ist es oft nötig, ein physikalisches Gerät in mehrere logische Geräte einzuteilen oder viele Attributdateien im Sys- oder Proc-Filesystem zu erstellen. Während der Datenaustausch über das normale Device läuft, sind Konfiguration und Statusabfrage mit Ascii-Kommandos über die Attributdateien möglich. Wer dennoch IO-Controls implementiert, sollte die zugehörigen Kommandos normgerecht kodieren. Der Standard ist in dem Kasten „Kodierung von IO-Controls“ nachzulesen.

Im Kernel-Quellcode ist meist jede Komponente in einer eigenen Quelldatei implementiert. Sie kann auf mehrere tausend Zeilen anwachsen. Den Code auf mehrere Files verteilen ist sinnvoll, wenn es zu einer besseren Skalierbarkeit führt. Der Treiber für eine Hardware beispielsweise, die es in der Ausprägung PCI und USB gibt, lässt sich gut in drei Quelldateien und damit auch drei Modulen realisieren: Ein Modul für die Hardware-Anbindung per PCI, eins für USB und ein auf beide aufsetzendes Modul, das die Funktionalität und die Schnittstelle zur Applikation realisiert.

Wer die Hardware mit USB-Schnittstelle verwendet, belegt keinen Speicher für die PCI-Anbindung. Umgekehrt verschwendet der Besitzer einer PCI-Hardware keine Ressourcen für den USB-Code (siehe Abbildung 1).

Bevor ein angehender Kernelentwickler seine Ideen und Entwürfe in Code umsetzt, ist die Textdatei »Documentation/

Tabelle 1: Qualitätssoftware

Ziel	Programmiertechnik
Sicherheit	Sicherheitsorientierte Programmierung • Fail-Safe-Verhalten • Hilfsfunktionen nutzen
Intuitive Nutzung	Einteilung in logische Geräte • API vollständig unterstützen • Umfassende Systemintegration • Automatische Hardware-Erkennung • IO-Controls sparsam verwenden • Konfiguration optional
Zeitverhalten	Kritische Abschnitte kurz
Skalierbarkeit	Modularisierung
Sonstiges	Dokumentation • Programmierstil

CodingStyle« Pflichtlektüre. Um Quelltexte besser lesen und besser pflegen zu können, geben Linus Torvalds und seine Helfer eine Reihe von Kodierungsrichtlinien vor. Dabei halten sie sich stark an den K&R-Stil (Kernighan und Ritchie, die Entwickler der Programmiersprache C). In diesem Stil wurde bereits das Ur-Unix programmiert. Er zeichnet sich durch hohe Codedichte aus.

Im Kernel setzen die Entwickler aus Effizienz- und Performance-Gründen gelegentlich Konstrukte ein, die einem Anwendungsprogrammierer die Haare zu Berge stehen lassen. Ein richtig verwendetes »goto« spart Code, Laufzeit und erhöht sogar die Verständlichkeit. Auf Rekursion ist dagegen zu verzichten – dem Kernel steht nur ein eingeschränkter Stack zur Verfügung. Das erklärt auch, warum Kernelfunktionen nur wenig lokale Daten reservieren dürfen.

Makros wie »likely()« und »unlikely()« helfen dem Compiler bei der Optimierung. Sie umrahmen die If-Bedingung:

```
if (likely(len>1)) {
  /* tritt wahrscheinlich ein */
} else {
  /* eher unwahrscheinlich */
}
```

So weiß der Compiler, dass die Bedingung fast immer erfüllt ist, und optimiert den Code dafür. Der Kasten „Do-While-0-Makros“ erklärt ein weiteres abenteuerliches Konstrukt.

Besonders im Kernel müssen Programmierer sicherheitsbewusst arbeiten. Gegen Puffer-Überläufe hilft es, die Funktion »snprintf()« anstelle von »sprintf()« zu verwenden und »strlcpy()« statt »strcpy()«. Besonders sorgfältig ist der Code auf kritische Abschnitte zu überprüfen [1]; jeder Zugriff auf eine globale Variable ist ein Kandidat dafür. Auch die Synchronisation zwischen Codeteilen gilt es zu prüfen und gegebenenfalls zu schützen. Linux gibt dem Entwickler dazu Spinlocks, Semaphore und Completion-Objekte an die Hand. Wichtig ist auch, dass ein Modul den Wertebereich der ihm übergebenen Parameter kontrolliert.

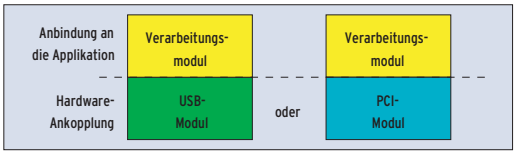


Abbildung 1: Gibt es ein Gerät in mehreren Varianten (hier mit USB- und PCI-Schnittstelle), dann ist ein modularer Gerätetreiber zu bevorzugen. Das Verarbeitungsmodul ist immer erforderlich, aber USB- und PCI-Modul sind austauschbar.

Sicherheit im Treiber

Fehler vermeidet, wer getestete Codeteile verwendet. Bietet der Kernel eine Standardfunktion für eine Aufgabe, sollte ein Treiber sie auch nutzen. Manchmal entstehen dabei aber Abhängigkeiten, die den Vorteil aufheben. Verwendet ein Treiber beispielsweise eine im ACPI-Subsystem definierte Hilfsfunktion, dann lässt er sich nur übersetzen, wenn auch ACPI konfiguriert ist.

Ein Modul muss sich auch sicherheitsbewusst verhalten. Dazu ist bei einem Treiber beispielsweise die Funktion »driver_close()« zu implementieren. Sobald sich die den Treiber nutzende Applikation beendet (etwa durch einen Fehler), ruft der Kernel »driver_close()« auf. Diese Funktion sollte Fail-Safe-Verhalten implementieren, also die vom Treiber bediente Hardware in einen sicheren Zustand bringen. Ähnlich kann ein Watchdog zu mehr Sicherheit führen: Er schaltet sich ein, sobald wichtige Aktionen nicht rechtzeitig erfolgen.

Eigene Daten halten

In Kernelcode zu speichernde Daten lassen sich in vier Kategorien einteilen:

- Treiberspezifische Daten, zum Beispiel der Treibername.
- Daten zur Hardware, beispielsweise Interruptnummer oder IO-Bereiche.
- Daten, die zu einem logischen Gerät gehören, etwa die Anzahl aktuell auf das Gerät zugreifender Instanzen.
- Daten, die zu einer Treiberinstanz gehören. Eine Instanz entsteht, wenn eine Applikation »open()« ausführt. Der Kernel entfernt die Instanz, wenn die Anwendung »close()« aufruft.

Zu jeder Kategorie passen andere Datenstrukturen. Nutzt ein Modul geeignete Strukturen, dann muss es sich nicht selbst um die Verwaltung der Datenobjekte kümmern. Viele Datenstrukturen des Kernels enthalten bereits Speicherplatz für einen Zeiger, den eine selbst erstellte Komponente nutzen darf. Im »struct file«-Objekt der Treiberinstanz heißt dieser Zeiger beispielsweise »void *private_data«. Ruft ein Rechenprozess »open()« auf, reserviert die »driver_open()«-Funktion des Moduls Speicher

Kodierung von IO-Controls

Wer IO-Controls verwendet, muss die einzelnen Kommandos, deren Parameter und ihr Verhalten (besonders im Fehlerfall) spezifizieren und dokumentieren. Für die Kodierung der IO-Control-Kommandos haben die Linux-Entwickler einen Standard festgelegt und Makros definiert, siehe »asm/ioctl.h«. Zwar ist kein Entwickler gezwungen sich daran zu halten, zu empfehlen ist es aber: Der Standard sorgt für eindeutige, unterscheidbare Kommandos innerhalb des Kernels und vereinfacht das Debugging.

Ein IO-Control-Kommando besteht aus 32 Bit (4 Byte). Dieses 32-Bit-Wort ist in vier Felder aufgeteilt (siehe Abbildung 2). Die Nummer kennzeichnet das Kommando. Mit 8 Bit lassen sich innerhalb eines Treibers 256 unterschiedliche Kommandos realisieren. Für den Typ stehen ebenfalls 8 Bit zur Verfügung. Er soll helfen IO-Controls systemweit eindeutig zu halten. Die leider veraltete Datei »Documentation/ioctl-numbers.txt« listet die bereits vergebenen Typen. Üblich ist es, den Anfangsbuchstaben des Modulnamens als Basis für diese Zahl zu verwenden.

Im IO-Control-Kommando ist auch die Länge der maximal zu transferierenden Daten kodiert. Hierzu stehen 14 Bit zur Verfügung. Null bedeutet, dass mehr als 16384 Bytes (2^{14}) übertragen werden. Für die Transferrichtung (2 Bit) gibt es vier Möglichkeiten:

- Keine Daten übertragen, der Wert ist im Makro »_IOC_NONE« abgelegt
- Daten nur lesen, Makro »_IOC_READ«
- Daten nur schreiben, Makro »_IOC_WRITE«
- Daten lesen und schreiben: »_IOC_READ | _IOC_WRITE«

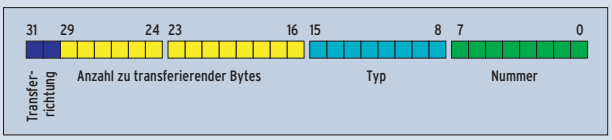


Abbildung 2: Standardkonforme IO-Control-Kommandos enthalten vier Informationen: Transferrichtung, Anzahl zu übertragender Bytes, Typ und Nummer. Mit dem Makro »_IOC(dir, type, nr, size)« vereinfacht Linux die Handhabung der IO-Controls.

für eigene Daten dieser Instanz. Die Adresse hängt es in das File-Objekt der Treiberinstanz ein (Listing 1).
Bei den darauf folgenden Zugriffen (zum Beispiel »read()«) kann das Modul per »instanz->private_data« auf seine Daten zugreifen. Der Kernel übergibt jeder Funktion einen Zeiger auf das Instanz-Objekt. Das Modul muss lediglich dafür sorgen, dass es den mit »kmalloc« reservierten Speicher später wieder freigibt.

Sparsamkeit

Wer sich um Skalierbarkeit sorgt, sollte optionale Codeteile mit »#define« ausmaskieren, beispielsweise Code für das Proc-Filesystem, das PCI-Subsystem, das Device-Filesystem oder das PNP-Subsystem (Plug&Play). Die Funktionsaufrufe dieser Kernelkomponenten sind so programmiert, dass sich Code auch kompilieren lässt, wenn die zugehörige Komponente fehlt (in der Kernelkonfiguration ausgelassen wurde). Wie Abbildung 3 verdeutlicht, spart das Ausmaskieren Code und damit Laufzeit.
Kernelcode bestimmt das Zeitverhalten des Betriebssystems. Unnötige Schleifen und lange kritische Abschnitte sind zu vermeiden. Interrupt-Service-Routinen müssen kurz sein, längere Berechnungen gehören in ein eigenes Tasklet [2]. Besseres Zeitverhalten erreichen auch Programmierer, die einen kurzen kritischen Abschnitt per Spinlock schützen statt mit einem Semaphor (Mutex) [1]. Zudem sollten sie für unterschiedliche Wartebedingungen auch unterschiedliche Waitqueues einsetzen. Das vermeidet unnötiges Aufwecken von Rechenprozessen und spart Rechenzeit.
Software muss immer exakt erledigen, was der Anwender von ihr erwartet, und dabei fehlerfrei arbeiten. Sie darf keine unerwarteten Aktionen ausführen. Wer diese Grundregeln befolgt, wird nicht – wie bei Excel geschehen [3] – einen funktionierenden Flugsimulator in seinem Code verstecken. *(fjl)* ■

Listing 1: Private Daten

```
01 int driver_open (
02   struct inode *geraetedatei,
03   struct file *instanz)
04 {
05   struct p_instanz *priv;
06
07   priv = (struct p_instanz *) kmalloc
08     (sizeof(struct p_instanz), GFP_USER);
09   [...]
10   instanz->private_data = (void *) priv;
11   [...]
12 }
```

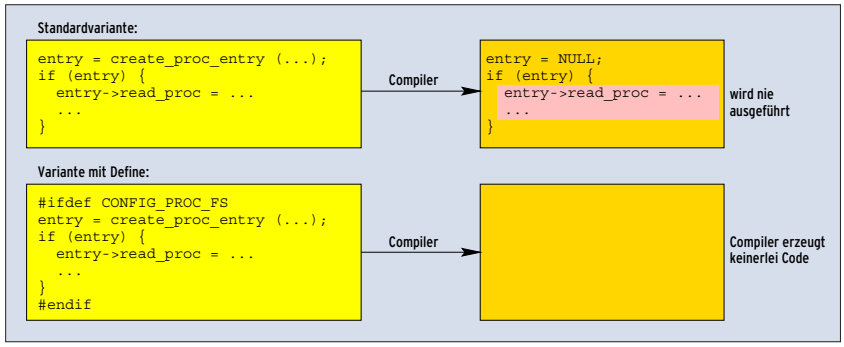


Abbildung 3: Das Ausmaskieren optionaler Teile per »#define« ist zwar nicht zwingend nötig, spart aber Code. Die Unterstützung für das Proc-Filesystem ist beide Male im Kernel deaktiviert. Im oberen Fall muss der Compiler dennoch Code erzeugen. Die Define-Variante vermeidet dagegen unnötige Funktionsaufrufe.

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik, Folge 5“: Linux-Magazin 12/03, S. 82
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik, Folge 4“: Linux-Magazin 11/03, S. 96
[3] In Excel 97 ist ein Flugsimulator versteckt: [http://www.mogelpower.de/easter/eggs/egg.php?id=58]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Do-While-0-Makros

Im Kernel tauchen häufig Makros auf, die in einer »do {...} while(0)«-Schleife verpackt sind. Auffällig daran: Die Bedingung ist fest vorgegeben und kann nie wahr sein, der Schleifenblock läuft also genau einmal ab. [http://www.kernelnewbies.org/faq/#dowhile] erklärt den Trick: So lassen sich Makros, die aus mehreren Anweisungen bestehen, gefahrlos in einer If-Anweisung verwenden. Einfacher wäre es, die Kommandos als Block zusammenzufassen:

```
#define FOO(x) { \
  pr_debug("error on input %d\n", x); \
  return -1; }
```

Das führt zu Fehlern, wenn ein Programmierer das Makro in einer If-Else-Anweisung nutzt:

```
if (fd<0)
  FOO(fd);
else
  something_else(fd);
```

Der C-Präprozessor ersetzt »FOO(fd)« durch das Makro, dabei entsteht aber ungültiger Code:

```
if (fd<0)
  {
  pr_debug("error on input %d\n", fd);
  return -1; };
else
  something_else(fd);
```

Problematisch ist das Semikolon nach dem eingefügten Block. Der Compiler interpretiert es als leeres Statement und koppelt den Else-Zweig ab. Der Programmierer hätte nach »FOO(fd)« kein Semikolon schreiben dürfen. Dazu müsste er aber wissen, wie »FOO()« implementiert ist – wäre es eine Funktion oder ein Makro, das nur eine Anweisung einfügt, wäre das Semikolon wieder erforderlich.

Im Dienste der Codequalität

Eine Do-While-0-Schleife umgeht das Problem recht elegant. Der Präprozessor fügt damit genau eine Anweisung ein, der Programmierer braucht sich daher nicht um die Interna des Makros zu sorgen:

```
#define FOO(x) do { \
    pr_debug("error on input %d\n", x); \
    return -1; \
  } while(0)
```

Eingesetzt in das If-Else-Statement ergibt sich fehlerfreier Code:

```
if (fd<0)
  do {
    pr_debug("error on input %d\n", fd);
    return -1;
  } while(0);
else
  something_else(fd);
```

Auf die Laufzeit hat dieses Konstrukt keinen negativen Einfluss, da gute Compiler die unnötige Do-While-Schleife wegoptimieren.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 13

Kern-Technik

Systemaufrufe bilden die Schnittstelle zwischen Userspace und Kernel. Dieser Artikel zeigt, wie sie funktionieren und wie man sogar eigene Systemcalls implementiert. Eva-Katharina Kunst, Jürgen Quade



Das Betriebssystem verwaltet Ressourcen wie Rechenzeit, Speicher oder Geräte und stellt Dienste dafür zur Verfügung: Will eine Anwendung auf Hardware zugreifen, tut sie das nicht direkt, sondern vermittelt durch den Kernel, der im Prozessormodus mit der dafür nötigen Priorität läuft. Für die meisten Zwecke bringt das Betriebssystem passende Interface-Funktionen mit, die so genannten Systemaufrufe, die einen definierten Übergang vom User- in den Kernelmodus darstellen. Mehr als 280 Systemcalls finden sich in Kernel 2.6.6.

Durchnummeriert

Jeder Systemcall besitzt eine eindeutige Nummer, die ihm in der Headerdatei »asm/unistd.h« (siehe Listing 1) zugeordnet wird. Der Systemcall »fork«, der einen neuen Prozess erzeugt, trägt beispielsweise die Nummer 2, »write« entspricht 4 und »gettimeofday« der 78.

Normalerweise ruft die C-Bibliothek Glibc hinter den Kulissen solche Systemcalls auf, wenn eine Anwendung entsprechende Funktionen benutzt. Um einen Systemcall direkt aufzurufen, muss manuell ein Software-Interrupt ausgelöst werden. Da die Programmiersprache C hierzu keine Funktion anbietet, braucht der Entwickler dafür Assembler.

Es gibt mehrere Software-Interrupts. Für Systemcalls kommt traditionell die Nummer 0x80 zum Einsatz, auch wenn es noch andere Wege gibt, um in den Kernel zu wechseln, siehe Kasten „Intel kann es schneller“. Der Interrupt löst den Übergang in den privilegierten Kernelmodus aus. Der Kernel selbst erwartet die Nummer, die den auszuführenden Dienst kennzeichnet (also beispielsweise 4 für »write«), in einem Register, auf einer x86-Plattform im EAX. Die zum Systemcall gehörigen Parameter werden ebenfalls in Registern abgelegt. Abbildung 1 zeigt die Register eines x86-Prozessors, weitere Informationen finden sich unter [1].

Parameter in Registern

Listing 2 ruft in Assembler den Systemcall »write« auf. Der Filedeskriptor, die Adresse für die Ausgabe und die Anzahl der schreibende Bytes liegen nacheinander in den Registern EBX, ECX und EDX. Um aus dem Assembler-Code ein Programm zu erzeugen, reicht der Aufruf von »make hello« aus. Zum selben Ergebnis führt »gcc -o hello hello.S«.

In den seltensten Fällen schreibt der Anwendungsentwickler seinen Assembler-Code selbst. Für die wichtigsten Systemcalls gibt es in der Standard-C-Library entsprechende Funktionen, die er nur aufzurufen braucht. Aber auch wenn der Systemcall nicht über eine Bibliotheksfunktion zur Verfügung steht, ist Assembler-Programmierung kein Muss. Vielmehr stellt der Kernel einen Satz Makros bereit und die Standardbibliothek hat eine Systemcall-Funktion.

Die »_syscall*X*()«-Makros versehen einen Systemcall mit einem Funktionsnamen, den der Programmierer später verwenden kann. Das »*X*« steht dabei für die Anzahl der Parameter. Um beispielsweise den Systemcall »write« mit seinen drei Parametern zu nutzen, kommt das Makro »_syscall3()« zum Einsatz. Listing 4 zeigt den Aufruf.

Da die »_syscall*X*()«-Makros intern auf die globale Variable »errno« zugreifen, muss der entsprechende Header eingebunden sein (Zeile 2). Außerdem ist sicherzustellen, dass der Compiler die

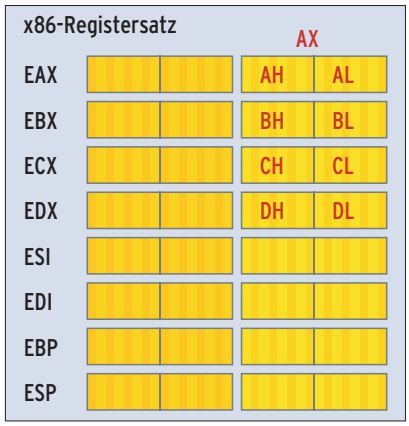


Abbildung 1: Die Parameterübergabe zwischen User- und Kernelspace läuft über die Register des x86-Prozessors: Im EAX steht die Nummer des Syscall, die anderen nehmen dessen Parameter auf.

Headerdateien des Kernels 2.6 verwendet, zum Beispiel durch den Aufruf:

```
cc -Wall -I/usr/src/linux-2.6.6/include -o ➥
syscallX syscallX.c
```

Der erste Parameter des »_syscall*X*()«-Makros steht für den Typ des Systemcall-Rückgabewerts. Der zweite Parameter entspricht dem Namen des Systemcall, der als solcher in der Datei »asm/unistd .h« festgelegt ist. Der dortige Namensvorsatz »__NR_« ist allerdings wegzulassen; aus »__NR_write« wird damit »write«. Danach folgt – immer paarweise – die Beschreibung der einzelnen Parameter: zuerst der Typ, dann der Name. Zum Einsatz der Makros ist die Datei »asm/unistd.h« einzubinden.

Funktion statt Makro

Linus Torvalds & Co. sind allerdings nicht sonderlich glücklich über die Verwendung der Makros. Tatsächlich denken die Kernelentwickler darüber nach, sie den Anwendungsprogrammierern in Zukunft vorzuenthalten. Stattdessen empfehlen sie, lieber die Funktion »syscall()« zu verwenden, die in »unistd.h« deklariert ist. Wie man sie benutzt, zeigt Listing 5.

Die »syscall«-Funktion und damit auch das Makro »_syscallX« verarbeiten den Rückgabewert des Systemcall weiter. Bewegt sich der Wert zwischen -1 und -125, belegt die Funktion damit die »errno«-Variable und gibt »-1« zurück. Im Kernel selbst gibt es für jeden Systemcall eine System Call Service Routine (Systemcall-Funktion), die den Auftrag weiterbearbeitet. Wie der Kernel die Systemcall-Parameter verarbeitet und die Serviceroutine aufruft, ist größtenteils in Assembler programmmiert und plattformabhängig. Die Adressen der Systemcall-Funktionen enthält die Funktionstabelle »syscall_table« gemäß den Systemcall-Nummern. Die entsprechende Tabelle ist für die x86-Architektur in der Datei »arch/i386/kernel/entry.S« im Kernel-Quellcode definiert.

Implementierung im Kernel

Die Bearbeitungsfunktion selbst ist plattformunabhängig und im Kernel als einfache Routine implementierbar. Vor der Funktionsdefinition muss nur das Schlüsselwort »asmlinkage« stehen. Es legt abhängig von der Plattform eine passende Aufrufkonvention fest.

Die Namen der Systemcall-Funktionen beginnen in der Regel mit dem Vorsatz »sys_«. Somit lautet die Funktion, die »write« implementiert, »sys_write()«. Systemcalls liefern grundsätzlich ein Ergebnis vom Typ »long«. Die Konvention dabei ist: 0 oder ein positiver Wert besagen, dass kein Fehler aufgetreten ist. Ein negativer Wert dagegen zeigt den Misserfolg der Operation an. Einige Systemcalls implementiert die Datei »kernel/sys.c«, »sys_write()« findet sich unter »fs/read_write.c«.

Im Kernel laufen die Systemcalls im Kontext des zugehörigen Prozesses ab. Das bedeutet, dass der Systemcall den Rechenprozess schlafen legen kann und dass Daten zwischen User- und Kernelspace über die Funktionen »copy_from

Intel kann es schneller

Einen Software-Interrupt über den Befehl »int« aufrufen ist auf einem x86-Prozessor vergleichsweise Ressourcen-intensiv. Bereits seit dem Pentium II hat Intel in den Befehlssatz seiner Prozessoren daher das neue Befehlspaar »sysenter« und »sysexit« aufgenommen. Mit ihm führt der Kernel Systemcalls erheblich schneller aus.

Allerdings ist ein Wechsel vom bisherigen »int«-Mechanismus zum »sysenter«-Mechanismus nicht ohne weiteres möglich. Da es sich um die Schnittstelle zwischen User- und Kernelspace handelt, müssen sowohl die Anwendungen als auch der Kernel angepasst werden. Die größte Schwierigkeit ist aber, dass es »sysenter« nicht bei allen x86-kompatiblen Prozessoren gibt. Deshalb führt Kernel 2.6 die so genannte »vsyscall«-Page ein. In den Adressraum einer Applikation wird eine Speicherseite eingeblendet, in der abhängig vom Prozessor entweder der Code für den Software-Interrupt oder für »sysenter« zu finden ist.

Den bisherigen Aufruf »int $0x80« ersetzt der Programmierer durch einen Unterprogrammaufruf an die Adresse »0xfffe400«. Der Beispielaufruf aus Listing 2 ist mit den entsprechenden Änderungen in Listing 3 dargestellt. Wird der Code mit diesem »make hello2« übersetzt und aufgerufen, muss ebenfalls „Hello World“ auf dem Bildschirm erscheinen.

Listing 1: Auszug aus »asm/unistd.h«

```
01 #ifndef _ASM_I386_UNISTD_H_
02 #define _ASM_I386_UNISTD_H_
03
04 /*
05  * This file contains the system call numbers.
06  */
07
08 #define __NR_restart_syscall      0
09 #define __NR_exit                 1
10 #define __NR_fork                 2
11 #define __NR_read                 3
12 #define __NR_write                4
13 #define __NR_open                 5
14 #define __NR_close                6
15 ...
```

Listing 2: »hello.S«

```
01 .text
02 .globl main
03 main:
04   movl $4,%eax        ; //Code fuer "write" systemcall
05   movl $1,%ebx        ; //File descriptor fd (1=stdout)
06   movl $message,%ecx  ; //Adresse des Textes (buffer)
07   movl $12,%edx       ; //Laenge des auszugebenden Textes
08   int $0x80           ; //SW-Interrupt, Auftrag an das BS
09   ret
10 .data
11 message:
12   .ascii "Hello World\n"
```

Listing 3: »hello2.S«

```
01 .text
02 .globl main
03 main:
04   movl $4,%eax        ; //Code fuer "write" systemcall
05   movl $1,%ebx        ; //File descriptor fd (1=stdout)
06   movl $message,%ecx  ; //Adresse des Textes (buffer)
07   movl $12,%edx       ; //Laenge des auszugebenden Textes
08   call 0xffffe400     ; //Auftrag an das BS
09   ret
10 .data
11 message:
12   .ascii "Hello World\n"
```

Listing 4: Syscall mit »_syscallX«-Makro

```
01 #include <asm/unistd.h>
02 #include <errno.h>
03
04 _syscall3( int, write, int, fd, char *, buffer, int,
   size );
05
06 int main( int argc, char **argv )
07 {
08         write( 1, "hello world\n", 13 );
09         return 0;
10 }
```

Listing 5: Systemcall über »syscall()«

```
01 #include <unistd.h>
02 #include <asm/unistd.h>
03
04 int main( int argc, char **argv )
05 {
06         syscall( __NR_write, 1, "Hello World\n", 13 );
07         return 0;
08 }
```

Listing 6: »sys_udelay()«

```
01 #include <linux/syscalls.h>
02 #include <linux/jiffies.h>
03 #include <asm/delay.h>
04
05 asmlinkage long sys_udelay( int museconds )
06 {
07         printk("sys_udelay( %d )\n", museconds );
08         if( (museconds < 0) || (museconds > 10000) )
09                 return -EINVAL;
10         udelay( museconds );
11         return 0;
12 }
```

Listing 7: Änderung in »kernel/Makefile«

```
01 #
02 # Makefile for the linux kernel.
03 #
04
05 obj-y     = sched.o fork.o exec_domain.o panic.o printk.o
   profile.o \
06             exit.o itimer.o time.o softirq.o resource.o \
07             sysctl.o capability.o ptrace.o timer.o user.o \
08             signal.o sys.o kmod.o workqueue.o pid.o \
09             rcupdate.o intermodule.o extable.o params.o
   posix-timers.o \
10             kthread.o sysudelay.o
11
12 obj-$(CONFIG_FUTEX) += futex.o
13 ...
```

Listing 8: »include/linux/syscalls.h«

```
01 ...
02 asmlinkage long sys_uselib(const char __user
   *library);
03 asmlinkage long sys_ni_syscall(void);
04 asmlinkage long sys_ni_syscall(void);
05
06 asmlinkage long sys_udelay(int musec);
07
08 #endif
```

Listing 9: »arch/i386/kernel/entry.S«

```
01          ...
02          .long sys_mq_timedsend
03          .long sys_mq_timedreceive       /* 280 */
04          .long sys_mq_notify
05          .long sys_mq_getsetattr
06          .long sys_udelay
07
08 syscall_table_size=(.-sys_call_table)
```

_user()« und »copy_to_user()« (siehe [2]) ausgetauscht werden.

Wer Erfahrung in der Programmierung mit C, mit dem Kompilieren und Installieren eines Linux-Kernels hat (siehe [3]), kann einen Systemcall in nur wenigen Schritten selbst programmieren. Als Kernelmodul ist das allerdings nicht möglich – zumindest nicht ohne besondere Hacks. Ein Systemcall wird statisch im Kernel implementiert.

Systemcall selbst gestrickt

Das hier aufgeführte Beispiel realisiert einen Systemcall »udelay«, der in einem Busy Loop zwischen 0 und 10 Millisekunden Rechenzeit beansprucht, die der Benutzer in Mikrosekunden angibt. Als Basis dient ein Kernel 2.6.6, der in dem Verzeichnis »/usr/src/linux-2.6.6« installiert ist. Die im Folgenden aufgeführten Pfad- und Dateinamen beginnen – falls nicht anders angegeben – in diesem Verzeichnis. Der komplette Dateiname von »include/linux/unistd.h« lautet damit »/usr/src/linux-2.6.6/include/linux/uni-std.h«.

Abbildung 2 zeigt die neun Schritte zum eigenen Systemcall: Zuerst legt der Programmierer in den Kernelquellen eine neue Datei an (Schritt 1), zum Beispiel im Verzeichnis »kernel«. Hier heißt sie »sysudelay.c« und besteht im Wesentlichen aus der Systemcall-Serviceroutine »sys_udelay()« (Schritt 2). Der zugehörige Code ist in Listing 6 zu sehen. Abgesehen vom Schlüsselwort »asmlinkage«, dem Rückgabewert vom Typ »long« und der maximalen Parameteranzahl von fünf gibt es hier nichts Besonderes zu beachten. Der Rückgabewert wird später im Userspace (im Makro »_syscall*X*()« respektive in der Funktion »syscall()«) in die globale Fehlervariable »errno« umgesetzt.

Als Headerdatei ist normalerweise nur »linux/syscalls.h« zu inkludieren. In diesem Fall muss zusätzlich »asm/delay.h« eingebunden werden, die das Makro »udelay()« definiert. Im dritten Schritt wird das Makefile angepasst: Die Variable »obj_y« erhält zusätzlich den Namen des neuen Objektfile. Listing 7 zeigt die Änderungen rot markiert. Schritt 4 erweitert dann die Headerdatei »include/linux/syscalls.h« um den Prototyp der neuen Systemcall-Servicefunktion, siehe Listing 8.

Schritt 1:	Im Linux-Quellcode eine neue Datei anlegen
Schritt 2:	In der neuen Datei den Systemcall implementieren
Schritt 3:	Makefile anpassen
Schritt 4:	Funktionsprototyp in <include/linux/syscalls.h> eintragen
Schritt 5:	Neuen Systemcall in die »sys_call_table« eintragen
Schritt 6:	Einführung der neuen Systemcall-Nummer in <include/asm/unistd.h>
Schritt 7:	Kernel erzeugen
Schritt 8:	Kernel installieren
Schritt 9:	Test

Abbildung 2: In neun einfachen Schritten lassen sich eigene Systemcalls im Kernel implementieren und benutzen.

Header ändern

Die Adresse des neuen Systemaufrufs »sys_udelay« muss jetzt noch ans Ende der Syscall-Tabelle »sys_call_table« angehängt werden (Schritt 5). Dieser Schritt ist plattformabhängig, wie auch der folgende. Auf der »i386« Architektur befindet sich die »sys_call_table« beispielsweise in der Assembler-Datei »arch/i386/kernel/entry.S« (siehe Listing 9). Für die PowerPC-Plattform heißt die entsprechende Datei »arch/ppc/kernel/misc.S«. Wer einen AMD 64 unterstützen möchte, trägt die Zeile in die Datei »include/asm-x86_64/unistd.h« ein.

Der neue Syscall bekommt in der Datei »unistd.h« eine Nummer zugeordnet – das gilt für alle Plattformen. Der Beispiel-Systemcall erhält gemäß Konvention die Bezeichnung »__NR_udelay«. Ein solcher Eintrag muss für jede unterstützte Plattform bestehen, auf dem PC in der Datei »include/asm-i386/unistd.h«. Der Systemcall bekommt auf dieser Plattform und dem Kernel 2.6.6 die Nummer 283. Das Define »__NR_syscalls«, das die Anzahl der Systemcalls angibt, wird um eins erhöht und erhält somit den Wert 284 (siehe Listing 10).

Jetzt kann der Kernel neu kompiliert (Schritt 7) und installiert (Schritt 8) werden.

Nach einem Reboot steht der neue Systemcall zum Test bereit. Dazu dient das in Listing 11 vorgestellte Programm. Es ruft »sys_udelay()« zweimal auf, davon einmal mit ungültigem Parameter.
Nach dem Übersetzen und Starten des Programms, gibt die Syscall-Routine beim ersten Aufruf von »syscall()« den Wert »0« zurück, beim zweiten Mal »-1« (vergleiche Listing 6). In diesem Fall setzt sie die globale Variable »errno« zusätzlich auf »-EINVAL«. Die Ausgabe des Testprogramms lautet damit:

```
# ./testsys
udelay(9000)=0
udelay: Invalid argument
udelay(10001)=-1
```

Wie geht's weiter?

Nur selten ist es erforderlich, einen eigenen Systemcall zu implementieren. Für die meisten Anwendungen genügen jene Systemaufrufe voll und ganz, die der Kernel von Haus aus mitbringt. Normalerweise benutzen nur Anwendungsprogramme oder Libraries Systemaufrufe. Manchmal möchte allerdings auch der Kernelhacker gerne einen Dienst des Betriebssystem-Kerns verwenden. Wie man das macht, zeigt die nächste Folge der Kern-Technik. *(ofr)*

Infos

[1] Informationen zu den x86-Prozessoren: [http://developer.intel.com/design/Pentium4/documentation.htm]
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 2: Linux-Magazin 9/03, S. 86
[3] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur": Linux-Magazin 2/04, S. 28

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

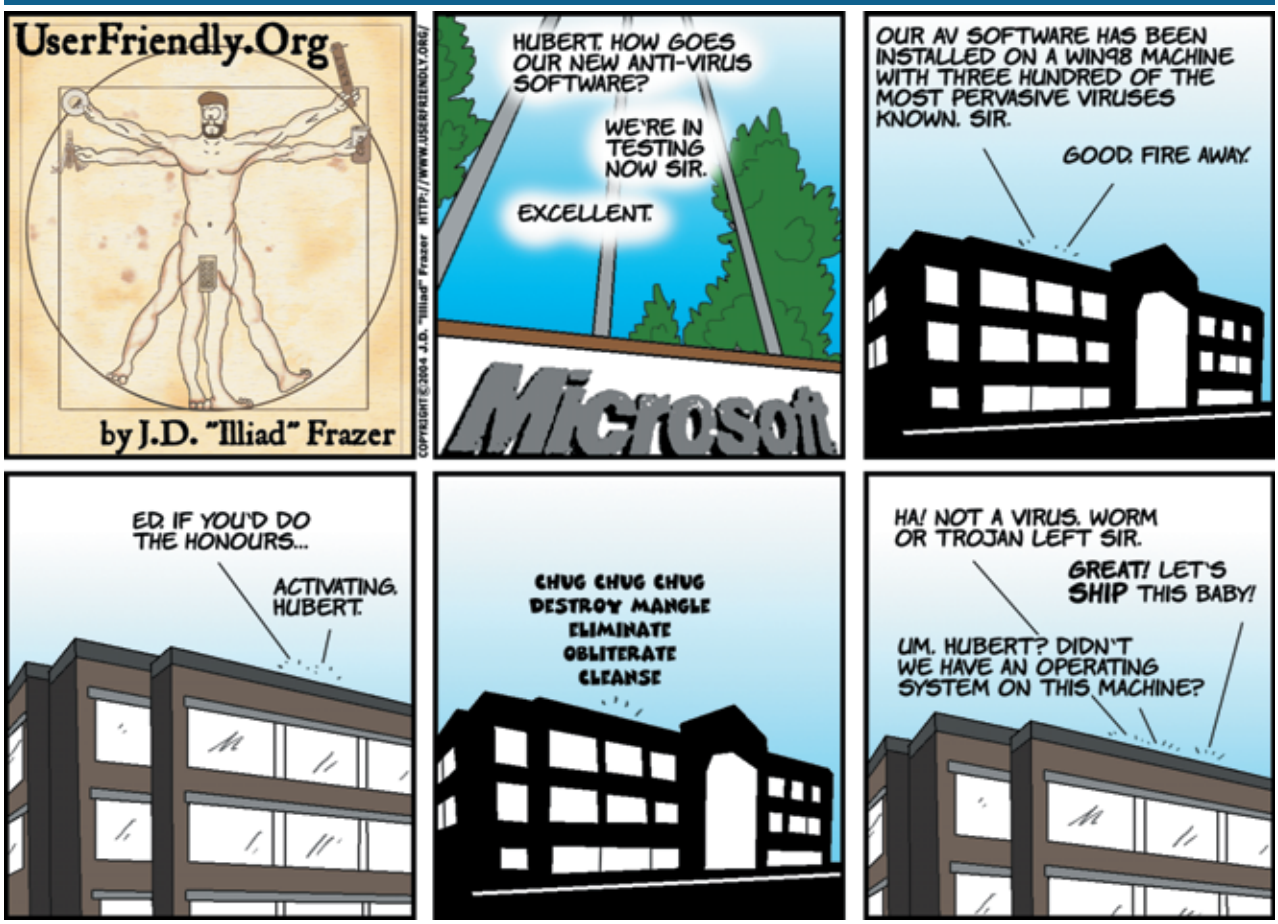
Listing 10: »include/asm-i386/unistd.h« neu

```
01 ...
02 #define __NR_mq_timedsend   (__NR_mq_open+2)
03 #define __NR_mq_timedreceive    (__NR_mq_open+3)
04 #define __NR_mq_notify      (__NR_mq_open+4)
05 #define __NR_mq_getsetattr  (__NR_mq_open+5)
06 #define __NR_udelay            283
07
08 #define NR_syscalls 284
09 ...
```

Listing 11: Testprogramm »testsys.c«

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <asm/unistd.h>
04
05 int main( int argc, char **argv )
06 {
07         int ret;
08
09         ret = syscall( __NR_udelay, 9000 );
10         printf("udelay(9000)=%d\n", ret);
11         ret = syscall( __NR_udelay, 10001 );
12         perror( "udelay" );
13         printf("udelay(10001)=%d\n", ret);
14         return 0;
15 }
```

User Friendly – der monatliche Comic im Linux-Magazin



Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 14

Kern-Technik

Das Betriebssystem stellt seine Systemaufrufe nur Anwendungen zur Verfügung. Doch manchmal will man solche Services im Kernel selbst verwenden. Das geht, erfordert aber einige Klimmzüge. Eva-Katharina Kunst, Jürgen Quade



Für den Kernel selbst sind Systemcalls in der Regel tabu. Trotzdem ist es in einigen Fällen erforderlich, innerhalb des Kernels Dateien zu lesen, Zugriffsrechte zu ändern oder Programme zu starten. Will der Kernel etwa einen Coredump schreiben oder will sein Hotplug-Subsystem ein Modul laden, sind Systemcalls auch für ihn selbst sehr praktisch. Grundsätzlich stehen dazu drei Möglichkeiten zur Verfügung, siehe Abbildungen 1 bis 3.

Anwendungen helfen

Lösungsansatz 1: Ein Rechenprozess wartet darauf, die Aufträge stellvertretend für den Kernel anzustoßen. Das erfordert aber nicht nur eine geeignete Applikation, sondern auch ein virtuelles Gerät, das der Kern bereitstellt. Beim Zugriff auf dieses Device, etwa über »read()«, legt sich die Anwendung nämlich schlafen. Dass sie vor Verwendung des Moduls schon laufen muss, macht die Sache nicht einfacher.

Etwas eleganter und verbreiteter ist der zweite Lösungsansatz. Hier startet der Kernel selbst die Applikation. Dem Programmierer steht dafür die Funktion »int call_usermodehelper(char *path, char **argv, char **envp, int wait)« zur Verfügung. Sie startet die Userspace-Applikation, die sich hinter »path« verbirgt. Die Funktion bekommt die mit »NULL« terminierte Parameterliste »argv« und die ebenfalls mit »NULL« abgeschlossene Liste des Environments »envp« übergeben.

Der vierte Parameter legt fest, ob »call_usermodehelper()« die Userspace-Applikation im Hintergrund startet oder ob der Kernel auf ihr Ende wartet. Lässt sich die Applikation nicht starten, zum Beispiel wegen fehlender Zugriffsrechte, gibt die Funktion denselben Fehlercode zurück, den sonst der Systemcall »execve()« liefert. Startet der Kernel die Applikation im Hintergrund, ist der Rückgabewert »0«. Wartet er, bis die Anwendung fertig ist, liefert »call_usermodehelper()« deren Exit-Status zurück. Der gestartete Prozess läuft übrigens als Kind des »keventd« (siehe dazu [1]) und mit Superuser-Rechten.

Listing 1 zeigt eine Anwendung, die im Auftrag des Kernels die Datei »/etc/motd« nach »/tmp/foo« kopiert. Wer einen Kernel 2.6 benutzt und die zugehörigen Kernelquellen installiert und konfiguriert hat, kann den Quellcode mit dem Makefile von der Website [2] übersetzen. Da dieses Beispielmodul ohnehin im Kontext von »insmod« abläuft, verhindert »return -EIO« (Listing 1, Zeile 17), dass sich das Modul im Kernel fest-

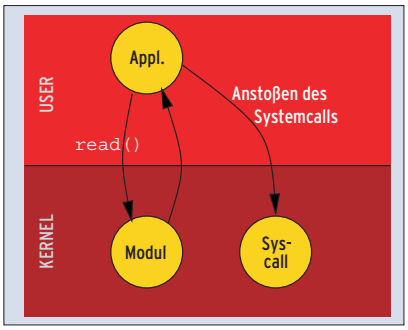


Abbildung 1: Erster Lösungsansatz: Eine laufende Anwendung nimmt Anfragen des Kernels entgegen und ruft für ihn die angefragten Systemcalls auf.

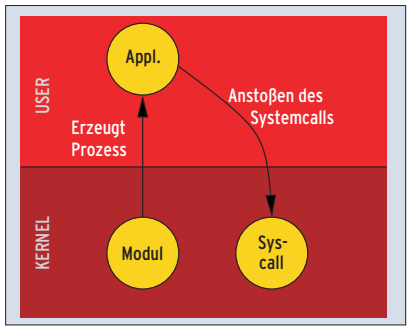


Abbildung 2: Zweiter Lösungsansatz: Die Funktion »call_usermodehelper()« startet eine Applikation, die stellvertretend den Systemcall aufruft.

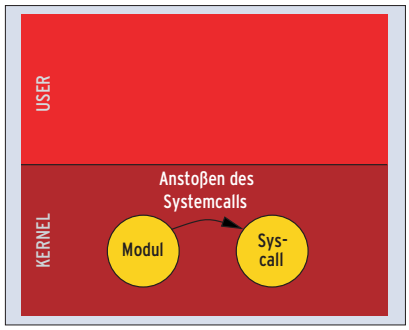


Abbildung 3: Dritter Lösungsansatz: Der Kernel ruft den Systemcall direkt auf. Bei dieser Methode ist keine Hilfe aus dem Userspace nötig.

setzt. Beim Laden des Moduls erscheint dann zwar eine Fehlermeldung, doch diese Methode verkürzt den Testlauf. Immerhin entfällt dadurch das Entladen des Moduls.

Alleingang des Kernels

Es gibt auch Aufgaben, für die das beschriebene Verfahren weniger gut geeignet ist. Sei es aus Performance- oder Design-Gründen: Manchmal soll ein Systemcall direkt vom Kernel, ohne Umweg über eine Applikation, aufgerufen werden. Problematisch ist dabei, dass Systemcalls unter anderem zwei Dinge voraussetzen: Ressourcen im Userspace und einen Prozesskontext.

Kritisch sind aus diesem Grund die Routinen »copy_to_user()« und »copy_from_user()«, die von vielen Systemcalls verwendet werden. Diese Funktionen können nicht nur Daten zwischen User- und Kernelspace, sondern auch innerhalb des Kernelspace austauschen. Zuvor muss allerdings der Sicherheits-Check überwunden werden. Der verhindert nämlich, dass eine Applikation auf Kernelspeicher zugreifen kann.

Mit Hilfe des Makros »get_fs()« lässt sich zunächst jener Ort im Speicher retten, der die Grenze zwischen User- und Kernelspace markiert. Mit »set_fs(KERNEL_DS)« setzt der Programmierer diese Grenze auf einen Maximalwert, sodass der anschließende Test immer positiv ausfällt. Nach dem Aufruf des Systemcalls empfiehlt es sich jedoch, die ursprüngliche Testgrenze wiederherzustellen. Mit diesen beiden Funktionen ergibt sich typischerweise die folgende Codesequenz:

```
mm_segment_t oldfs;
...
oldfs = get_fs();
set_fs(KERNEL_DS);
... // Aufruf des Systemcalls
set_fs(oldfs);
```

Ein Prozesskontext ist nötig, weil Systemcalls häufig die aufrufende Instanz schlafen legen. Wird der Systemcall aber von einer beliebigen Kernelfunktion aufgerufen, legt er die gerade zufällig aktive Anwendung schlafen. Er hat sich praktisch ihre Umgebung geraubt. Daher muss man in den meisten Fällen – wie es auch »call_usermodehelper()« praktiziert – einen eigenen Kernel-Thread erzeugen, siehe [1].

Aufruf des Systemcalls

Um innerhalb des Kernels einen Systemcall zu nutzen, kann nicht – wie in [3] beschrieben – ein Software-Interrupt ausgelöst werden. Stattdessen ruft man die Handler-Funktionen direkt auf. Diese sind meist nach dem Schema »sys_*SysCallName*()« benannt. So heißt die Kernelfunktion zu »open()« »sys_open()«, die zu »read()« »sys_read()«. Sie besitzen im Regelfall dieselben Parameter wie die entsprechenden Funktionen auf Applikationsebene.

Die Rückgabewerte der Handlerfunktionen entsprechen der Konvention für Rückgabewerte im Kernel. Ein Wert zwischen -1 und -1000 zeigt einen Fehler an, alles andere signalisiert Erfolg. Für die Fehlerauswertung verwendet der geübte Kernelhacker das Makro »long IS_ERR (const void *ptr)«.

Wer ein dynamisches Kernelmodul schreibt, kann allerdings nur auf die Handlerfunktionen »sys_open()«, »sys_close()«, »sys_lseek()« und »sys_read()« zurückgreifen. Nur diese Funktionen exportiert der Kernel 2.6.7 bei einer x86-Architektur – das sind nicht gerade viele im Vergleich zu den Anwendungen, de-

Listing 1: Kernel startet Applikation »umh.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 static int __init mod_init(void)
07 {
08   int ret;
09   char *argv[4] = {"/bin/cp", "/etc/motd", "/tmp/
   foo", NULL};
10    char *envp[3] = {"HOME=/", "PATH=/sbin:/bin:/
   usr/sbin:/usr/bin", NULL};
11
12   ret=call_usermodehelper(argv[0],argv,envp,1);
13    if( ret < 0 ) {
14     printk("call_usermodehelper failed ...\n");
15     return -EIO;
16   }
17   return -EIO; // nur zum Test
18 // return 0;
19 }
20
21 static void __exit mod_exit(void)
22 {
23 }
24
25 module_init( mod_init );
26 module_exit( mod_exit );
```

Listing 2: »readfile.c«

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/completion.h>
05 #include <linux/fs.h>
06
07 MODULE_LICENSE("GPL");
08
09 static int ThreadID = 0;
10 static DECLARE_COMPLETION(OnExit);
11 static char buf[128];
12
13 static int ThreadCode(void *data)
        //  Prozessorkontext
14 {
15   struct file *config_file;
16   daemonize("MyKThread");
17   allow_signal( SIGTERM );
18
19   // Dateizugriffe im Kernel
20   config_file = filp_open( "/etc/motd", O_RDONLY, 0 );
21   if( IS_ERR( config_file ) )
22      return -EIO;
23   kernel_read( config_file, 0, buf, sizeof(buf) );
24   printk( buf );
25   filp_close( config_file, NULL );
26
27   // Hier sind Zugriffe auf Systemcall-Handler
        möglich
28   complete_and_exit( &OnExit, 0 );
29 }
30
31 static int __init kthreadInit(void)
32 {
33   ThreadID=kernel_thread( ThreadCode, NULL,
       CLONE_KERNEL );
34   if( ThreadID==0 )
35     return -EIO;
36   return 0;
37 }
38
39 static void __exit kthreadExit(void)
40 {
41   kill_proc( ThreadID, SIGTERM, 1 );
42   wait_for_completion( &OnExit );
43 }
44 module_init( kthreadInit );
45 module_exit( kthreadExit ); // Modul-Handling
```

nen fast 280 Systemcalls zur Verfügung stehen. Aber das bedeutet nicht, dass der Kernelprogrammierer auf die übrigen Syscalls verzichten muss.

Überhaupt lassen sich wenige Systemaufrufe im Kernel sinnvoll einsetzen. Schließlich dienen viele dazu, Informationen aus dem Kernel in den Applikationsbereich zu transportieren, zum Beispiel die UID eines Rechenprozesses. Innerhalb des Betriebssystemkerns lassen sich solche Information ohne Umweg abfragen. Die Handlerfunktionen sind vielfach nichts weiter als Wrapper um die Routinen im Kernel, die die eigentliche Arbeit ausführen.

Dateizugriff aus dem Kernel

Auch beim Zugriff auf Dateien sind die wichtigen Funktionen im Kernel versteckt. Die Handlerfunktion »sys_open()« ruft nämlich die Kernelroutine »struct file *filp_open(const char * filename, int flags, int mode)« auf. Zuvor erzeugt »sys_open()« den Filedeskriptor, den aber normalerweise nur die Applikation benötigt.

Kernelcode greift effizienter direkt über die von »filp_open()« zurückgegebene Instanz vom Typ »struct file« zu, siehe [4]. Die Parameter »flags« und »mode« der Funktion »filp_open()« geben Zugriffsart und -rechte an, wie vom Systemcall »open« bekannt ist. So öffnet die Codezeile

```
log_file = filp_open( "/tmp/foo",
   O_RDWR | O_CREAT, 0644 );
```

die Datei »/tmp/foo« zum Lesen und Schreiben. Existiert die Datei nicht, legt der Kernel sie mit den Zugriffsrechten 644 an. Die Funktion »filp_open()« gibt einen Zeiger auf »struct file *« zurück oder einen Fehlerwert. Die entsprechende Funktion zur Freigabe lautet »int filp_close(struct file *filp, fl_owner_t id)«. Der Parameter »id« vom Typ »fl_owner_t« (»struct files_struct«) ist ein Zeiger auf die zum Prozess gehörigen Dateien, wenn »filp_close()« indirekt über einen Syscall aufgerufen wurde. Im Kernelfall trägt der Programmierer entweder »0« oder »current->files« ein. Beide Funktionen sind in der Headerdatei »linux/fs.h« deklariert.

Statt »sys_read()« zum Lesen zu verwenden (was ohne Filedeskriptor auch nicht geht), bietet sich die Funktion »ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)« an (Abbildung 4). Zuvor muss aber noch wie beschrieben der Speicher-Check ausgehebelt werden. Eine Kernelfunktion macht das selbst und kopiert auch gleich die Daten: »int kernel_read(struct file *file, unsigned long offset, char *addr, unsigned long count)«.

Im Fall des File- oder Gerätezugriffs ließen sich auch direkt die Lese- und Schreibfunktionen benutzen, die in der »struct file_operations« abgelegt sind, siehe [4]. Doch ist das mit mehr Programmieraufwand verbunden. So ist vorher zu prüfen, ob die Zugriffsfunktion überhaupt existiert. Schließlich gibt es kein Gesetz, dass jeder Treiber jede Funktion implementieren muss.

Listing 2 zeigt einen Programmausschnitt, der innerhalb des Kernels die Datei »/etc/motd« einliest und über den Syslog-Daemon wieder ausgibt. Zur Übersetzung des Moduls ist im Makefile

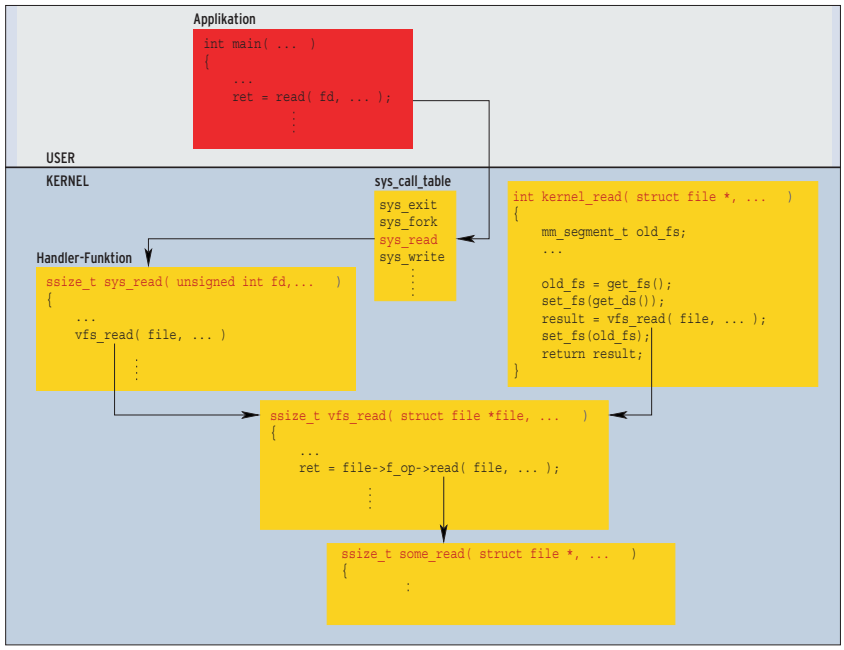


Abbildung 4: Im Kernel sind einige Funktionen beteiligt, wenn aus einer Datei gelesen wird. Die Userspace-Funktion »read()« ruft die Handler-Funktion »sys_read()« auf, diese wiederum »vfs_read()«.

Listing 3: Schreibzugriff aus dem Kernel

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03 #include <linux/fs.h>
04 #include <asm/uaccess.h>
05
06 MODULE_LICENSE("GPL");
07
08 static char filename[255];
09 module_param_string( filename, filename,
   sizeof(filename), 666 );
10 struct file *log_file;
11
12 static int __init mod_init(void)
13 {
14    mm_segment_t oldfs;
15
16    if( filename[0]=='\0' )
17       strncpy( filename, "/tmp/kernel_file",
   sizeof(filename) );
18    printk("opening filename: %s\n", filename);
19    log_file = filp_open( filename,
   O_WRONLY|O_CREAT, 0 );
20    if( IS_ERR( log_file ) )
21       return -EIO;
22
23    oldfs = get_fs();
24    set_fs( KERNEL_DS );
25    vfs_write( log_file, "Hello World\n", 12,
   &log_file->f_pos );
26    set_fs( oldfs );
27    filp_close( log_file, NULL );
28    return 0;
29 }
30
31 static void __exit mod_exit(void)
32 {
33 }
34 module_init( mod_init );
35 module_exit( mod_exit );
```

nur der Modulname durch den Namen des Moduls aus Listing 2 zu ersetzen: Aus »umh.o« wird »readfile.o«. Nach dem Laden des Moduls taucht im Syslog, zum Beispiel in »/var/log/messages«, der Inhalt von »/etc/motd« auf.

Schreiben von Dateien

Eine Datei direkt vom Kernel schreiben funktioniert ähnlich wie das Lesen. Auch hier bietet ein Aufruf von

```
Instanz->f_op->write(log_file,
    "hallo\n", 7,  &offset );
```

direkten Zugriff, falls die Schreibfunktion »write()« implementiert ist. Die Funktion »ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)« nimmt dem Programmierer derartige Überprüfungen ab. Eine Funktion, die analog zu »kernel_read()« die Überprüfung selbst aushebelt, gibt es allerdings nicht. Beim Aufruf der Funktion ist noch darauf zu achten, dass der Parameter »pos« auch wirklich ein Zeiger ist.

Der in Listing 3 vorgestellte Kernelcode zeigt ein einfaches Beispiel, das den eben beschriebenen Weg geht. Es schreibt den String »Hello World« in eine Datei. Zum Übersetzen ist wiederum das Makefile anzupassen. Ohne Angabe des Modulparameters schreibt das Modul seinen String in die Datei »/tmp/kernel_file«. Dieses Beispiel erzeugt keinen eigenen Thread, es borgt sich den Prozesskontext von »insmod«.

Ausblick

Mit einigen Tricks lassen sich also auch im Kernel Systemaufrufe verwenden. Man sollte dabei aber auch an Sicherheitsaspekte denken, denn wo mehr Code im Kernelmodus abläuft, dort sind auch mehr potenziell gefährliche Fehler. Dem Programmierer steht durch die vorgestellten Techniken im Kernel eine Vielzahl von Funktionen zur Verfügung, mit denen er zum Beispiel Dateien lesen und schreiben kann. Wie man im Kernel Netzwerkfunktionen nutzt, zeigt die nächste Folge dieser Reihe. *(ofr)* ■

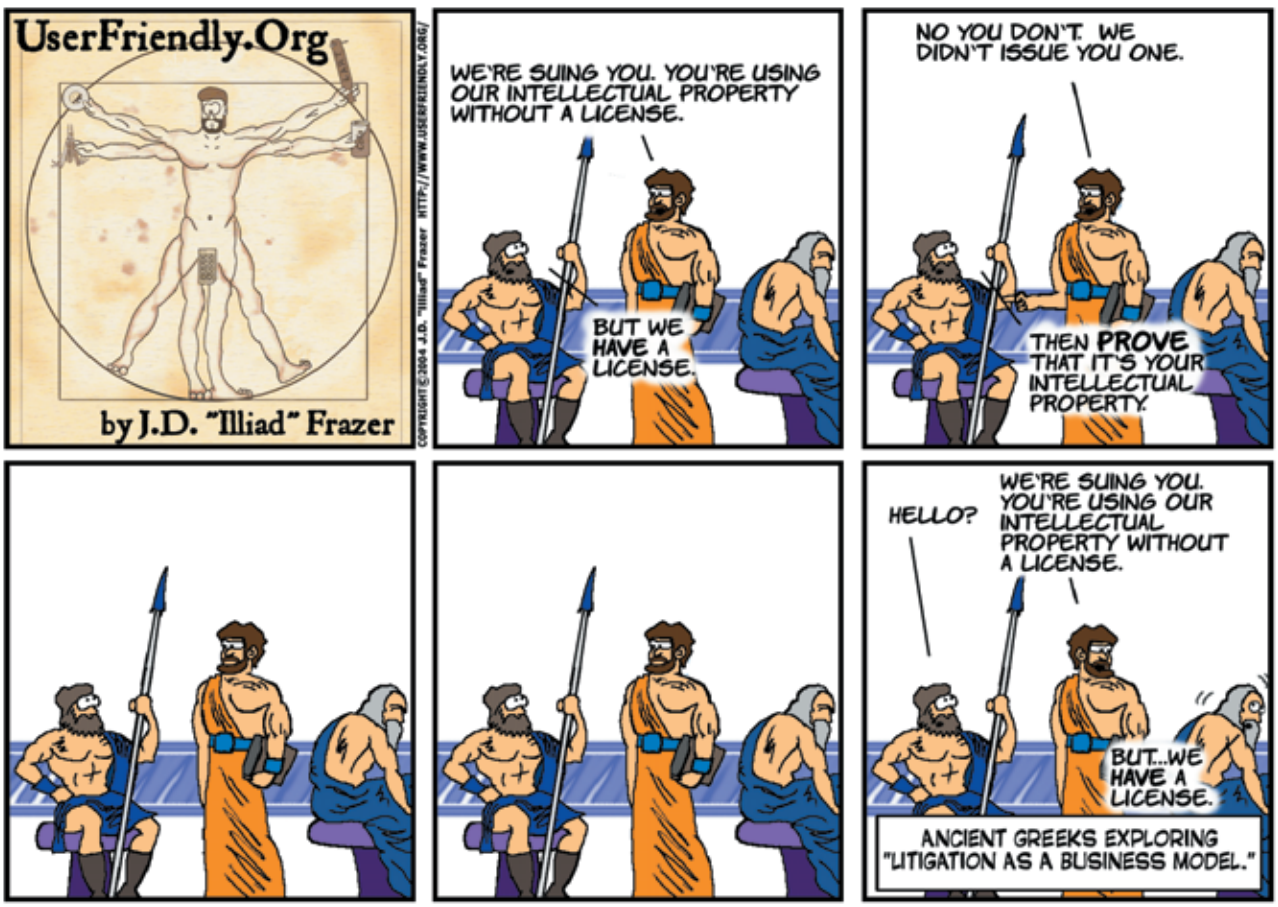
Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 4: Linux-Magazin 11/03, S. 96
[2] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/09/Kern-Technik]
[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 13: Linux-Magazin 8/04, S. 92
[4] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 9/03, S. 86

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, haben unter dem Titel »Linux-Treiber entwickeln« ein Buch zum Kernel 2.6 veröffentlicht.

User Friendly – der monatliche Comic im Linux-Magazin



Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 15

Kern-Technik

Dass es TCP/IP-Netzwerkfunktionen im Kernel gibt, überrascht kaum, denn schließlich implementiert der Betriebssystemkern den zugehörigen Stack. Das Nutzen der Funktionen in eigenen Kernelmodulen funktioniert aber etwas anders als bei den Userspace-Verwandten. Eva-Katharina Kunst, Jürgen Quade



Wer im Kernel eigene Netzwerkdienste programmiert, muss einige Eigenheiten des dort implentierten TCP/IP-Codes beachten, ähnlich wie beim Zugriff auf Dateien [1]. Glücklicherweise erinnert das Kernel-Interface an das der bekannten Userspace-Funktionen, siehe Kasten „TCP/IP im Userspace“. Die meisten Netzwerkzugriffe sind allerdings nur in einem Prozesskontext möglich. Deshalb muss sich eine TCP/IP-Funktionen im Kernel entweder den Prozesskontext einer Userspace-Anwendung borgen oder braucht einen eigenen Kernel-Thread, wie [2] beschreibt.

Listing 1 zeigt Ausschnitte eines Kernel-internen Echoservers, der die vorgestellten Funktionen benutzt – der komplette Code und das Makefile liegen auf [3]. Alle Daten, die der Server empfängt, schickt er an den Absender zurück. Das lässt sich leicht prüfen, siehe Kasten „Test mit Telnet“.

Im Kernel wird ein Socket durch die Funktion »sock_create()« erzeugt, siehe Kasten „TCP/IP-Schnittstellenfunktionen im Kernel“ und Abbildung 2. Das auf diese Weise erzeugte Socket-Objekt stellt in der Struktur »ops« weitere Methoden über Funktionspointer zur Verfügung, siehe den Typ »proto_ops« im Header »linux/net.h«.

Die Methode »bind()« weist dem Socket eine Adresse zu, die sie als erstes Argument erwartet. Die beiden übrigen Parameter entsprechen denen ihres Gegenstücks im Userspace: Das Element vom Typ »struct sockaddr_in« spezifiziert unter anderem den Port, auf den der Server hört, »addrlen« gibt die Länge dieser Datenstruktur an.

Der Port muss, wie bei Netzanwendungen üblich, mit Hilfe von »htons()« in Network Byte Order konvertiert werden. Die Funktion »listen()« lässt den vorbereiteten Socket schließlich auf Verbinungen warten. Neben der Adresse des Sockets steht hier als zweites Argument die Anzahl der Clients, die gleichzeitig zugreifen dürfen.

Sockets klonen

Nimmt ein Client Kontakt zum Server auf, brauchen er zur Kommunikation den existierenden Socket. Also muss der Server für jede neue Verbindung einen neuen Socket anlegen, was er im Kernel mit »socket_alloc()« erledigt, Listing 1 Zeile 47. Der neue Socket und der auf Verbindungen wartende sind Argumente der Funktion »ops->accept()«. Zusätzlich erwartet sie den Zugriffsmodus, wie er von der »open()« bekannt ist. Ohne besonderen Zugriffsmodus (»flags = 0«), blockiert der Kernel-Thread so lange, bis ein Client eine Verbindung aufbaut. Wird »flags« mit »O_NONBLOCK« initialisiert, kehrt »ops->accept()« sofort zurück. Die Funktion »socket_accept()« (Zeilen 40 bis 69) implementiert ungefähr dieselbe Funktionalität wie das im Userspace bestehende »accept()«.

Ein positiver Return-Wert von »ops->accept()« zeigt an, dass der Verbindungsaufbau erfolgreich war. Wer mehr über den Client erfahren möchte, ruft die Methode »getname()« auf, die dessen IP-Adresse und Port zurückgibt, siehe Listing 1, Zeilen 57 bis 67.

Zugriffsmodus festlegen

Der Datenaustausch läuft über die Funktionen »sock_recvmsg()« und »sock_sendmsg()«, Zeilen 89 und 112. Sie besitzen drei gleiche Parameter: den Socket, ein Objekt vom Typ »msghdr« und die Länge des Speicherbereichs. Bei »sock_recvmsg()« kommen noch die Flags für die Zugriffsart dazu, die »linux/socket.h« festlegt. Für einen nicht blockierenden Zugriff steht beispielsweise »MSG_DONTWAIT«.

Der wichtigste Parameter dieser Funktionen ist die Datenstruktur vom Typ »msghdr«. Sie übernimmt nämlich in »msg_iov« die Adresse des Speicherbereichs »iov«, in dem die Daten bei Lese- und Schreibzugriffen abgelegt werden, siehe Listing 1, Zeilen 79 bis 84. In »msg_iovlen« steht die Anzahl der verwendeten Speicherblocks, in diesem Fall nur einer. Die Strukturelemente »msg_

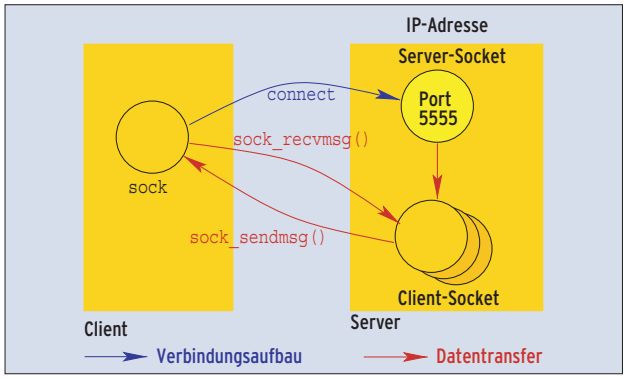


Abbildung 1: Beim Verbindungsaufbau erzeugt der Server einen neuen Port, der für den eigentlichen Datenaustausch verwendet wird.

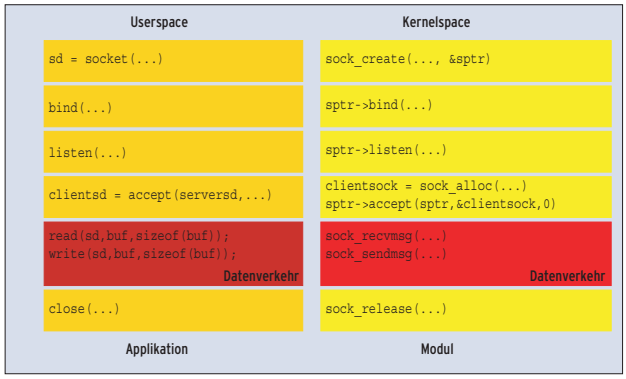


Abbildung 2: Die meisten Interfaces zu TCP/IP auf der Applikationsseite haben direkte Entsprechungen im Kernel.

control« und »msg_controllen« spielen im Beispiel keine Rolle und erhalten dementsprechend nur Standardwerte. Sie dienen sonst unter anderem der Übermittlung von IP-Optionen über Unix-Sockets. Näher sind die einzelnen Elemente von »msghdr« in der Manualseite zu der Funktion »recvmsg()« beschrieben, die so funktioniert wie die hier verwendete Kernelfunktion »sock_recvmsg()«.

Tricks zum Speicherzugriff

Die Makros »get_fs()« und »set_fs()« ermöglichen, wie in [1] beschrieben, den Datentransfer zwischen User- und Kernelspace, siehe Listing 1, Zeilen 87 bis 90. Die Funktion »sock_recvmsg()« gibt die Anzahl der empfangenen, »sock_sendmsg()« die der verschickten Bytes zurück. Ist zwischenzeitlich die Verbindung abgebrochen, liefern sie einen Wert kleiner null. Der Server ruft in diesem Fall – oder wenn er seinerseits die Verbindung beenden will – die Funktion »sock_release()« auf. Damit gibt der Programmierer auch später den verwendeten Socket wieder frei.

Nächster Kunde, bitte!

Im Gegensatz zum Server muss der Client nicht auf Verbindungen warten. Deshalb beschränkt sich die Implementation auf den Verbindungsaufbau. Zuerst erzeugt »socket_create()« einen Socket. Die Struktur »sockaddr_in« spezifiziert dann den gewünschten Server, siehe Zeilen 25 bis 27 in Listing 2. Dessen IP-Adresse darf übrigens auch in Ascii vorliegen, der so genannten Dotted Decimal Notation. Dafür bietet der Kernel die Funktion »in_aton()«, die die IP-Adresse in Binärform übersetzt. Die Struktur »sockaddr_in« ist einer der vier Parameter der Methode »connect()«, die den Server kontaktiert (Zeile 28).

Für den Datenaustausch benutzen Client wie Server die beiden Methoden »sock_recvmsg()« und »sock_sendmsg()«. Die Verbindung schließt der Client ebenfalls mit »sock_release()«. Der Client schickt ein »hallo« an den Echoserver, der auf demselben Rechner auf Port 5555 wartet (siehe Kasten „Test mit Telnet“). Er

TCP/IP im Userspace

Die Kommunikation mit der Internet-Standardprotokollfamilie TCP/IP funktioniert nach dem Client-Server-Prinzip. Der Client addressiert einen Server über die Kombination von IP-Adresse und Port, der Server wartet auf die Anfrage eines Clients. Ist es so weit, erzeugt er einen neuen Socket, über den der Datentransfer abläuft. Parallel dazu nimmt der Server bereits die nächste Anfrage entgegen (siehe Abbildung 1).

Um einen Userspace-Server zu programmieren, kommen überlicherweise die Funktionen »socket()«, »bind()«, »listen()« und »accept()« zum Einsatz. Der Client baut eine Verbindung über »socket()« und »connect()« auf (Abbildung 2).

Der Aufruf von »socket()« richtet einen Kommunikationsendpunkt ein, der als Socket bezeichnet wird. Die Funktion »bind()« weist dem Socket einen Port zu, der dem Client bekannt sein muss. Für einige Applikationen sind die Ports festgelegt (siehe die Datei »/etc/services«). So steht die Portnummer 80 für den HTTP-Service, also das übliche WWW-Protokoll. Der Aufruf von »listen()« legt die maximale Anzahl gleichzeitiger Verbindungen fest. Damit lässt sich die Gefahr einer Überlastung des Servers von vornherein minimieren.

»accept()« ist die Funktion, mit der der Server auf eine neue Verbindung wartet. Sobald ein Client per »connect()« eine Verbindung aufbaut, erzeugt diese Funktion einen neuen Socket, über den Client und Server Daten austauschen, meist über »read()« und »write()«. Mit dem ursprünglichen Socket wartet der Server auf die nächste Verbindung. Wird ein Socket nicht mehr benötigt, gibt die Funktion »close()« ihn wieder frei.

Test mit Telnet

Haben Sie das Server-Modul »echo.c« übersetzt und mit »insmod echo.ko« geladen, sollten Sie sicherstellen, dass Sie den hier verwendeten Port 5555 nicht anderweitig verwenden oder durch eine Firewall geblockt haben. Dann geben Sie »telnet localhost 5555« ein. Der Aufruf kann natürlich auch von einem anderen Rechner aus erfolgen. In diesem Fall ist »localhost« durch den Rechnernamen des neuen Echoservers zu ersetzen.

Sie können jetzt Nachrichten eintippen, die nach jedem Return zum Server gesendet und von diesem zurückgeschickt werden. Telnet gibt die Antworten des Echoservers dann aus. Um die Verbindung zu beenden, verlassen Sie einfach das Telnet-Programm. Dazu geben Sie [Strg]+[ ]] gefolgt von »quit« ein. Die Verbindung wird ebenfalls abgebrochen, wenn Sie den Echoserver per »rmmod echo« aus dem Kernel entladen.

Um den Client zu testen, laden sie entweder das Servermodul oder verwenden den Standard-Echoserver im Userspace. Läuft der Internet-Superdaemon »xinetd«, setzen Sie in »/etc/xinet.d/echo« die Variable »disabled« auf »No« und schicken dem Server ein HUP-Signal. Dann müssen Sie allerdings auch im Sourcecode des Clients den Port auf »7« ändern.

wartet auf die Antwort und baut die Verbindung anschließend wieder ab. Trägt man im Code von Listing 2 anstelle von »127.0.0.1« die IP-Adresse des Rechners im lokalen Netz ein, können Client und Server auch auf unterschiedlichen Rechnern laufen.

Um das Beispiel kurz zu halten, leiht sich das Modul den Prozesskontext von Insmod. Das Modul zeigt nach erfolgter Kommunikation dem Kernel einen Initialisierungsfehler an. Das erspart manuelles Entladen. Entfernt man den Kommentar von »//return 0;« in Zeile 55, verschwindet dieser Fehler, das Modul muss dann allerdings mit »rmmod« von Hand entladen werden. Den Testablauf dokumentiert der Client über »printk()« im Syslog.

Laufen Netzwerk-Services im Betriebssystemkern, sind sie besonders perfor-

Listing 1: »echo.c«

```
39 ...
40 static struct socket *socket_accept( struct
   socket *server )
41 {
42   struct socket *clientsocket=NULL;
43   struct sockaddr address;
44   int error, len;
45
46   if( server==NULL ) return NULL;
47   clientsocket = sock_alloc();
48   if( clientsocket==NULL ) return NULL;
49
50   clientsocket->type = server->type;
51   clientsocket->ops = server->ops;
52   error=server->ops->accept(server,
       clientsocket,0);
53   if( error<0 ) {
54     sock_release(clientsocket);
55     return NULL;
56   }
57   error=server->ops->getname(clientsocket,
58                         (struct sockaddr *)
       &address, &len,2);
59   if( error<0 ) {
60     sock_release(clientsocket);
61     return NULL;
62   }
63   printk(KERN_INFO "new connection (%d) from
       %u.%u.%u.%u\n", error,
64      (unsigned char)address.sa_data[2],
65      (unsigned char)address.sa_data[3],
66      (unsigned char)address.sa_data[4],
67      (unsigned char)address.sa_data[5] );
68   return clientsocket;
69 }
70
71 static int server_send( struct socket *sock,
   unsigned char *buf, int len )
72 {
73   struct msghdr msg;
74   struct iovec iov;
75   mm_segment_t oldfs;
76
77   if( sock->sk==NULL )
78     return 0;
79   iov.iov_base = buf;
80   iov.iov_len  = len;
81   msg.msg_control = NULL;
82   msg.msg_controllen = 0;
83   msg.msg_iov        = &iov;
84   msg.msg_iovlen = 1;
85   msg.msg_flags = 0;
86
87   oldfs = get_fs();
88   set_fs( KERNEL_DS );
89   len = sock_sendmsg( sock, &msg, len );
90   set_fs( oldfs );
91
92   return len;
93 }
94
95 static int server_receive( struct socket
   *sptr, unsigned char *buf, int len )
96 {
97   struct msghdr msg;
98   struct iovec iov;
99   mm_segment_t oldfs;
100
101   if( sptr->sk==NULL )
102     return 0;
103   iov.iov_base = buf;
104   iov.iov_len  = len;
105   msg.msg_control = NULL;
106   msg.msg_controllen = 0;
107   msg.msg_iov       = &iov;
108   msg.msg_iovlen = 1;
109
110   oldfs = get_fs();
111   set_fs( KERNEL_DS );
112   len = sock_recvmsg( sptr, &msg, len, 0 );
113   set_fs( oldfs );
114
115   return len;
116 }
117 ...
```

Listing 2: »client.c«

```
11 ...
12 static int __init client_init( void )
13 {
14   int len, clienterror;
15   char buf[64];
16   struct msghdr msg;
17   struct iovec iov;
18   mm_segment_t oldfs;
19   struct sockaddr_in client;
20
21   if( sock_create( PF_INET,SOCK_STREAM,
       IPPROTO_TCP,&clientsocket)<0 ) {
22     printk( KERN_ERR "server: Error
         creating clientsocket.\n" );
23     return -EIO;
24   }
25   client.sin_family = AF_INET;
26   client.sin_addr.s_addr = in_aton
       ("127.0.0.1"); /* destination addr */
27   client.sin_port = htons( (unsigned
       short)serverport );
28   clienterror = clientsocket->ops->connect
       ( clientsocket,
29   (struct sockaddr *) &client, sizeof
       ( client ), 0 );
30   if( clienterror < 0 ) {
31     printk( KERN_ERR "Carrera: Connect
         error = %d on clientsocket",
32         clienterror );
33     return -EIO;
34   }
35   memcpy( buf, "hallo", 6 );
36   iov.iov_base = buf;
37   iov.iov_len  = 6;
38   msg.msg_control = NULL;
39   msg.msg_controllen = 0;
40   msg.msg_iov        = &iov;
41   msg.msg_iovlen = 1;
42   msg.msg_flags = 0;
43
44   oldfs = get_fs();
45   set_fs( KERNEL_DS );
46   len = sock_sendmsg( clientsocket, &msg,
       6 );
47   if( len > 0 ) {
48     iov.iov_len  = sizeof(buf);
49     len = sock_recvmsg( clientsocket,
         &msg, sizeof(buf), 0 );
50     if( len > 0 )
51       printk( KERN_INFO "returned:
           \"%s\"\n", buf);
52   }
53   set_fs( oldfs );
54
55   //return 0;
56   if( clientsocket ) {
57     sock_release( clientsocket );
58     clientsocket = NULL;
59   }
60   return -EIO;
61 }
62
63 static void __exit client_exit( void )
64 {
65   if( clientsocket )
66     sock_release( clientsocket );
67 }
68
69 module_init( client_init );
70 module_exit( client_exit );
```

mant. So ergab eine von IBM durchgeführte Untersuchung, dass ein im Kernel implementierter Webserver eine dreifach bessere Performance aufwies als ein Webserver im Userspace [5].

Eingebauter Server in 2.6

Im Kernel 2.4 überzeugte der eingebaute Webserver Khttpd durch seine Geschwindigkeit, doch er ist mittlerweile nicht mehr Teil des Standardkernels. Der In-Kernel-Server Tux [6] von Ingo Molnar funktioniert mit den aktuellen Kernen, zum Redaktionsschluss zumindest mit Linux 2.6.5 – also eine Gelegenheit für eigene Experimente ohne großen Programmieraufwand. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 14: Linux-Magazin 9/04, S. 92
[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 4: Linux-Magazin 11/03, S. 96
[3] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/10/Kern-Technik/]
[4] E.-K. Kunst und J. Quade, „Kern-Technik", Folge 13: Linux-Magazin 8/04, S. 92
[5] Joubert, King et al.: „High-Performance Memory-Based Web Servers: Kernel and User-Space Performance": [http://www.usenix.org/events/usenix01/full_papers/joubert/joubert_html/index.html]
[6] In-Kernel-Webserver Tux: [http://people.redhat.com/mingo/TUX-patches/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

TCP/IP-Schnittstellenfunktionen im Kernel

Sockets erzeugen und freigeben	
`int sock_create(int family, int type, int protocol, struct socket **res);`	Die Funktion erzeugt einen Socket vom Typ »type« aus der Protokollfamilie »family« für das Protokoll »protocol«. Die Adresse des erzeugten Sockets steht in »res«. Die »family« ist normalerweise »PF_INET«, der Typ »SOCK_DGRAM« für einen UDP-Socket und »SOCK_STREAM« für einen TCP-Socket. Bei Erfolg gibt die Funktion »0« zurück, sonst einen negativen Fehlercode. Unter anderem kommen die folgenden Fehlercodes vor: »EAFNOSUPPORT« (ausgewählte »family« wird nicht unterstützt), »EINVAL« (ausgewählter »type« wird nicht unterstützt) und »ENFILE« (keine Socket-Ressource mehr verfügbar).
`void sock_release(struct socket *sock)`	Die Funktion gibt den Socket »sock« wieder frei.
`struct socket *sock_alloc(void);`	Die Funktion allokiert und initialisiert ein Socket-Objekt. Bei Erfolg gibt die Funktion die Adresse des neu erzeugten Socket-Objekts zurück, ansonsten »0«.
Methoden des Socket-Objekts	
`int (*bind)(struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);`	Die Funktion weist dem Socket »sock« den in »myaddr« angegebenen Namen (Portnummer) zu. »myaddr« hat die Länge »sockaddr_len«. Läuft alles glatt, gibt die Funktion »0« zurück, sonst einen negativen Fehlercode.
`int (*listen)(struct socket *sock, int len);`	Setzt die Anzahl der gleichzeitig offenen Verbindungen des Sockets »sock« auf »len«. Die Funktion gibt nach erfolgreichem Ablauf »0« zurück, sonst einen negativen Fehlercode.
`int (*accept)(struct socket *sock, struct socket *newsock, int flags);`	Die Methode nimmt vom verbindungsorientierten Socket »sock« den nächsten Verbindungswunsch entgegen und initialisiert »newsock« entsprechend. Mit »flags« wird der Zugriffsmodus (siehe Headerdatei »linux/socket.h«) eingestellt. Liegt ein Verbindungswunsch vor, gibt die Funktion »0« zurück, andernfalls einen negativen Fehlercode.
`int (*getname)(struct socket *sock, struct sockaddr *addr, int *sockaddr_len, int peer);`	Diese Funktion schreibt in »addr« Informationen über Portnummer und IP-Adresse des Sockets »sock«. Ist »peer« mit »0« belegt, gibt die Methode die Adressangaben des Sockets selbst zurück. Bei jedem anderen Wert für »peer« schreibt sie die Adressangaben des Remote-Sockets in »addr«. In »sockaddr_len« wird die Größe der abgelegten Adressinformation zurückgegeben. Der Rückgabewert ist »-ENOTCONN«, falls der Socket nicht verbunden ist, sonst »0«.
Funktionen zum Datenaustausch	
`int sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t size, int flags);`	Diese Funktion liest Daten vom Socket »sock« und legt sie im Speicher ab, der neben anderen Informationen in »msg« definiert ist. »size« gibt die Größe des Speicherbereichs in Bytes an. »flags« spezifiziert die Zugriffsart (beispielsweise »MSG_DONTWAIT« für nicht blockierend). Die Funktion gibt die Anzahl der gelesenen Bytes zurück. Sollte die Verbindung zwischenzeitlich abbrechen, gibt die Funktion einen negativen Fehlerwert zurück.
`int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size);`	Verschickt die im Speicherbereich abgelegten Daten an den über »sock« verbundenen Kommunikationspartner. Der Speicherbereich selbst ist in einem Element der Struktur »msg« definiert. »size« gibt die Größe dieses Speicherbereichs an. Die Funktion gibt die Anzahl der gesendeten Bytes zurück, im Fehlerfall oder beim Verbindungsabbruch stattdessen einen negativen Fehlercode. Weitere Informationen über die Fehlerursache lassen sich der Struktur »msg« entnehmen.
Hilfsfunktionen	
`__u32 in_aton(const char *str)`	Die Funktion konvertiert einen Ascii-String (Dotted Decimal) in eine binäre Repräsentation der IP-Adresse. Rückgabewert ist die konvertierte Adresse.
`unsigned short int htons( unsigned short int );` `__u32 htonl(__u32 );` `unsigned short int ntohs( unsigned short int );` `__u32 ntohl(__u32 );`	Diese vier Funktionen übersetzen einen Long- oder einen Integerwert vom der Host Byte Order in Network Byte Order und umgekehrt.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 16

Kern-Technik

Verbindungslose Kommunikation über UDP ist im Kernel mit weniger Aufwand zu programmieren als das verbindungsorientierte TCP. Eine geschickte Einstellung der Socket-Parameter lässt eigene Module auf Broadcasts hören und steigert die Performance. Eva-Katharina Kunst, Jürgen Quade



Nachdem die letzte Folge der Kern-Technik die Verwendung von TCP im Kernel demonstriert hat [1], geht es dieses Mal um UDP. Dabei handelt es sich um ein verbindungsloses Protokoll, was sich bereits am einfachen Programmierinterface bemerkbar macht: Der Server öffnet einen Socket, bindet ihn an einen Port und wartet auf die an ihn geschickten Nachrichten. Kommt eine an, nutzt der Server die übermittelte Absenderinformation, um dem Client zu antworten. Dazu braucht er nicht einmal einen neuen Socket anzulegen.

Der Client öffnet ebenfalls einen Socket und schickt seine Nachricht an den angegebenen Empfänger, also den Server. Dessen Identifikation erfolgt wie bei TCP über die Kombination von IP-Adresse und Port. Nicht mehr benötigte Sockets gibt das Modul schließlich wieder frei. Innerhalb des Kernels stehen für UDP-Kommunikation insgesamt fünf Funktionen zur Verfügung (siehe Abbildung 1): »sock_create()«, »bind()«, »sock_recvmsg()«, »sock_sendmsg()« und »sock_release()«.

Listing 1 zeigt, wie diese Funktionen zu verwenden sind. In Zeile 56 erzeugt »sock_create()« den Socket, wobei die Symbole »SOCK_DGRAM« und »IPPROTO_UDP« festlegen, dass es sich um einen UDP-Socket handelt. Den in der Struktur »struct sockaddr_in« spezifizierten Empfangsport bindet die Funktion »bind()« an den Socket (Zeile 63).

Etwas komplizierter ist der Aufruf der Empfangsfunktion selbst. Sie benötigt neben dem Speicher für die Daten nämlich noch drei Datenstrukturen (siehe Abbildung 2): eine Struktur »struct iovec« für die Adresse und Größe des Empfangspuffers, eine Struktur vom Typ »struct sockaddr_in« für den Paketabsender (IP-Adresse und Portnummer) und eine Struktur »struct msghdr msg«. Sie speichert die Adressen der anderen Datenstrukturen (Zeilen 25 und 29). Zur Initialisierung der Datenstruktur »struct iovec« müssen die Adresse und Größe des Empfangs-Datenspeichers zudem spezifiziert sein (Zeilen 32 und 33).

Wiederholt initialisieren

Den Inhalt der Datenstruktur »struct iovec« modifiziert der Kernel. Deshalb ist sie vor jedem Aufruf von »sock_recvmsg()« neu zu initialisieren. Das ist auch der Grund, warum die Initialisierung in Listing 1 innerhalb der »while«-Schleife (Zeile 31) stattfindet.

Zum Aufruf von »sock_recvmsg()« noch zwei Bemerkungen: Erstens braucht die Funktion einen Prozesskontext. Deshalb erzeugt Listing 1 einen Kernel-Thread (Zeile 70). Zweitens muss dafür gesorgt sein, dass man in den vorgesehenen Speicher schreiben darf. Da dieser im Kernelspace liegt, die Kopierfunktion aber vom Userspace ausgeht, ist die entsprechende Überprüfung auszuschalten. Dafür sorgt der Code der Zeilen 34 und 35. Der Aufruf »set_fs()« in Zeile 37 stellt den alten Wert wieder her.

Der Client, bitte

Ein UDP-Paket vom Kernel aus senden ist einfacher, als eins zu empfangen. Listing 3 zeigt dazu ein Kernelmodul, das die Aufgabe des vorgestellten Client-Programms übernimmt. Um das Modul kompakt zu halten, verschickt es das UDP-Paket gleich bei der Initialisierung. Der Code kann allerdings auch an fast jeder anderen Stelle im eigenen Kernelcode auftauchen. Zunächst erzeugt »sock_create()« den UDP-Socket, siehe Listing 3, Zeile 19. Dann folgt die Definition des Paketempfängers in der Struktur »struct sockaddr_in«.

Wie beim Empfang enthält auch beim Senden die Struktur »struct msghdr« alle wesentlichen Parameter (Abbildung 2). Sie besteht im Wesentlichen aus der Netzadresse des Empfängers (»struct sockaddr_in«) und der Speicheradresse des Sendepuffers (definiert über eine »struct iovec«). Das Feld »msg_flags« muss vor dem Senden auf »0« stehen.

Eine andere Variante, den Empfänger eines Paketes zu spezifizieren, macht die

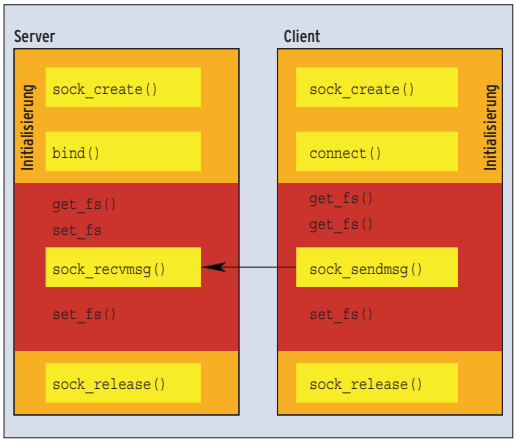


Abbildung 1: Zur Kommunikation über UDP bringt der Kernel sechs Netzwerk- und zwei Hilfsfunktionen mit.

Compile und Test

Der Code von Listing 1 lässt sich auf einem System mit Kernel 2.6 mit Hilfe des Makefile [2] übersetzen. Der Befehl »insmod udprcv.ko« lädt das resultierende Kernelmodul. Jetzt muss der Programmierer noch eine UDP-Nachricht an den Port 5555 des Rechners schicken, auf dem er das Modul geladen hat. Im Syslog dieses Rechners erfolgt dann eine Ausgabe, die die Nachricht selbst und den Absender enthält. Wichtig ist, auf diesem Wege nur Ascii-Strings zu schicken, da das Programm nicht überprüft, ob es sich um Binärdaten handelt.

Vom Userspace testen

Für einen Test eignet sich beispielsweise die Applikation aus Listing 2. Sie sendet ein einfaches „Hallo vom User-Space" an Port 5555 auf dem eigenen Rechner (IP-Adresse »localhost«, »127.0.0.1«). Die Testapplikation lässt sich jetzt am einfachsten durch den Aufruf von »make apudpsend« generieren. Es handelt sich ja um eine normale Applikation, sodass kein besonderes Makefile benötigt wird. Vor dem Aufruf der Applikation muss noch sichergestellt sein, dass der UDP-Port 5555 nicht durch eine Firewall blockiert ist.

Software zum Zeitpunkt des Verschickens besonders performant: Der Eintrag der Adresse der »struct sockaddr_in«, die die Empfänger-IP-Adresse und Empfänger-Portnummer enthält, erfolgt nicht in der »struct msghdr«. Stattdessen bindet sie der Aufruf der Methode »connect()« an den Socket. Das Feld »msg_name« setzt man dann vor dem Aufruf von »sock_sendmsg()« auf »0«, siehe Listing 4. Um Missverständnissen vorzubeugen: UDP ist und bleibt ein verbindungsloses Protokoll.

Die Methode »connect()« bindet in diesem Fall nur die Adresse des Empfängers an den Port und führt keinen wirklichen Verbindungsaufbau durch. Auch dieses Modul lässt sich mit Hilfe des Makefile [2] übersetzen und mit »insmod udpsend.ko« zusätzlich zum bereits geladenen Modul laden. Bei Erfolg erscheint im Syslog eine Meldung über den Empfang des gesendeten »hallo«. Natürlich ist es möglich, das Modul auch auf einem an-

Listing 1: UDP-Server »udprcv.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/in.h>
04 #include <net/sock.h>
05
06 #define SERVER_PORT 5555
07 static struct socket *udpsocket=NULL;
08 static DECLARE_COMPLETION( threadcomplete );
09 static int com_thread_pid;
10
11 static int com_thread( void *data )
12 {
13   struct sockaddr_in client;
14   struct sockaddr *address;
15   unsigned char buffer[100];
16   int len;
17   struct msghdr msg;
18   struct iovec iov;
19   mm_segment_t oldfs;
20
21   daemonize( "udpserver" );
22   allow_signal( SIGTERM );
23   if( udpsocket->sk==NULL )
24     return 0;
25   msg.msg_name = &client;
26   msg.msg_namelen = sizeof( struct
       sockaddr_in );
27   msg.msg_control = NULL;
28   msg.msg_controllen = 0;
29   msg.msg_iov     = &iov;
30   msg.msg_iovlen = 1;
31   while( !signal_pending(current) ) {
32     iov.iov_base = buffer;
33     iov.iov_len  = sizeof(buffer);
34     oldfs = get_fs();
35     set_fs( KERNEL_DS );
36     len = sock_recvmsg( udpsocket, &msg,
          sizeof(buffer), 0 );
37     set_fs( oldfs );
38     if( len>0 ) {
39       address = (struct sockaddr *)&client;
40       printk(KERN_INFO "msg \"%s\" from
            %u.%u.%u.%u\n", buffer,
41         (unsigned char)address->sa_data[2],
42         (unsigned char)address->sa_data[3],
43         (unsigned char)address->sa_data[4],
44         (unsigned char)address->sa_data[5] );
45     }
46   }
47   complete( &threadcomplete );
48   return 0;
49 }
50
51 static int __init server_init( void )
52 {
53   struct sockaddr_in server;
54   int servererror;
55
56   if( sock_create( PF_INET,SOCK_DGRAM,
         IPPROTO_UDP,&udpsocket)<0 ) {
57     printk( KERN_ERR "server: Error creating
          udpsocket.\n" );
58     return -EIO;
59   }
60   server.sin_family = AF_INET;
61   server.sin_addr.s_addr = INADDR_ANY;
62   server.sin_port = htons( (unsigned
       short)SERVER_PORT );
63   servererror = udpsocket->ops->bind(
        udpsocket,
64     (struct sockaddr *) &server, sizeof(
       server ) );
65   if( servererror ) {
66     sock_release( udpsocket );
67     return -EIO;
68   }
69
70   com_thread_pid=kernel_
      thread(com_thread,NULL, CLONE_KERNEL );
71   if( com_thread_pid < 0 ) {
72     sock_release( udpsocket );
73     return -EIO;
74   }
75   return 0;
76 }
77
78 static void __exit server_exit( void )
79 {
80   if( com_thread_pid ) {
81     kill_proc( com_thread_pid, SIGTERM, 0 );
82     wait_for_completion( &threadcomplete );
83   }
84   if( udpsocket )
85     sock_release( udpsocket );
86 }
87
88 module_init( server_init );
89 module_exit( server_exit );
90 MODULE_LICENSE("GPL");
```

deren Rechner zu installieren. Dann ist jedoch an Stelle von »127.0.0.1« die richtige IP-Adresse des Empfängers im Sourcecode einzutragen.

Socket-Parameter

Die höheren Ligen der Netzwerk-Programmierung beginnen, wenn beispielsweise mehrere Applikationen gleichzeitig auf ein und demselben Port warten oder wenn es um den Empfang von Multicast- oder Broadcast-Nachrichten geht. Der Applikationsprogrammierer weiß: Mit Hilfe von »setsockopt()« und »getsockopt()« lassen sich die dafür notwendigen Parameter einstellen.

Innerhalb des Kernels stehen entsprechende Funktionen zur Verfügung: »sock_setsockopt()« und »sock_getsockopt()«. Diese Funktionen erwarten ihr Argument im Userspace. Die Verwendung der Makros »set_fs()« und »get_fs()« – falls die Funktionen direkt im Kernel aufgerufen werden – ist damit obligatorisch (siehe [3]).

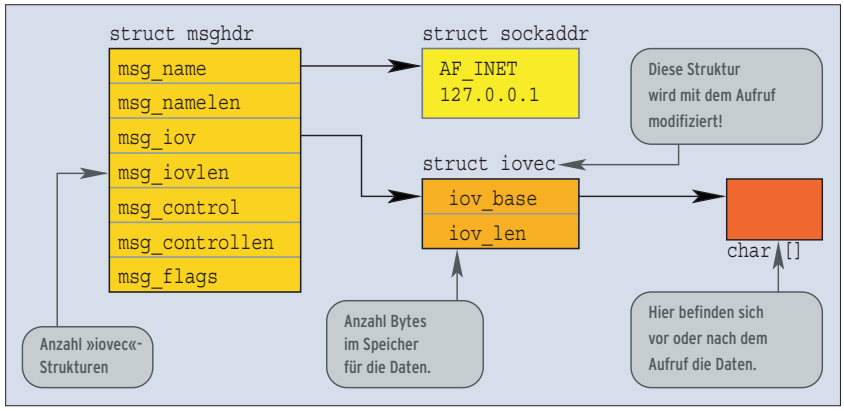


Abbildung 2: Die Datenstruktur vom Typ »struct msghdr« speichert die wesentlichen Parameter für das Senden und Empfangen der UDP-Pakete.

Allerdings gibt es für viele Optionen eine effizientere Variante. Vielfach wird nämlich mit dem Aufruf von »sock_setsockopt()« nur ein Element oder auch nur ein Flag der Socket-Datenstruktur gesetzt oder gelöscht. Ein Element liest oder schreibt man im Kernel aber besser direkt. Für das Setzen eines Flags kann der Programmierer auf die Inline-Funktionen (definiert in »net/sock.h«) »sock_set_flag()«, »sock_reset_flag()« und »sock_valbool_flag()« zurückgreifen.

Der bei diesen Makros verwendete erste Parameter »struct sock« ist Teil von

Listing 2: »apudpsend.c«

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <string.h>
05 #include <sys/types.h>
06 #include <sys/socket.h>
07 #include <netinet/in.h>
08 #include <arpa/inet.h>
09
10 int main( int argc, char **argv )
11 {
12   struct sockaddr_in to;
13   char *buf;
14   int sd, ret;
15
16   sd = socket( AF_INET, SOCK_DGRAM, 0 );
17   if( sd <= 0 ) {
18     perror( "socket" );
19     exit( -1 );
20   }
21   bzero( &to, sizeof(to) );
22   to.sin_family = AF_INET;
23   inet_aton( "127.0.0.1", &to.sin_addr );
24   to.sin_port = htons( (unsigned short)5555 );
25   buf = "Hallo vom User-Space";
26   ret = sendto( sd, buf, strlen(buf)+1, 0,
        (struct sockaddr *)&to,
27                 sizeof(to) );
28   close( sd );
29   return 0;
30 }
```

Listing 3: UDP-Client »udpsend.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/in.h>
04 #include <linux/inet.h>
05 #include <net/sock.h>
06
07 #define SERVERPORT 5555
08 static struct socket *clientsocket=NULL;
09
10 static int __init client_init( void )
11 {
12   int len;
13   char buf[64];
14   struct msghdr msg;
15   struct iovec iov;
16   mm_segment_t oldfs;
17   struct sockaddr_in to;
18
19   if( sock_create( PF_INET,SOCK_DGRAM,
       IPPROTO_UDP,&clientsocket)<0 ) {
20     printk( KERN_ERR "server: Error creating
          clientsocket.\n" );
21     return -EIO;
22   }
23   to.sin_family = AF_INET;
24   to.sin_addr.s_addr = in_aton( "127.0.0.1" );
       /* destination address */
25   to.sin_port = htons( (unsigned short)
       SERVERPORT );
26
27   msg.msg_name = &to;
28   msg.msg_namelen = sizeof(to);
29   memcpy( buf, "hallo", 6 );
30   iov.iov_base = buf;
31   iov.iov_len  = 6;
32   msg.msg_control = NULL;
33   msg.msg_controllen = 0;
34   msg.msg_iov    = &iov;
35   msg.msg_iovlen = 1;
36
37   oldfs = get_fs();
38   set_fs( KERNEL_DS );
39   len = sock_sendmsg( clientsocket, &msg, 6 );
40   set_fs( oldfs );
41   if( len < 0 )
42     printk( KERN_ERR "sock_sendmsg returned:
         %d\n", len);
43   return 0;
44 }
45
46 static void __exit client_exit( void )
47 {
48   if( clientsocket )
49     sock_release( clientsocket );
50 }
51
52 module_init( client_init );
53 module_exit( client_exit );
54 MODULE_LICENSE("GPL");
```

»struct socket«, siehe »net.h«. Um beispielsweise mehrere Kernel-Threads gleichzeitig auf einem Socket warten zu lassen, ist nach »sock_create()« folgender Code einzufügen:

```
oldfs = get_fs();
set_fs(KERNEL_DS);
optval = 1;
sock_setsockopt(sock,SOL_SOCKET, ➥
  SO_REUSEADDR,&optval,sizeof(int));
set_fs(oldfs);
```

Alternativ erfüllt eine einzige Zeile denselben Zweck:

```
socket->sk->sk_reuse = 1;
```

Um Broadcast-Pakete senden oder empfangen zu können, sind noch Optionen zu setzen:

```
oldfs = get_fs();
set_fs(KERNEL_DS);
optval = 1;
sock_setsockopt(sock,SOL_SOCKET, ➥
  SO_BROADCAST,&optval,sizeof(int));
set_fs(oldfs);
```

Als Alternative für diesen Fall kommt noch in Frage:

```
sock_valbool_flag(sock, SOCK_BROADCAST, 1);
```

Damit nimmt der Kernel auch Pakete mit Broadcast-Adressen an.

Vorschau

Mit den vorgestellten Techniken lassen sich beliebige, verbindungslose Netzwerkfunktionen im Kernel programmieren, ob als Server oder Client. Die nächste Folge der Kern-Technik-Reihe beschäftigt sich mit der darunter liegenden Schicht des Netzwerk-Stacks und erklärt, wie Treiber für Netzwerkadapter aufgebaut sind. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 15: Linux-Magazin 10/04, S. 114.

[2] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/11/Kern-Technik]

[3] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 14: Linux-Magazin 9/04, S. 92

[4] Quellcode zur Funktion »sock_setsockopt()«: [http://lxr.linux.no/source/net/core/sock.c?v=2.6.8.1#L183]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr gemeinsames Buch zum Kernel 2.6 heißt „Linux Treiber entwickeln“.

Listing 4: Optimierung

```
01 to.sin_family = AF_INET;
02 to.sin_addr.s_addr = in_aton( "127.0.0.1" );
03 to.sin_port = htons( (unsigned short)serverport );
04 errorcode=clientsocket->ops->connect(clientsocket,
     (struct sockaddr *)&to,sizeof(to),0);
05 msg.msg_name = NULL;
```

TCP/UDP/IP-Schnittstellenfunktionen im Kernel

Socket-Parametrierung	
`int sock_setsockopt(struct socket ➥ *sock, int level, int optname, char ➥ __user *optval, int optlen)` `int sock_getsockopt(struct socket ➥ *sock, int level, int optname, char ➥ __user *optval, int __user *optlen)`	Mit der Funktion »sock_setsockopt()« wird der Parameter »optname« des Socket »sock« mit dem Wert »optval« belegt, dessen Größe »optlen« festlegt. Die Funktion »sock_getsockopt()« liest den Parameter »optname« des Socket »sock« aus, wobei der Parameterwert in »optval« steht. Bei Aufruf der Funktion enthält die Variable »optlen« (deren Adresse übergeben wird) die Größe des mit »optval« zur Verfügung stehenden Speicherbereichs. Nach dem Aufruf ist »optlen« mit der Größe der kopierten Daten belegt. Der Kernel erwartet, dass »optval« und »optlen« im Userspace liegen. Wer die Funktionen innerhalb des Kernels aufruft, muss die Makros »get_fs()« und »set_fs()« einsetzen. Der Parameter »level« bestimmt die Protokollebene. Alle Parameter beziehen sich auf die Ebene »SOL_SOCKET«. Die Funktionen entsprechen den Applikationsfunktionen »setsockopt()« und »getsockopt()« (siehe dazu »man setsockopt«). Die verfügbaren Parameter lassen sich am einfachsten der Datei »/usr/src/linux/net/core/sock.c« entnehmen. Oft ist es effizienter, an Stelle der Funktionen »sock_setsockopt()« und »sock_getsockopt()« direkt die jeweils zugehörigen Codefragmente zu verwenden (siehe [4]). Im fehlerfreien Fall geben die Funktionen »0« zurück, sonst einen negativen Fehlercode.
`inline void sock_set_flag(struct sock ➥ *sk, enum sock_flags flag)` `inline void sock_reset_flag(struct ➥ sock *sk, enum sock_flags flag)` `inline void sock_valbool_flag(struct ➥ sock *sk, int bit, int valbool)`	Die in der Headerdatei »net/sock.h« definierten Inline-Funktionen dienen zum Setzen beziehungsweise Zurücksetzen des Flag »flag« von »sk«. Mit diesen Funktionen sind die Flags »SOCK_DEAD«, »SOCK_DONE«, »SOCK_URGINLINE«, »SOCK_LINGER«, »SOCK_DESTROY«, »SOCK_ BROADCAST« und »SOCK_TIMESTAMP« ansprechbar. Mit der Inline-Funktion »sock_ valbool_flag« wird das Flag »bit« zurückgesetzt, falls »valbool« »0« ist, ansonsten wird »bit« gesetzt.
`inline int sock_flag(struct sock ➥ *sk, enum sock_flags flag)`	Die in der Headerdatei »net/sock.h« definierte Inline-Funktion liest das Flag »flag« des Sockets »sk« aus und gibt das Ergebnis der aufrufenden Funktion zurück.
Methoden des Socket-Objekts	
`int (*connect)(struct socket *sock,➥ struct sockaddr *vaddr, int ➥ sockaddr_len, int flags)`	Im Fall eines verbindungslosen Protokolls (»SOCK_DGRAM«, UDP) wird die Empfängeradresse »vaddr« an den Socket »sock« gebunden. Ist bei einem nachfolgenden »sock_sendmsg()« das Element »msg_name« der Struktur »struct msghdr« auf »NULL« gesetzt, wird das Paket an den Empfänger »to« geschickt. Bei einem nachfolgenden »sock_recvmsg()« werden nur Pakete vom angegebenen Absender »vaddr« akzeptiert. Im Fall eines verbindungsorientierten Protokolls (»SOCK_STREAM«, TCP) wird eine Verbindung zu »vaddr« aufgebaut. Ist »flags« nicht auf »O_NONBLOCK« eingestellt, ist der aufrufende Thread so lange blockiert, bis entweder die Verbindung hergestellt oder ein Fehler aufgetreten ist. Konnte »vaddr« erfolgreich an den Socket gebunden oder eine Verbindung aufgebaut werden, gibt die Funktion »0« zurück, sonst einen negativen Fehlercode.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 17

Kern-Technik

Der unüberschaubar große Zoo von Netzwerkgeräten spiegelt sich im Kernel in der Anzahl zugehöriger Treiber wider. Wie sie intern funktionieren, zeigt diese Folge der Kern-Technik. Ein virtuelles Netzwerk-Device dient als Einstieg für eigene Experimente. Eva-Katharina Kunst, Jürgen Quade



Wer den Umgang mit »ifconfig« und »route« gewohnt ist, findet sich in der Welt der Netzwerktreiber schnell zurecht, denn ihre Grundstruktur ist übersichtlich. Erst die Komplexität der Hardwarezugriffe lässt die Treiber zu umfangreichen Modulen anwachsen.

Real und virtuell

Die meisten Netzwerktreiber implementieren Zugriffe auf reale Kommunikationshardware. Doch nicht selten sind auch virtuelle Kommunikationsgeräte im Einsatz, zum Beispiel das Loopback- oder das Dummy-Device.

Egal ob virtuell oder real, ein Netzwerktreiber besteht im Wesentlichen aus acht Funktionen: der Treiber-Initialisierung und -Deinitialisierung, der Geräte-Initialisierung und -Deinitialisierung, dem Öffnen und Stoppen des Netzwerkinterface, dem Senden und schließlich dem Empfang eines Pakets. Die ersten vier Funktionen integrieren den Treiber in die Kernelinfrastruktur und initialisieren die Hardware, siehe [1]. Die vier übrigen Funktionen (Open, Stopp, Senden und Empfangen) stellen den Kern des Treibers dar. Sie implementieren das so genannte Netzwerk-Interface, also den Teil, über den Anwendungen respektive das Netzwerk-Subsystem auf die Hardware zugreifen.

Aus Sicht des Programmierers ist das Netzwerk-Interface ein zu erzeugendes Objekt. Meist erledigt er das bei der Treiber-Initialisierung. Die Funktion »alloc_netdev()« erzeugt und initialisiert ein solches Objekt vom Typ »struct net_device«. Dabei gilt es, eine ganze Reihe von Feldern sinnvoll zu belegen. Da dies schnell unübersichtlich wird, schlagen die Kernelentwickler vor, die Initialisierung in eine eigene Setup-Funktion auszulagern. Ihre Adresse übergibt man der Funktion »alloc_netdev()« als Parameter. Das Netzwerk-Subsystem ruft die Setup-Funktion auf, nachdem es Speicher für das Netzwerk-Interface reserviert hat.

Zur Initialisierung des Interface gibt es aber noch mehr Hilfe. So existieren fertige Funktionen, die zu einem spezifischen Übertragungsmedium gehörige Felder sinnvoll belegen. Für Ethernet beispielsweise heißt diese Funktion »ether_setup()«, für FDDI »fddi_setup()«. Der Programmierer ruft in seiner Setup-Funktion zuerst »ether_setup()« auf und belegt dann die übrigen Felder der Struktur »struct net_dev«. Insbesondere trägt er hier die Adressen der grundlegenden Treiberroutinen (Open, Stopp und Senden) ein.

Netzwerk-Interfaces

Die Funktion »alloc_netdev()« erwartet neben der Adresse der Setup-Funktion einen Formatstring, der den Namen des Netzwerk-Interface festlegt. Da mehrere gleiche oder auch unterschiedliche Treiber gleichnamige Interfaces anlegen können, unterscheiden Anwender und Kernel sie anhand einer laufende Nummer, zum Beispiel »eth0« und »eth1«. Der Format-String repräsentiert diese Nummer durch ein »%d«, für ein Ethernet-Interface also durch »eth%d«.

Die Funktion »alloc_netdev()« hat noch einen letzten Parameter: die Anzahl Bytes, die das Netzwerk-Subsystem zusätzlich am Ende der »struct net_device« reserviert, siehe Abbildung 1. Dieser Bereich steht für treiberspezifische und private Daten zur Verfügung.

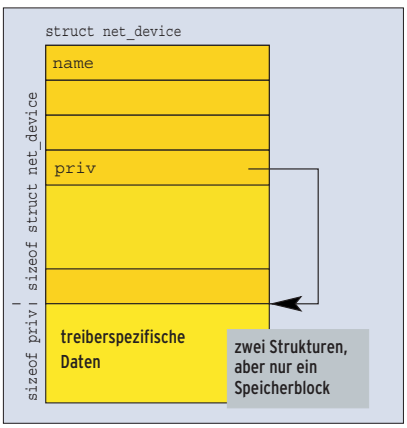


Abbildung 1: Platz für treiberspezifische Daten wird direkt im Anschluss an die Struktur »struct net_device« reserviert.

Für die beiden Speicherbereiche (»struct net_device« und für die privaten Daten) genügt es, einmal »kmalloc()« aufzurufen. Genauso gibt sie ein einziges »kfree()« wieder frei. Dabei muss sich niemand darum kümmern, ob der private Speicherbereich dynamisch oder statisch reserviert ist. Die so vorbereitete »struct net_device« übergibt man der Funktion »register_netdev()«. Damit meldet der Treiber während der Initialisierung das Netzwerk-Interface beim Subsystem an.

An- und Abmelden

Bei der Treiber-Deinitialisierung macht »unregister_netdev()« diese Registrierung wieder rückgängig. Die Funktion »free_netdev()« gibt das Interface-Objekt »struct net_device« frei.

Der Treiber eines echten Netzwerkadapters muss natürlich auch das Gerät selbst (de-)initialisieren. Wie das geht, hängt von der Hardware ab. Handelt es sich etwa um eine PCI-Karte, meldet sich der Treiber nicht nur beim Netzwerk-Subsystem, sondern auch beim PCI-Subsystem an, siehe [2]. Das wiederum ruft die zugehörige (De-)Initialisierungsfunktionen im Treiber auf.

Bei alter ISA-Hardware kann auch das Netzwerk-Subsystem die Geräte-Initialisierung auslösen. In diesem Fall ist im Element »init« der »struct net_device« die Funktionsadresse der Geräte-Initialisierung und im Element »uninit« jene der Deinitialisierungs-Routine abzulegen. Den zugehörigen Code führt das Netzwerk-Subsystem nach Aufruf der Setup-Funktion aus.

Da der Empfang von Paketen bei realer Hardware fast ausschließlich per Interrupt signalisiert wird, braucht jeder Treiber eine Interrupt-Service-Routine. Ist der Treiber beim Netzwerk-Subsystem angemeldet und die Hardware initialisiert, steht das neue Interface bereit. Es wird nicht über eine Major- und Minornummer identifiziert, sondern über den Namen, zum Beispiel »eth0«.

Anwendungen benutzen die Treiberfunktionen nicht direkt, sondern über Wrapper-Funktionen des Kernels. Die Methoden Open und Stopp ruft der Kernel auf, wenn der Anwender das Kommando »ifconfig up« respektive »ifconfig down« einsetzt, siehe Abbildung 2. Ist der Treiber dazu bereit, vom Netzwerk-Subsystem Pakete zu empfangen und zu verschicken, ruft er die Funktion »netif_start_queue()« auf. Diesen Aufruf findet man meist in den Open-Funktionen von Treibern. Bei Erfolg geben solche Funktionen »0« zurück, sonst einen negativen Fehlercode.

Die Stopp-Methode deaktiviert das Netzwerk-Interface. Der Aufruf von »netif_stop_queue()« sorgt dafür, dass der Kernel dem zugehörigen Interface nicht länger Aufträge zum Verschicken oder Empfangen erteilt.

Datentransfer

Die Hauptfunktionen des Treibers bestehen im Senden und Empfangen der Datenpakete. Die Schnittstelle zwischen Treiber und Netzwerk-Subsystem besteht im Wesentlichen aus den Socket-Puffern »struct sk_,buff«, deklariert in »linux/skbuff.h«. Über diese Datenstruktur haben die Instanzen Zugriff auf Nutzdaten und Verwaltungsinformationen. Falls das Netzwerk-Subsystem ein Paket senden will, ruft es die in der »struct net_device« angegebene Sendefunktion auf und übergibt ihr einen Socket-Puffer.

Im Normalfall entnimmt der Treiber das dort abgelegte Paket und kopiert es ohne weitere Modifikation in die Hardware. Netzwerkadapter besitzen dafür eigenen Speicher (Sende- und Empfangspuffer). Die Sendefunktion quittiert die erfolgreiche Übergabe des Pakets an die Hardware durch den Rückgabewert »0«. Konnte das Paket nicht übergeben werden, gibt die Funktion »1« zurück.

Zeitstempel für Timeouts

Falls nach dem Kopieren kein Sendepuffer mehr frei ist, stoppt man den Versand mit »netif_stop_queue()«. Wird ein Sendepuffer frei, aktiviert »netif_start_queue()« den Versand wieder. Nach dem Kopieren der Daten speichert das Subsystem die aktuelle Zeit (im Kernelmaß Jiffies; Listing 1, Zeile 38). Diesen Zeitstempel nutzt es zum Beispiel, um Time-

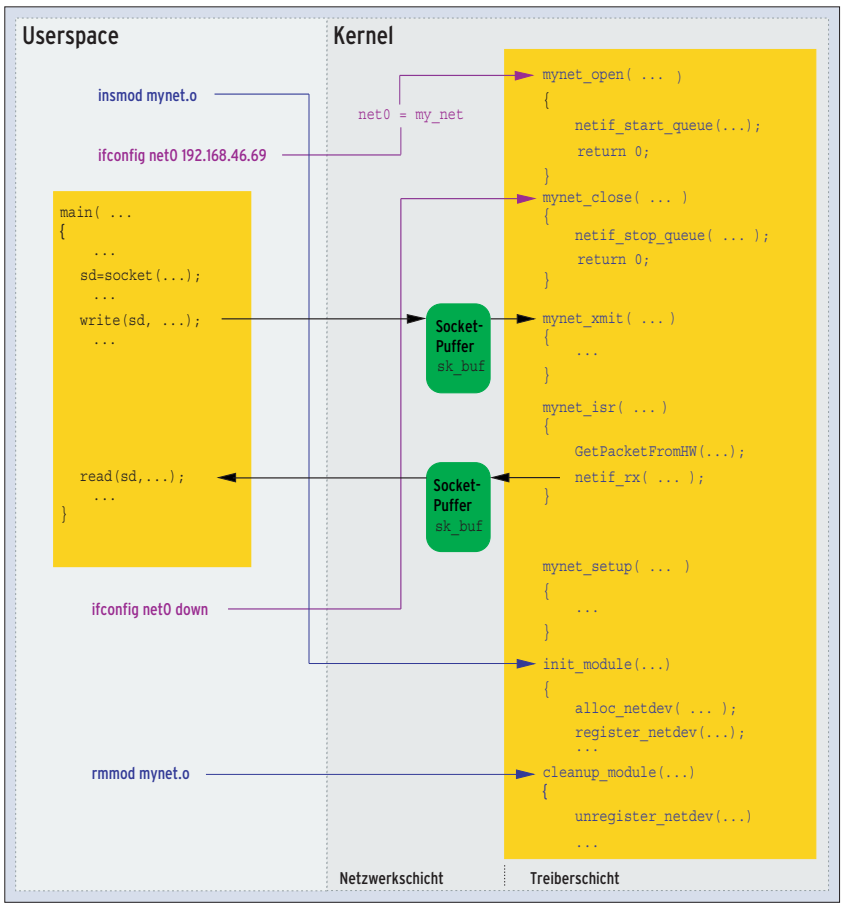


Abbildung 2: Funktionen und Befehle im Userspace rufen ihre Gegenstücke im Kernel auf.

outs zu überwachen, siehe Kasten „Timeout“. Sobald die Hardware signalisiert, dass sie das Paket verschickt hat, aktualisiert der Treiber die Sendestatistik (Listing 1, Zeilen 39 und 40) und gibt den Socketpuffer frei (Listing 1, Zeile 37). Bei älteren Karten kann man leider nur spekulieren, dass mit der Übergabe des Pakets an die Hardware der Versand erfolgreich war. Dabei laufen in der Sendefunktion die Aktualisierung der Statistik und die Socket-Freigabe ab.

Rückmeldung per Interrupt

Moderne Karten melden das Ergebnis per Interrupt. In diesem Fall protokolliert die Interrupt-Service-Routine (ISR) die statistischen Daten, wobei sie bei fehlgeschlagenem Versand statt der Sendestatistik die Fehlerstatistik aktualisiert.

Empfängt der Netzwerkadapter ein Paket, löst er normalerweise auch einen Interrupt aus. In diesem Fall muss der Treiber einen Socket-Puffer erzeugen, also »dev_alloc_skb()« oder »alloc_skb()« mit dem Parameter »gfp_mask« als »GFP_ATOMIC«.

Die Hardware kopiert die Daten in den Socket-Puffer und passt dessen Zeiger auf die Daten an. Im Socket-Buffer muss außerdem stehen, um welchen Protokolltyp es sich handelt und über welches Gerät die Daten entgegengenommen wurden. Gängige Ethernet-Protokolltypen sind beispielsweise ARP »ETH_P_ARP« oder IP »ETH_P_IP«. Den Protokolltyp bestimmt bei Ethernet-Adaptern die Funktion »eth_type_trans()«, deren Rückgabewert der Treiber ins Protokollfeld des Socket-Puffers einträgt, Listing 1, Zeile 23.

Das empfangene und im Socket-Puffer aufbereitete Paket übergibt man dem Netzwerk-Subsystem mit der Funktion »netif_rx()«, die das Paket im Prozess-Kontext weiter verarbeitet. Auch die Freigabe des Socket-Puffers erfolgt durch das Subsystem, nicht durch den Treiber.

Bleibt nur noch die Statistik zu aktualisieren (Listing 1, Zeilen 24 und 25) und den Empfangszeitstempel abzulegen (Zeile 26). Für die Statistik besitzt der Kernel eine eigene Datenstruktur. Der Anwender ruft die hier gesammelten Informationen etwa durch den Aufruf von »ifconfig« ab. Die Ausgabe der Informationen ist standardisiert, sodass der Entwickler in der Methode »get_stats()« nur die Adresse der Datenstruktur zurückgeben muss (Zeile 58).

Listing 1 zeigt den Code für ein Netzwerk-Nulldevice. Es wirft mangels realer Hardware alle zu verschickenden Pakete einfach weg. Es besitzt weder eine Geräte-Initialisierung, noch -Deinitialisierung. Die vollständige Version auf dem Linux-Magazin-Server [3] enthält eine Empfangsfunktion als Interrupt-Service-Routine, die über die Präprozessor-Direktive »#if 0« auskommentiert ist, denn ohne Hardware gibt es auch keine Interrupts. Der Treiber lässt sich mit dem Makefile von [3] übersetzen und mit »insmod net.ko« laden.

Das implementierte Netzwerk-Interface trägt den Namen »net0«. Mit »ifconfig net0 10.10.10.10« weist man ihm eine Beispiel-IP-Adresse zu. Es bringt allerdings nichts, vom eigenen Rechner aus auf die Adresse »10.10.10.10« zuzugreifen, da der Kernel lokale Zugriffe direkt

Timeout

Der Treiber kann das Verschicken von Daten über das Netzwerk-Interface zeitlich überwachen lassen. Dazu muss er in die »struct net_device« die Adresse einer Watchdog-Funktion und die Zeit eintragen, nach der die Watchdog-Funktion aufgerufen werden soll:

```
dev->tx_timeout = my_net_watchdog;
dev->watchdog_timeo = 5 * HZ;
```

Die Funktion »void my_net_watchdog(struct net_device *dev)« führt der Kernel aus, falls der Treiber nach dem letzten Senden die Funktion »netif_stop_queue()« aufgerufen hat und seither eine Zeit von »watchdog_timeo« vergangen ist.

Listing 1: Null-Device »net.c«

```
01 #include <linux/fs.h>
02 #include <linux/module.h>
03 #include <linux/netdevice.h>
04 #include <linux/init.h>
05
06 static struct net_device *my_net;
07 ...
23     skb->protocol = eth_type_trans(skb, dev);
24     stats->rx_packets++;
25     stats->rx_bytes += packet_length;
26     dev->last_rx = jiffies;
27 ...
33 static int my_net_send(struct sk_buff *skb,
   struct net_device *dev)
34 {
35     struct net_device_stats *stats = dev->priv;
   //netdev_priv(dev);
36
37     kfree_skb( skb );
38     dev->trans_start = jiffies;
39     stats->tx_bytes += skb->len;
40     stats->tx_packets++;
41     return 0;
42 }
43
44 static int my_net_open(struct net_device *dev)
45 {
46     netif_start_queue(dev);
47     return 0;
48 }
49
50 static int my_net_close(struct net_device *dev)
51 {
51     netif_stop_queue(dev);
53     return 0;
54 }
55
56 static struct net_device_stats
   *my_net_get_stats( struct net_device *dev )
57 {
58         return dev->priv;
59 }
60
61 static void __init my_net_setup( struct
   net_device *dev )
62 {
63     ether_setup(dev);
64     dev->open = my_net_open;
65     dev->stop = my_net_close;
66     dev->hard_start_xmit= my_net_send;
67     dev->get_stats = my_net_get_stats;
68     dev->flags |= IFF_NOARP;
69 }
70
71 static int __init net_init(void)
72 {
73     if( (my_net=alloc_netdev(sizeof(struct
   net_device_stats),
74             "net%d",my_net_setup))==NULL )
75         return -ENOMEM;
76     return register_netdev(my_net);
77 }
78
79 static void __exit net_exit(void)
80 {
81     unregister_netdev(my_net);
82     free_netdev(my_net);
83 }
84
85 module_init( net_init );
86 module_exit( net_exit );
87 MODULE_LICENSE("GPL");
```

abfängt und gar nicht an den Treiber weiterreicht. Ein Ping auf die Adresse »10.10.10.1« ruft wie gewünscht den Treiber auf. Dass er aktiv ist, lässt sich nun durch Aufruf der Netzstatistik überprüfen: »ifconfig net0«.

Vorschau

Netzwerktreiber können recht komplex werden. Weil sich in Kernel 2.6 aber wenig geändert hat, sind ältere Dokumentationen noch aktuell. Wer tiefer einsteigen will, sollte sich zum Beispiel die passenden Abschnitte in den Büchern **[4]**, **[5]** und **[6]** anschauen und den Skeleton-Treiber »isa-skeleton.c« im Kernel-Quellcode (siehe **[7]**) studieren.

In anderen Kernelbereichen hat sich wesentlich mehr geändert. So bringt Version 2.6 Mechanismen für asynchrone Ein- und Ausgabe mit, die den Datentransport beschleunigt. Was sie können und wie man sie benutzt, zeigt die nächste Folge der Kern-Technik. *(ofr)* ■

Infos

[1] Quade, Kunst: „Linux-Treiber entwickeln", Dpunkt Verlag, Heidelberg, Juni 2004
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 3: Linux-Magazin 10/03, S. 81
[3] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/12/Kern-Technik/]
[4] Wehrle et al., „Linux Netzwerkarchitektur", (Kernel 2.4): Addison-Wesley 2002
[5] Beck et al., „Linux Kernelprogrammierung", (Kernel 2.4): Addison-Wesley, 6. Auflage, 2001
[6] Rubini, Corbet, „Linux Device Drivers", (Kernel 2.4): O'Reilly, 2. Auflage, 2001
[7] Template-Netzwerktreiber: [http://lxr.linux.no/source/drivers/net/isa-skeleton.c?v=2.6.5]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

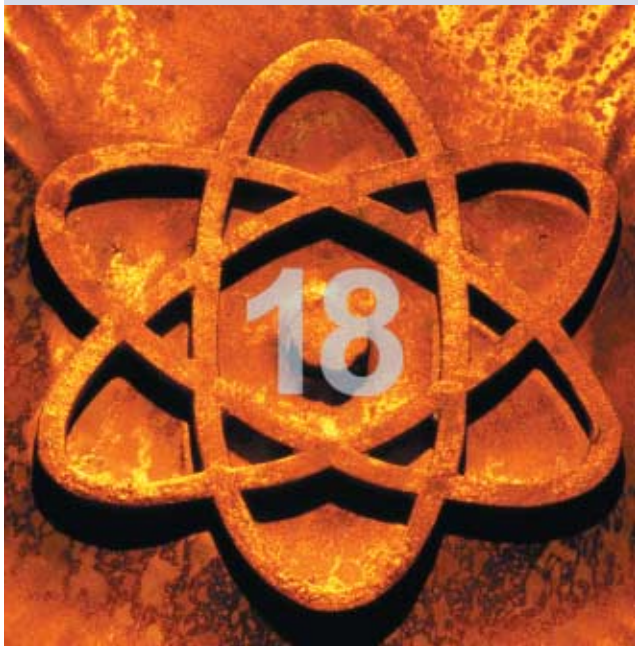
Gebräuchliche Funktionen von Netzwerktreibern

»struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct net_device *))« (in »linux/netdevice.h«)	Diese Funktion legt Speicher für ein Objekt »struct net_device« an. Zum Initialisieren des Speichers dient »setup()«. Das erzeugte Netzwerk-Interface spricht man später über den Namen »name« an. Zusätzlich zum Objekt reserviert die Funktion »sizeof_priv« Bytes für treiberspezifische (private) Daten. Der Pointer »priv« enthält die Startadresse dieser Daten. Konnte sie keinen Speicher allozieren, gibt die Funktion »0« zurück.
»struct net_device *alloc_etherdev(int sizeof_priv)« (in »linux/etherdevice.h«)	Die Funktion erzeugt ein Objekt vom Typ »struct net_device« und ruft dabei zur Initialisierung »ether_setup()« auf. Damit repräsentiert sie den folgenden Aufruf: »netstruct = alloc_netdev(sizeof_priv,„eth%d",ether_setup);«
»void free_netdev(struct net_device *dev)« (in »linux/netdevice.h«)	Löst die Referenz auf das Netzwerk-Interface »dev«. Falls es sich um die letzte Referenz handelt, gibt sie das Objekt frei. Mit »alloc_netdev()« zusätzlich angelegter Speicher wird ebenfalls verfügbar.
»int register_netdev(struct net_device *dev)« (in »linux/netdevice.h«)	Die Funktion registriert den durch die Struktur »dev« gegebenen Netzwerktreiber beim Betriebssystem. Ohne Fehler gibt die Funktion »0«, sonst einen Fehlercode zurück.
»void unregister_netdev(struct net _device *dev)« (in »linux/netdevice.h«)	Meldet das Netzwerk-Interface »dev« wieder beim Kernel ab.
»void ether_setup(struct net_device *dev)« (in »linux/netdevice.h«)	Diese Funktion initialisiert das Netzwerk-Interface »dev« als ein Ethernet-Interface.
»void netif_start_queue(struct net_device *dev)« (in »linux/netdevice.h«)	Mit dieser Inline-Funktion signalisiert der Treiber dem Netzwerk-Subsystem, dass das Netzwerk-Interface »dev« bereit ist Pakete entgegenzunehmen und weiterzureichen.
»void netif_stop_queue(struct net_device *dev)« (in »linux/netdevice.h«)	Über diese Inline-Funktion signalisiert der Treiber dem Netzwerk-Subsystem, dass das Netzwerk-Interface »dev« nicht in der Lage ist Pakete entgegenzunehmen und zu versenden.
»struct sk_buff *alloc_skb (unsigned int size, int gfp_mask)« (in »linux/skbuff.h«)	Diese Funktion reserviert Speicher für einen Socket-Puffer mit einem Datenpuffer der Länge »size«. Da unter Umständen innerhalb von »alloc_skb()« per »kmalloc()« Speicher alloziert wird, gibt »gfp_mask« an, ob die aufrufende Instanz schlafen gelegt werden darf oder nicht. Im Interrrupt-Kontext muss daher für »gfp_mask« der Wert »GFP_ATOMIC« stehen. In diesem Fall kann man alternativ die Funktion »dev_alloc_skb(unsigned int siz)« verwenden (entspricht einem Aufruf von »alloc_skb(size,GFP_ATOMIC)«). Konnte sie keinen Puffer reservieren, gibt die Funktion »0« zurück, sonst die Adresse des Socket-Puffers.
»void kfree_skb(struct sk_buff *skb)« (in »linux/skbuff.h«)	Die Funktion dekrementiert den Referenzzähler des Puffers »skb«. Wird der Socket-Puffer von keiner Instanz mehr verwendet, gibt sie ihn frei.
»unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)« (in »linux/etherdevice.h«)	Diese Funktion bestimmt den Protokolltyp des in »skb« enthaltenen Ethernet-Pakets. Außerdem stellt sie fest, ob es sich um ein Broadcast- oder ein Multicast-Paket handelt und ob das Paket überhaupt für den eigenen Rechner (»dev«) bestimmt ist. In »skb->pkt_type« wird das zugehörige Flag gesetzt.
»int netif_rx(struct sk_buff *skb)« (in »linux/netdevice.h><«)	Mit Hilfe dieser Funktion übergibt ein Treiber das empfangene Paket »skb« dem Netzwerk-Subsystem zur Weiterverarbeitung. Die oberen Protokollschichten können bei hoher Last das Paket verwerfen. Folgende Rückgabewerte der Funktion gibt es: »NET_RX_SUCCESS«, »NET_RX_CN_LOW«, »NET_RX_CN_MOD«, »NET_RX_CN_HIGH« und »NET_RX_DROP«.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 18

Kern-Technik

Ob für Applikations- oder Treiberprogrammierer: Der asynchrone Zugriffsmodus im Kernel 2.6 verspricht Geschwindigkeitsvorteile bei der Ein- und Ausgabe. Dieser Artikel zeigt, wie Entwickler von Async-I/O profitieren, und untersucht, ob die Implementation die Versprechungen erfüllt. Eva-Katharina Kunst, Jürgen Quade



Schreibt eine Anwendung auf die Festplatte, sind Verzögerungen programmiert: Das langsame Medium bremst die ganze Applikation und macht anderweitig erzielte Geschwindigkeitsvorteile zunichte. Abhilfe soll in diesem Fall der asynchrone Zugriffsmodus (Async-I/O) schaffen: Während das Ein- oder Ausgabesystem noch mit dem Lesen oder Schreiben beschäftigt ist, bereitet die Applikation bereits den nächsten I/O-Auftrag vor (Abbildung 1).

Im Unterschied dazu besitzen klassische Unix-Systeme nur den synchronen Zugriffsmodus, bei dem eine Applikation auf das Ende der Ein- und Ausgabe wartet. Will ein Programm trotz eines laufenden Auftrags weiterarbeiten, muss es einen zweiten Thread starten. Nach diesem Prinzip funktioniert die Posix-Variante des asynchronen Zugriffs [1]. Deren Nachteil ist, dass mit jedem I/O-Auftrag der Scheduler aktiv werden muss. Unnötige Performance-Einbußen sind die Folge.

Neben der Posix-Lösung gab es für Linux 2.4 bereits Patches, die den gewünschten Zugriffsmodus direkt im Kernel realisieren. Kernel 2.6 hat ihn schließlich eingebaut. Dennoch setzen bisher nur wenige Applikationen Async-I/O ein, denn es ist kaum dokumentiert, wie sich diese Zugriffsart nutzen lässt. Außerdem bringt die Implementierung in Kernel 2.6 nur teilweise die erwarteten Performance-Vorteile.

Zudem ist die Implementierung nicht vollständig und ihre Interfaces sind nicht allgemein verwendbar. Kein Wunder, dass außer Oracle kaum jemand den asynchronen Zugriff verwendet und das OSDL potenzielle Nutzer sucht [2].

Neue Systemcalls

Als User-API für Async-I/O implementiert Linux insgesamt fünf Systemaufrufe: »io_setup()«, »io_destroy()«, »io_submit()«, »io_getevents()« sowie »io_cancel()«. Mit »io_setup()« erzeugt die Applikation im Kernel einen Async-I/O-Kontext. Diesem Kontext ordnet der Kernel später I/O-Aufträge zu, sodass Anwendungen den Auftragsfortschritt erfragen können. Ein so genannter »iocb« (Input Output Control Block) beschreibt den Auftrag.

Der Systemcall »io_submit()« übergibt dem Kernel die »iocb«-Blöcke zur Abarbeitung. Die Applikation muss ihrerseits für jeden Auftrag Speicher bereitstellen, aus dem der Kernel die Daten bei einem Auftrag entnimmt respektive dort ablegt.

Speicher ausrichten

Dieser Speicherbereich muss bei asynchronem Zugriff Sector-aligned sein, also auf eine Sektorgröße ausgerichtet (genauer: ein Vielfaches von 512). Den notwendigen Speicher reserviert am einfachsten die Funktion »int posix_memalign (void **memptr, size_t alignment, size_t size)«, siehe Listing 1, Zeilen 31 und 32. Die Funktion »io_cancel()« beendet einzelne Aufträge wieder.

Schließlich benötigen Applikation und Kernel noch eine Datenstruktur, die den Auftragsstatus verwaltet: »struct io_event«. Eine Liste dieser Event-Daten-

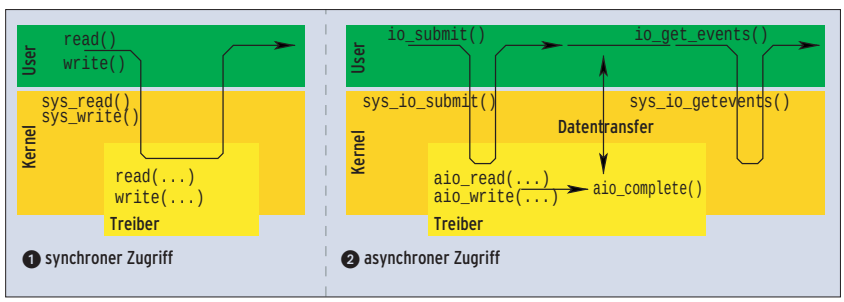


Abbildung 1: Beim synchronen Zugriff (❶) schläft die Applikation, bis der Treiber die Daten zum oder vom Speicher transferiert hat. Beim asynchronen Zugriff (❷) arbeiten Applikation sowie Treiber und Kernel parallel.

strukturen übergibt »io_getevents()« dem Kernel. Der Systemcall blockiert aufrufende Rechenprozesse so lange, bis eine minimale Anzahl von Aufträgen abgearbeitet oder die angegebene Timeout-Zeit abgelaufen ist.

Eine Unterstützung für die vorgestellten Systemcalls sucht man in der Standard-C-Bibliothek vergebens. Wer in seiner Applikation asynchron zugreifen möchte, muss die Aufrufe per »_syscall*X*«-Makro selbst programmieren (siehe [3]) oder die LibAIO herunterladen und installieren (siehe [4]).

User-Bibliothek: LibAIO

Dass es den Quellcode der Library nur als RPM-Paket gibt, ist dank »alien« auch für Debian-User kein großes Problem. Der Aufruf »alien libaio-0.3.102-1.src.rpm« erzeugt ein Verzeichnis »libaio-0.3.102« und legt darin das Archiv »libaio-0.3.102.tar.gz« ab. In dem mit »tar« extrahierten Verzeichnis »/usr/src/libaio-0.3.102« kompiliert und installiert der Befehl »make prefix=/usr install« die Bibliothek. Er kopiert auch die Headerdatei »libaio.h« an die richtige Stelle. Eine Beschreibung der wesentlichen Bibliotheksfunktionen und Makros gibt der Kasten „Applikationsfunktionen der LibAIO“.

Listing 1 zeigt eine einfache LibAIO-Applikation, die aus zwei Dateien jeweils einen Datenblock liest. Ein einziger Zugriff übergibt beide Lese-Aufträge dem Kernel. Das zugehörige Makefile, das besonders das Linken mit der LibAIO organisiert, ist unter [5] zu finden. Ein Test der Applikation benötigt die beiden Dateien /tmp/foo« und »/tmp/too«.

Kernel hilft

Abbildung 2 zeigt am Beispiel eines Lese-Auftrags, wie der Kernel die Systemcalls abarbeitet. Beim Aufruf von »io_submit()« führt er die Funktion »sys_io_submit()« aus. Sie zerlegt den Auftrag in einzelne Teile und reicht sie auch einzeln an »io_submit_one()« weiter. Diese Routine verwendet die Zugriffsfunktionen (»struct file_operations«) für die zum Auftrag gehörige Datei beziehungsweise für das entsprechende Gerät.

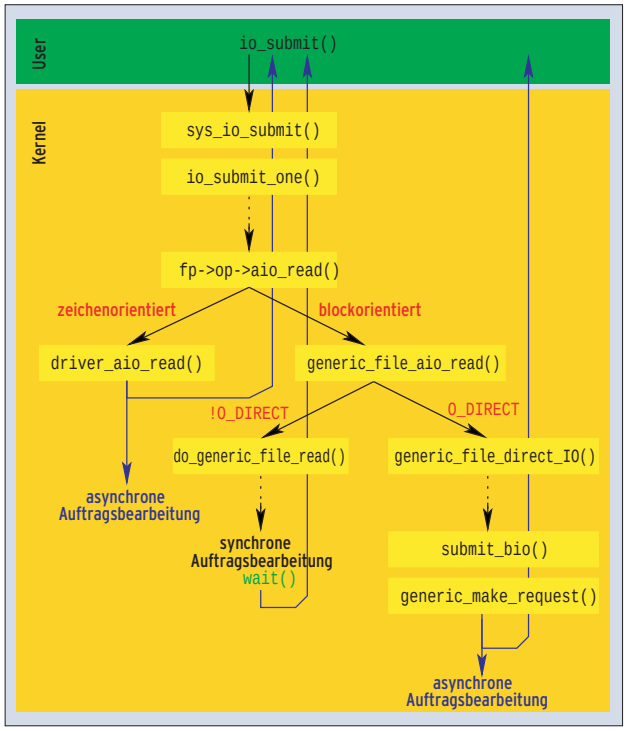


Abbildung 2: Nicht jeden mit »io_submit()« übergebenen Auftrag verarbeitet der Kernel asynchron. Aufträge ohne »O_DIRECT« laufen beispielsweise synchron ab.

Beim asynchronen Lesen ruft der Kernel die im »struct file_operations« abgelegte Zugriffsfunktion »aio_read()« auf. Unterstützt ein zeichenorientierter Treiber Async-I/O, hinterlegt er in dieser Struktur eine passende Funktion. Bei einem normalen Dateizugriff hat der Kernel hier bereits die Standardfunktion »generic_file_aio_read()« eingehängt.

Abbildung 2 zeigt auch, dass bei einem normalen Dateizugriff keine asynchrone Verarbeitung stattfindet. Stattdessen initiiert »do_generic_file_read()« einen synchronen Zugriff. Dabei werden die Daten nicht direkt zwischen Applikationsspeicher und Festplatte ausgetauscht, sondern über den Page-Cache. Die Funktion wartet bei einem Leseauftrag so lange, bis der Page-Cache gefüllt ist und

Listing 1: »aio_appl.c«

```
01 #define _GNU_SOURCE
02 #include <stdio.h>
03 #include <stdlib.h>
04 #include <fcntl.h>
05 #include <libaio.h>
06
07 #define BUF_SIZE (64*1024)
08 static char *buf, *buf2;
09
10 int main( int argc, char **argv )
11 {
12    io_context_t ctxp = NULL;
13    struct iocb iocb, iocb2;
14    struct iocb *iocbs[] = { &iocb, &iocb2 };
15    struct io_event event[2];
16    int fd1, fd2;
17    struct timespec ts;
18    int result;
19
20    if( io_setup( 1024, &ctxp ) ) {
21       perror( "io_setup" );
22       return -1;
23    }
24    fd1 = open( "/tmp/foo", O_RDWR|O_DIRECT );
25    //fd1 = open( "/tmp/aiodev", O_RDONLY );
26    fd2 = open( "/tmp/too", O_RDWR|O_DIRECT );
27    if( fd1<0 || fd2<0 ) {
28       perror("open");
29       return -2;
30    }
31    posix_memalign( (void**)&buf, 512, BUF_SIZE
   );
32    posix_memalign( (void**)&buf2, 512, BUF_SIZE
   );
33    io_prep_pread( &iocb,  fd1, buf,
   BUF_SIZE, 0 );
34    io_prep_pread( &iocb2, fd2, buf2, BUF_SIZE, 0
   );
35    io_submit( ctxp, 2, iocbs );
36    // ... do something else ...
37    ts.tv_sec = 5;
38    ts.tv_nsec= 0;
39    result = io_getevents( ctxp, 2, 2, &event[0],
   &ts );
40    printf("result= %d | result= %ld/%ld res2=
   %ld/%ld\n", result,
41       event[0].res, event[0].res2,
42       event[1].res, event[1].res2);
43    //printf("Inhalt von buf: %s\n", buf ); //
   Test mit "asynctest"
44    io_destroy( ctxp );
45    return 0;
46 }
```

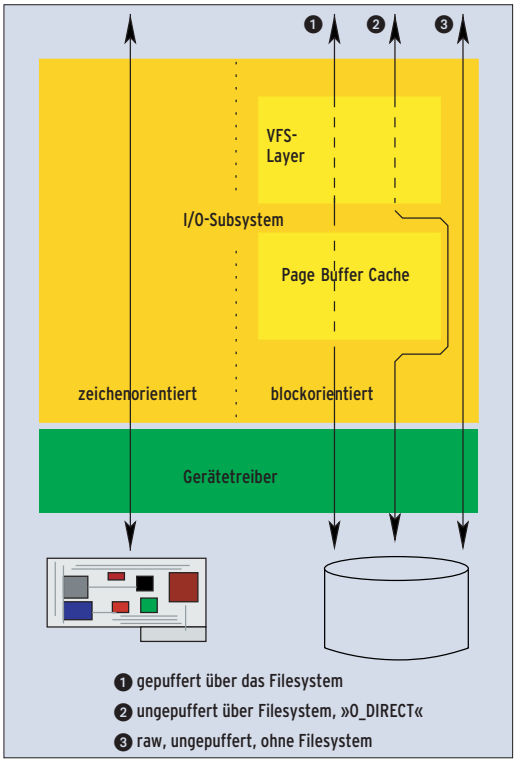


Abbildung 3: Neben dem direkten Zugriff auf zeichenorientierte Geräte gibt es beim Zugriff auf Block-Devices drei alternative Pfade.

die Daten in den Applikationsspeicher transferiert sind.

Asynchrone Verarbeitung findet also nur statt, wenn die Datei mit dem Flag »O_DIRECT« geöffnet wurde. Dann greift die Applikation ohne Page-Cache auf die Daten zu (siehe Abbildung 3) und der Kernel führt »generic_file_direct_IO()« aus, das »generic_make_request()« aufruft. An dieser Stelle greift er auf die Daten zu und beendet den Systemcall »io_submit()«.

Während sich der Treiber um den Datentransfer kümmert, arbeitet die Applikation weiter. Allerdings gibt es auch bei Verwendung von »O_DIRECT« eine Einschränkung: Nur die Dateisysteme Ext 2, Ext 3, XFS und JFS unterstützen Async-Direct-I/O [6]. Liegt die Datei auf einem anderen Filesystem, schaltet der Kernel stillschweigend auf synchronen Zugriff um.

Character-Devices

Abbildung 4 zeigt die Abläufe innerhalb des Kernels beim Zugriff auf ein zeichenorientiertes Gerät. Der Aufruf des Systemcalls »io_submit()« (1) legt im Kernel eine Kopie des »iocb« an, die Struktur »kiocb«. Sie beschreibt den Auftrag und dient der im Treiber kodierten Funktion »aio_read()« als Argument (3). Falls »aio_read()« den Auftrag direkt (also synchron) bearbeiten kann, gibt diese Treiberfunktion »0« zurück und die Bearbeitung ist abgeschlossen.

Wahrscheinlicher ist der Fall, dass die Funktion »aio_read()« die Bearbeitung des Auftrags nur anstößt. Dann gibt sie den Wert »-EIOCBQUEUED« zurück (4). Problematisch bei der nun wirklich asynchron ablaufenden Verarbeitung ist, dass kein Userkontext mehr existiert. Stehen die angeforderten Daten bereit, kann man sie nicht einfach per »copy_to_user()« an die vorher übergebene Speicheradresse kopieren. Wahrscheinlich benutzt längst ein anderer Prozess diese Adresse.

Um dennoch Zugriff auf die Speicherbereiche der zugehörigen Applikation zu erhalten, bietet Async-I/O einen Ausweg: den Datentransfer im Kontext des »keventd« durchzuführen. Diese Lösung heißt in Kernel-Kreisen Kick-Interface.

Kicker im Kernel

Läuft die Verarbeitung im Keventd-Kontext ab, sorgt Async-I/O dafür, dass das Memory-Management zum entsprechenden Zeitpunkt die für den Auftrag korrekte Speicherzuordnung wiederherstellt. Um diese Möglichkeit zu nutzen, muss der Entwickler eine so genannte Retry-Funktion zur Verfügung stellen, die im Userkontext arbeitet. Ein Aufruf der Methode »kick_iocb()« aktiviert diese Retry-Funktion.

Sobald der (asynchron stattfindende) Datentransfer (5) abgeschlossen ist, ruft

Tabelle 1: Applikationsfunktionen der LibAIO

`long io_setup(unsigned nr_events, aio_context_t *ctxp)`	Erzeugt einen Async-I/O-Kontext mit dem maximal »nr_events« Aufträge überwacht werden können. Läuft alles glatt, gibt die Funktion »0« zurück.
`long io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbb)`	Übergibt dem Kernel die Liste »iocbb« mit »nr« Aufträgen zur asynchronen Bearbeitung. Tritt kein Fehler auf, gibt sie die Anzahl der übernommenen Aufträge zurück.
`long io_getevents(aio_context_t ctx_id, long min_nr, long nr, struct io_events *event, struct timespec *timeout)`	Blockiert den aufrufenden Thread so lange, bis mindestens »min_nr« Aufträge ausgeführt sind. »event« stellt eine Liste von »nr« Strukturen vom Typ »struct io_event« dar. Die maximale Wartezeit legt »timeout« fest. Das Ergebnis eines Auftrags ist dann in der jeweiligen »struct io_event« abzulesen. Hier gibt es für jeden abgeschlossenen Auftrag gleich zwei Rückgabewerte: »res« gibt die Anzahl der transferierten Daten an, die Bedeutung des Rückgabewerts »res2« ist nicht festgelegt und wird von den meisten Treibern auf »0« gesetzt.
`long io_cancel (aio_context_t ctx_id, struct iocb *iocb, struct io_event *result)`	Bricht den in »iocb« angegebenen Auftrag ab. Tritt kein Fehler auf, gibt die Funktion »0« zurück.
`long io_destroy (aio_context_t ctx)`	Gibt den Async-I/O-Kontext »ctx« wieder frei. Sind zum Zeitpunkt der Freigabe noch Aufträge in Bearbeitung, versucht der Kernel sie abzubrechen. Solange aber nicht alle Aufträge abgeschlossen sind, blockiert die Funktion. Sie gibt »0« zurück, wenn kein Fehler auftritt.
`void io_prep_pread(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)`	Diese Inline-Funktion initialisiert den »iocb« für einen Leseauftrag.
`void io_prep_pwrite(struct iocb *iocb, int fd, void *buf, size_t count, long long offset)`	Diese Inline-Funktion initialisiert den »iocb« für einen Schreibauftrag.

der Treiber die Funktion »aio_complete()« auf. Dies kann beispielsweise in der »retry()«-Funktion geschehen. Sie signalisiert dem Kernel, dass der Treiber den Auftrag abgeschlossen hat.

Die Methode »aio_complete()« erwartet zwei Rückgabewerte: Der erste steht für die Anzahl der übertragenen Zeichen, der zweite ist nicht weiter spezifiziert und wird üblicherweise gleich »0« gesetzt. Während der Kernel die Rückgabewerte in der IO-Event-Datenstruktur zwischenspeichert (6), kann der Treiber den »kiocb« wieder freigeben. Für ihn ist an dieser Stelle nämlich die Bearbeitung des Auftrags beendet.

Die Applikation kann zu jeder Zeit den Zustand des Auftrags über den Systemcall »io_getevents()« abfragen (7). Ist der Auftrag beendet, werden die in den Event-Strukturen abgelegten Rückgabewerte in den Userspace kopiert (8).

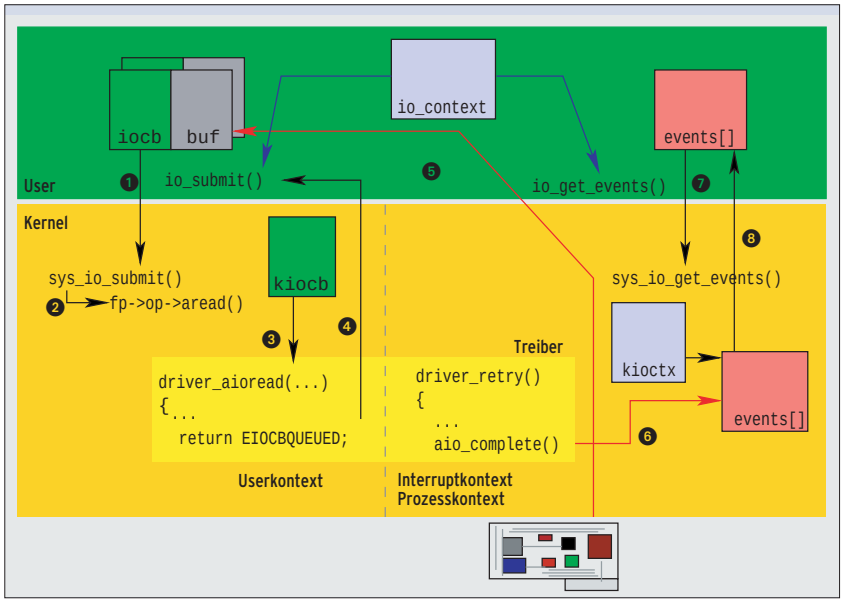


Abbildung 4: Bei Async-I/O muss der Zugriff auf die Speicherbereiche der Applikation im Interrupt- oder Prozesskontext stattfinden. Im Treiber sorgt »driver_retry()« fürs Abholen der angeforderten Daten.

Funktion zum Abbrechen

Wer in einem Treiber den asynchronen Zugriffsmodus unterstützen möchte, muss also im Wesentlichen die beiden Funktionen »aio_read()« und »aio_write()« implementieren. Falls der Treiber auch den Systemcall »io_cancel()« beherrschen soll, muss er zusätzlich eine Cancel-Funktion implementieren und die Adresse dieser Funktion im »kiocb« des Auftrags ablegen (siehe Listing 2, Zeile 47). Die Cancel-Funktion selbst führt zum einen die im Treiber notwendigen Aktionen zum Rückruf des Auftrags aus. Zum anderen kennzeichnet sie die Struktur »kiocb« als »cancelled« (abgebrochen). Beim Programmieren dieser Funktion hilft das Makro »kiocbSetCancelled()«.

Der in Listing 2 dargestellte Treiber implementiert nur die Lesefunktion, aber inklusive Cancel. Er simuliert das asynchron arbeitende Gerät durch eine Timerfunktion, die zwei Sekunden nach Auftragsvergabe die Daten in den Userspace kopiert. Um die Implementierung einfach zu halten, legt das Beispiel die Aufrufparameter »buf« und »count« in zwei globalen Variablen ab. Ein echter Treiber müsste diese Parameter in einer separaten Datenstruktur speichern. Übersetzen lässt sich der Treiber nur mit einem aktuellen Kernel (etwa Version 2.6.8.1). Das zugehörige Makefile findet sich unter [5]. Nach dem Übersetzen und Laden des Treibers fehlt noch eine Gerätedatei mit der Major-Nummer 240:

```
make
insmod asynctest.ko
mknod /tmp/aiodev c 240 0
```

Der Code lässt sich mit der Applikation aus Listing 1 testen. Es ist allerdings nö-

Kernelzugriff auf Speicherbereiche im Userspace

Mit ein paar Tricks lassen sich auch ohne Userkontext Daten zwischen Kernel- und Userspace austauschen. Dazu muss man wissen, dass Linux den gesamten Hauptspeicher in Speicherseiten (Pages) einer festen Größe, meist 4096 Bytes, organisiert (siehe dazu [8]). Folgende Aktionen sind im Kernel durchzuführen:

- Im Userkontext lockt der Programmierer jene Speicherseite des physischen Speichers, die die Applikation nutzt. Damit ist sicher, dass sich die Seite im physisch vorhandenen Hauptspeicher befindet. Außerdem benötigt der Treiber für den späteren Zugriff deren (physische) Adresse.
  Sowohl das Locking als auch die Identifikation der physischen Adresse sind über die Funktion »get_user_pages()« möglich. Neben der Page-Adresse wird später auch der Offset innerhalb der Seite benötigt. Dieser ist wie folgt zu berechnen (»buf« ist die virtuelle Adresse im Userspace):

```
offset = (unsigned long)buf &
~PAGE_MASK;
```

  Je nach zu übertragender Datenmenge muss der Programmierer mit »get_user_pages()« mehrere Seiten anfordern. Der Bedarf ist vorher zu berechnen und entsprechend viel Speicherplatz für die Zeiger auf die Pages zu reservieren (siehe [9]).
- Zum Zeitpunkt des Kopierens zwischen Kernel und Userspace braucht das Programm die gültige virtuelle (Kernel-) Adresse. Dazu dient die Funktion »page_address(struct page *)«. Der Zugriff selbst erfolgt mit Hilfe der Funktion »memcpy()«. Achtung: den Offset nicht vergessen! Außerdem ist sicherzustellen, dass sich die Speicherseiten nicht im Highmem befinden. Gegebenenfalls schafft »kmap()« hier Abhilfe. Falls Datenmengen kopiert werden, die sich über mehr als eine Page verteilen, ist das Kopieren für jede Seite durchzuführen.
- Die Speicherseite muss zuerst als verwendet markiert, später aber wieder freigegeben werden, sodass der Kernel sie bei Bedarf auslagern kann. Dazu dienen die Funktionen »SetPageDirty()« und »page_cache_release()«.

Das Markieren und Freigeben muss der Programmierer für jede zuvor gelockte Speicherseite durchführen.

tig, das Öffnen der Datei »/tmp/foo« auszukommentieren (Zeile 24) und dafür das Öffnen von »/tmp/aiodev« (Zeile 25) einzufügen. Wichtig ist, nicht das Flag »O_DIRECT« anzugeben.

Der in Listing 2 dargestellte Treiber nutzt das schon beschriebene Kick-Interface. Wer es vorzieht, direkt vom Treiber aus auf den Speicher der Applikation zuzugreifen, findet Hinweise dazu im Kasten „Kernelzugriff auf Speicherbereiche im Userspace".

Fazit

Insgesamt wird deutlich, dass die gegenwärtige Implementierung von Async-I/O im Kernel 2.6 noch nicht zufrieden stellend ist. Angefangen bei Schwächen des Userinterface bis zum problematischen direkten Datentransfer zwischen asynchron arbeitender Applikation und dem Kernel gibt es noch einige Optimierungsmöglichkeiten. Da Async-I/O zudem bis jetzt keine spektakulären Performance-Wunder vollbringt, ist die gegenwärtige Zurückhaltung der Anwender nicht überraschend.

Auch die bereits vorhandene Reimplementierung [7], die den Zugriff auf gepufferte Dateien ermöglicht, weckt wenig Hoffnung: Erste Performance-Tests waren alles andere als Erfolg versprechend und Linus Torvalds ist folglich zurückhaltend. Die grundlegenden Probleme scheinen damit nämlich nicht beseitigt zu sein. Vielleicht ist die seit jeher vorhandene synchrone Ein- und Ausgabe in Linux auch einfach zu gut implementiert.

Bei aller Kritik sollte man allerdings nicht übersehen: Die Datenbankhersteller sind durchaus zufrieden. Ihre Anwendungen können dank geschickter Programmierung spürbare Performance-Steigerungen verbuchen. Mit den Einschränkungen können sie gut leben, da sie ohnehin nur direkt auf die Daten zugreifen. *(ofr)* ■

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Infos

[1] Chinmay Albal, „Comparative study of the Asynchronous I/O programming interfaces on Linux": Proceedings of the 11th International Linux-Kongress 2004, Erlangen 2004

[2] Jonathan Corbet, „Kernel Summit: Asynchronous I/O": [http://lwn.net/Articles/94566/]

[3] Eva-Katharina Kunst und Jürgen Quade: „Kern-Technik", Folge 13: Linux-Magazin 8/04, S. 92

[4] Quellcode der LibAIO: [http://rpmseek.com/rpm-pl/libaio.html?hl=de&cs=libaio:PN:0:0:0:0]

[5] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2005/02/Kern-Technik/]

[6] Kernel Asynchronous I/O (AIO) Support: [http://lse.sourceforge.net/io/aio.html]

[7] Suparna Batthacharya: Patches für Async-I/O: [http://www.kernel.org/pub/linux/kernel/people/suparna/aio/]

[8] Stefan Klett, „Virtual Memory. So verwaltet der Linux-Kernel den Speicher": Linux-Magazin 09/03, S. 91

[9] Jonathan Corbet, „ Driver porting: Zero-copy user-space access": [http://lwn.net/Articles/28548]

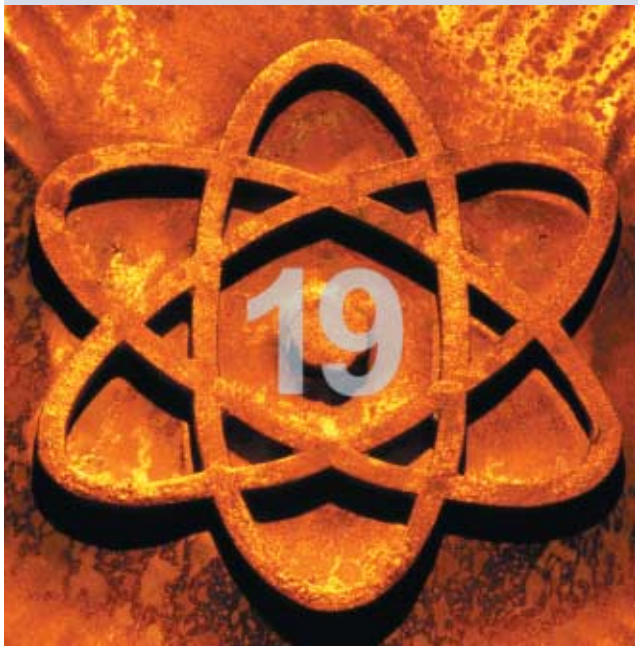
Listing 2: »asynctest.c«

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <asm/uaccess.h>
04 #include <linux/mm.h>
05
06 static wait_queue_head_t read_wq;
07 static struct timer_list ptimer;
08 static void *copy_to;
09 static unsigned long max_copy;
10
11 void timer_function( unsigned long p )
12 {
13    struct kiocb *iocb = (struct kiocb *)p;
14
15    if( !kiocbIsCancelled( iocb ) ) {
16       kick_iocb(iocb);
17    } else {
18       printk("iocb is cancelled ...\n");
19    }
20    return;
21 }
22
23 static int driver_cancel( struct kiocb *iocb,
   struct io_event *event )
24 {
25    printk("driver_cancel ...\n");
26    del_timer( &ptimer );
27    kiocbSetCancelled( iocb );
28    aio_complete( iocb, -1, 0 );
29    return 0;
30 }
31
32 static long driver_retry( struct kiocb *iocb )
33 {
34    printk("driver_retry ...\n");
35    if( copy_to )
36          copy_to_user(copy_to, "Hello
   World",(max_copy>12)?12:max_copy);
37    aio_put_req(iocb);
38    return (max_copy>12)?12:max_copy;
39 }
40
41 static ssize_t driver_aioread(struct kiocb
   *iocb, char __user *buf,
42        size_t count, loff_t offset)
43 {
44    printk( "driver_aioread(%p,%p,%d)\n", iocb,
   buf, count );
45    copy_to  = (void *)buf;
46    max_copy = count;
47    iocb->ki_cancel = driver_cancel;
48    iocb->ki_retry = driver_retry;
49    ptimer.function = timer_function;
50    ptimer.data = (unsigned long)iocb;
51    ptimer.expires = jiffies + (2*HZ);
52    add_timer( &ptimer );
53    return -EIOCBQUEUED;
54 }
55
56 static struct file_operations fops = {
57    .owner=THIS_MODULE,
58    .aio_read=driver_aioread,
59 };
60
61 static int __init async_init(void)
62 {
63    if(register_chrdev(240, "async_test",
   &fops) == 0) {
64       init_waitqueue_head( &read_wq );
65       init_timer( &ptimer );
66       return 0;
67    };
68    return -EIO;
69 }
70
71 static void __exit async_exit(void)
72 {
73    del_timer_sync( &ptimer );
74    unregister_chrdev(240,"async_test");
75 }
76
77 module_init( async_init );
78 module_exit( async_exit );
79 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 19

Kern-Technik

Bei Linux 2.6.10 kann der Anwender über die Auswahl von IO-Schedulern bestimmen, wie der Kernel auf Festplatten zugreift. Wer weiß, wie sie intern funktionieren, passt das System optimal an die eigene Anwendung an. Zur Laufzeit lassen sich die IO-Scheduler über das SysFS austauschen und einstellen. Eva-Katharina Kunst, Jürgen Quade



Das Blockgeräte-Subsystem – im Kernel 2.6 grunderneuert – ist für den effizienten Transport von Daten zwischen Applikationen und Festplatten verantwortlich [1]. Damit hat es wesentlichen Anteil an der Performance und an der so genannten Interaktivität des Systems. Eine ganze Reihe von Techniken gestaltet den Zugriff möglichst optimal (siehe Abbildung 1).

Getrennt lesen und schreiben

So trennt der VFS (Virtual Filesystem Switch) die Aufträge für zeichenorientierte Geräte von denen für blockorientierte. Außerdem löst er Dateinamen in Sektornummern auf. Die zu einer Sektornummer gehörigen Daten sucht der Kernel im Page-Cache, der die Applikation von dem eigentlichen Gerät entkoppelt. Für diese Aufgabe hält der Page-Cache jeweils eine Kopie genau jenes Blocks (Sektors) vorrätig, auf den die Applikation zugreifen möchte. Der Page-Cache legt die Kopie dabei asynchron zum Ablauf der Applikation an.

Die im Page-Cache generierten Aufträge (Requests) werden dem so genannten Elevator übergeben, der das Zusammenfassen und Sortieren von Aufträgen im IO-Scheduler veranlasst. Darüber hinaus sorgt der Elevator (Fahrstuhl) dafür, dass sich der Gerätetreiber zu gegebener Zeit seine Aufträge abholt. Allerdings tut er dies erst dann, wenn eine ausreichende Anzahl von Aufträgen vorhanden ist – sonst gibt es schließlich nichts zu sortieren (siehe Kasten „IO-Scheduler im Blockgeräte-Subsystem“).

Verstopfen und füllen

Bildlich gesprochen verschließt (Plugging) der Elevator den Pool (Queue), in dem sich die Aufträge befinden und zum Sortieren ansammeln. Dann entfernt der Scheduler den Stopfen (Unplugging) und reicht die Aufträge an den Treiber weiter. Natürlich darf der Pool nicht zu lange zugepfropft bleiben. Deshalb startet man beim ersten Auftrag einen Timer mit einem Wert von beispielsweise drei Millisekunden.

Das Sortieren und Zusammenfassen der Aufträge beschleunigt den Festplattenzugriff insgesamt, denn diese Strategie minimiert Bewegungen des Schreib-Lese-Kopfes. Das deutet bereits der Name Elevator an: Die Aufträge werden nach Möglichkeit so an die Platte weitergereicht, dass der Schreib-Lese-Kopf der Platte keine Zickzack-Bewegungen ausführen muss, sondern sich fahrstuhlähnlich von den niedrigen Sektornummern zu den hohen bewegt und genauso wieder wieder zurück.

So einfach das Prinzip klingt, so kompliziert ist es zu implementieren. Ein einfaches Einsortieren neuer Aufträge führt nämlich unter ungünstigen Umständen zum Verhungern (Starvation) einzelner Requests. Erteilt eine Applikation beispielsweise den Auftrag, die Sektoren mit den Nummern 20 und 30000 zu lesen, wird der Block 30000 eventuell erst zu einem viel späteren Zeitpunkt gelesen. Das wäre der Fall, wenn andere Applikationen nur Sektoren kleiner 30000 anfordern.

Das Einsortieren der Aufträge steigert zwar die IO-Performance, verschlechtert aber die Interaktivität. Hier gilt es also, einen möglichst guten Kompromiss zu finden. Dass dieser möglichst wenig Rechenleistung verbrauchen soll, versteht sich von selbst. Da es keinen für jede Anwendung optimalen IO-Scheduler gibt, stellt Kernel 2.6 vier verschiedene Algorithmen zur Wahl: No-Op-, Deadline-, Anticipatory- und CFQ-IO-Scheduler (siehe Abbildung 2).

Der No-Op-Scheduler ist in jedem Fall im Kernel einkompiliert, schon um beim Booten auf das Root-Filesystem zugreifen zu können. Der User darf beim Kernel-Compile auch mehrere IO-Scheduler konfigurieren und beim Booten einen davon auswählen. Noch besser ist allerdings: Ab Kernel 2.6.10 lässt sich der IO-Scheduler für jedes Blockgerät jederzeit separat auswählen. Entsprechende Ein-

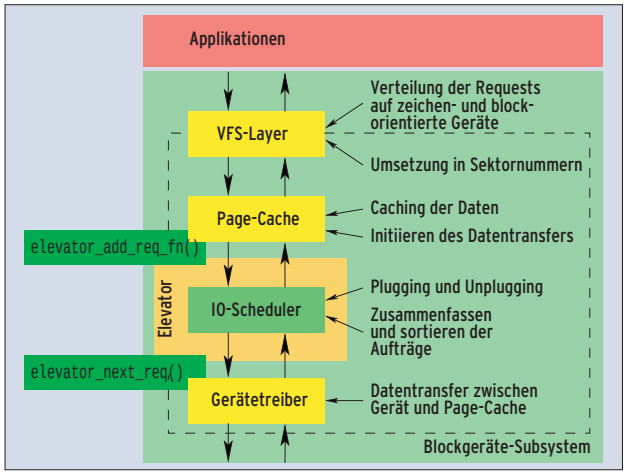


Abbildung 1: Verschiedene Mechanismen beschleunigen den Zugriff auf langsame Festplatten. Die Funktionen »elv_add_request_fn()« und »elevator_next_req()« sind die Hauptschnittstellen zum IO-Scheduler.

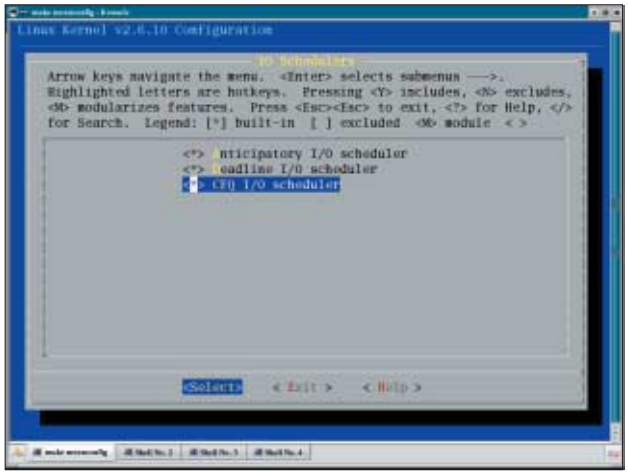


Abbildung 2: Auswahl der IO-Scheduler bei der Kernelkonfiguration. Sind alternative IO-Scheduler Teil des Kernels, kann der Benutzer sie zur Laufzeit über das Sys-Filesystem austauschen.

träge im Sys-Filesystem passen den Algorithmus optimal an die eigene Anwendung an (siehe Kasten: „IO-Scheduler im Kernel 2.6.10“).

Ein- und Ausgabe in Zahlen

Dass der IO-Scheduler großen Einfluss auf die Interaktivität des Systems besitzt, hat Andrew Morton mit einem einfachen Benchmark demonstriert [3]. Um IO-Last zu erzeugen, startet der User im Hintergrund einen großen Datentransfer. Er wartet so lange, bis der Page-Cache mit den Blöcken gefüllt ist, und misst dann die Zeit, bis das Auslesen mehrerer Dateien abgeschlossen ist. Listing 1 implementiert diesen Ablauf. Wie die meisten anderen Benchmarks ist auch dieser nicht völlig zuverlässig: Die 15-Sekunden-Pause entfernt nicht immer alle eingelesenen Dateien aus dem Page-Cache. Das führt bei darauf folgenden Messungen zu – jedoch nur vermeintlich – besseren Werten.

Im Übrigen setzt Listing 1 voraus, dass das Verzeichnis »/tmp/« und die Kernelquellen auf der ersten IDE-Platte »hda« liegen. Allerdings dürfen sich die Kernelquellen und das Tmp-Verzeichnis durchaus auf unterschiedlichen Partitionen dieser Platte befinden. Außerdem muss im Tmp-Verzeichnis etwa 7 GByte freier Platz zur Verfügung stehen.

Abbildung 4 zeigt das Ergebnis: Mit Abstand am schlechtesten schneidet der No-Op-IO-Scheduler ab. Er benötigt zum Auslesen der Dateien ungefähr zehnmal so viel Zeit wie die Konkurrenten. Am besten schneiden der Anticipatory- und der CFQ-IO-Scheduler ab. Die nahe liegende Schlussfolgerung, der Anticipatory-Scheduler sei in jedem Fall der beste, ist allerdings nicht gerechtfertigt. Denn sowohl der Durchsatz als auch die Interaktivität hängen von der Charakteristik der laufenden Applikationen und von der Hardware ab. ▶

IO-Scheduler im Blockgeräte-Subsystem

Aus Sicht des Kernels ist ein IO-Scheduler ein Objekt vom Typ »struct elevator_type«, das mindestens vier Methoden zur Verfügung stellt. Die Funktion »elv_register()« registriert den IO-Scheduler beim Kernel.

Der IO-Scheduler arbeitet dabei wie ein Pool, in dem der Page-Cache seine Aufträge ablegt und von dem der Gerätetreiber Requests zur Bearbeitung entnimmt. Welcher Pool (IO-Scheduler) verwendet wird, steht in der Request-Queue, dem Objekt, das der Treiber bei seiner Initialisierung dem Blockgeräte-Subsystem übergibt (über die Funktion »blk_init_queue()«, siehe [5]).

Der Aufruf der IO-Scheduler-Methoden vom Page-Cache aus ist in der Funktion »__make_request()« in »drivers/block/blk_ll_rw.c« kodiert (siehe Abbildung 2). Zuerst versucht der Page-Cache einen Auftrag mit anderen zu vereinen (Aufruf der Methode »elevator_merge_fn()«). Falls dies nicht gelingt, fügt er den Auftrag über »elevator_add_req_fn()« dem Pool hinzu. Die Funktion »__generic_unplug_device()« übernimmt das Unplugging und damit die Entnahme von Aufträgen durch den Gerätetreiber. Diese Funktion ruft der Kernel auf, wenn der Timer abläuft.

Falls sich im Pool des IO-Schedulers Aufträge befinden, ruft »__generic_unplug_device()« die Request-Funktion des Gerätetreibers auf. Diese Funktion wiederum entnimmt per »elevator_next_ req()« die Aufträge und bearbeitet sie anschließend.

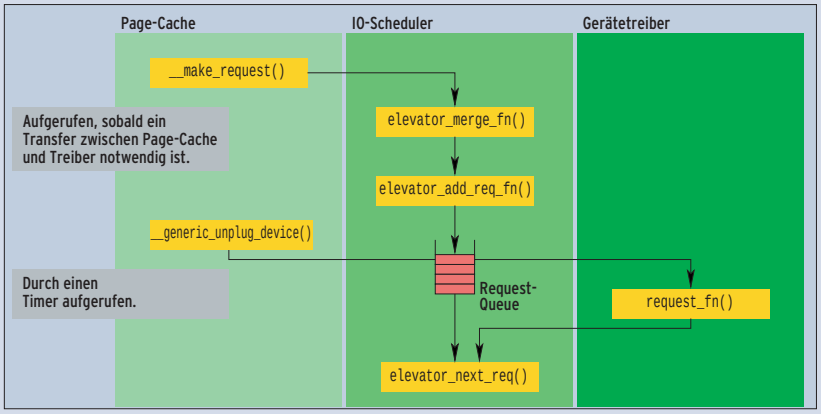


Abbildung 3: Einbettung des IO-Schedulers in das Blockgeräte-Subsystem.

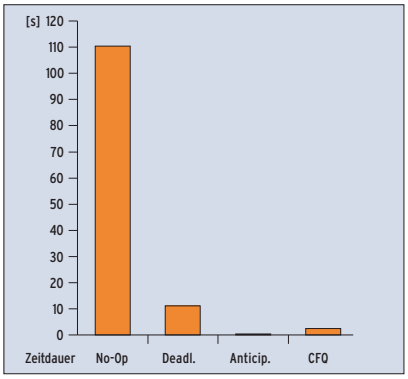


Abbildung 4: Das Diagramm zeigt die Dauer für die Ausgabe einiger C-Dateien bei unterschiedlichen IO-Schedulern an, wenn das System unter IO-Last steht (Pentium M, 1,4 GHz, 512 MByte Hauptspeicher).

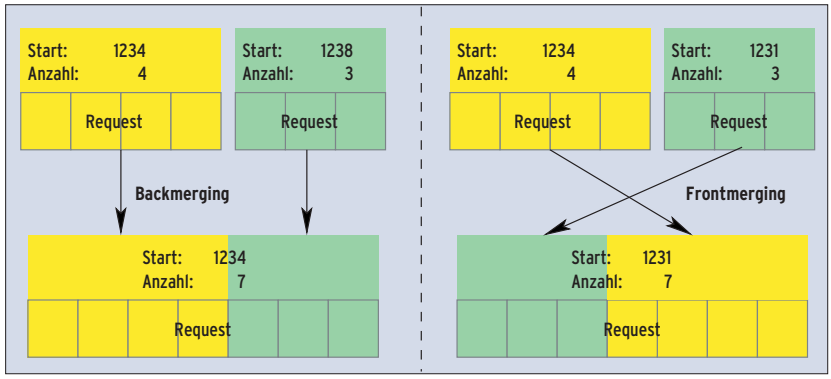


Abbildung 5: Das Zusammenfügen zweier Requests zu einem heißt Merging. Backmerging hängt den zweiten Auftrag an den ersten an, Frontmerging setzt den zweiten vor den ersten.

Moderne SCSI-Festplatten bringen ein so genanntes TCQ (Tagged Command Queuing) mit, womit sie Aufträge selber noch umsortieren. Das wiederum hebelt aber das Umsortieren durch die Kernelalgorithmen aus. Ähnliches gilt für Raid-Systeme, denn bei diesen verteilt sich eine Datei auf mehrere Festplatten. Dass hier mehrere Schreib-Lese-Köpfe bewegt werden, berücksichtigen die Algorithmen aber nicht.

Der einfachste der vier Algorithmen ist der No-Op-IO-Scheduler. No-Op steht für No-Operation, was bedeutet, dass die Requests nicht sortiert werden. Immerhin versucht der Scheduler aber Aufträge zusammenzufassen. Wenn sich ein Auftrag der Art „Lese ab Sektor 1234 vier Sektoren“ in der Request-Queue befindet und danach ein Auftrag „Lese ab Sektor 1238 drei Sektoren“ folgt, fasst der No-Op-Scheduler beide zum Auftrag „Lese ab Sektor 1234 sieben Sektoren“ zusammen (siehe Abbildung 5).

Listing 1: Messung der Interaktivität

```
01 #!/bin/bash
02 # Simples Skript zur Messung der Interaktivität der
   IO-Scheduler.
03 # GPL, (c) 2004, Juergen Quade.
04
05 SCHEDULERFILE=/sys/block/hda/queue/scheduler
06
07 function do_benchmark()
08 {
09    dd if=/dev/zero of=/tmp/foo bs=1M count=7000 &
10    echo $1 >$SCHEDULERFILE
11    sleep 15
12    echo -n "---> $1: "
13    time cat /usr/src/linux/kernel/*.c > /dev/null
14    # clean up
15    echo anticipatory >$SCHEDULERFILE
16    kill -9 $! # kill last background process (dd)
17    rm /tmp/foo
18 }
19
20 do_benchmark noop
21 do_benchmark anticipatory
22 do_benchmark deadline
23 do_benchmark cfq
```

Aufträge zusammenfassen

Hängt der Scheduler einen Auftrag an einen anderen an, so heißt das Backmerging. Entsprechend spricht man von Frontmerging, wenn der Scheduler einen zweiten Auftrag vor einen bereits existierenden stellt. Wenn also nach „Lese ab Sektor 1234 vier Sektoren“ der Auftrag „Lese ab Sektor 1231 drei Sektoren“ folgt, erzeugt der IO-Scheduler durch Frontmerging den Auftrag „Lese ab Sektor 1231 sieben Sektoren“.

Der No-Op-Scheduler hängt einen neuen Auftrag – wenn er ihn nicht mit einem bestehenden zusammenfügen kann – ans Ende einer Liste an. Der Treiber nimmt sich immer das erste Element aus dieser Liste, die damit ein Fifo implementiert, Verhungern der Aufträge kommt nicht vor. Dieser IO-Scheduler ist wirklich simpel, er verbraucht kaum Rechenzeit und besitzt keine für den Anwender einstellbaren Parameter.

Wie das Beispiel in Abbildung 6 zeigt, arbeitet er nicht unbedingt optimal: Drei Rechenprozesse benötigen Daten aus den Festplattensektoren 100, 50 und 70. Wenn Rechenprozess B aus Sektor 50 gelesen hat, braucht er zusätzlich noch Daten aus Sektor 51. Kommen die Aufträge in dieser Reihenfolge beim No-Op-Scheduler an, übergibt er sie beim Unplugging in derselben Reihenfolge dem Gerätetreiber. Der Schreib-Lese-Kopf der Festplatte hat damit aber einen vergleichsweise weiten Weg (Seek-Distanz) zurückzulegen: Erst 100 Sektoren vorwärts, dann 50 Sektoren zurück (was zeitlich betrachtet besonders teuer ist), dann wieder 20 Sektoren nach vorne und schließlich noch einmal 19 Sektoren rückwärts.

Der zweite IO-Algorithmus, der zur Auswahl steht, ist der Deadline-Scheduler. Bei ihm handelt es sich um eine Weiterentwicklung des Linus-Elevators, des Standard-IO-Scheduler in Kernel 2.4.

Im Zeitplan mit dem Deadline-Scheduler

Als Erweiterung zum No-Op-Scheduler gibt der Deadline-Scheduler die einzelnen Lese- und Schreibaufträge in aufsteigender Reihenfolge der Sektornummern an die Festplatte weiter. Weil es ungünstig ist, Schreib- und Leseaufträgen zu mischen, verwaltet der Scheduler die beiden Sorten getrennt.

Seinen Namen erhielt dieser IO-Scheduler, weil er die Aufträge (um das Verhungern zu verhindern) mit einer oberen Zeitschranke (der Deadline) versieht. Damit kann der Deadline-IO-Scheduler die Aufträge nicht nur gemäß der jeweiligen Sektornummer (in der Datenstruktur »sort_list«), sondern zugleich (in einer eigenen Datenstruktur »fifo_list«) auch entsprechend ihres zeitlichen Auftretens sortieren.

Auch die zeitliche Sortierung erfolgt für Lese- und für Schreibaufträge getrennt (Abbildung 7). Der Gerätetreiber ent-

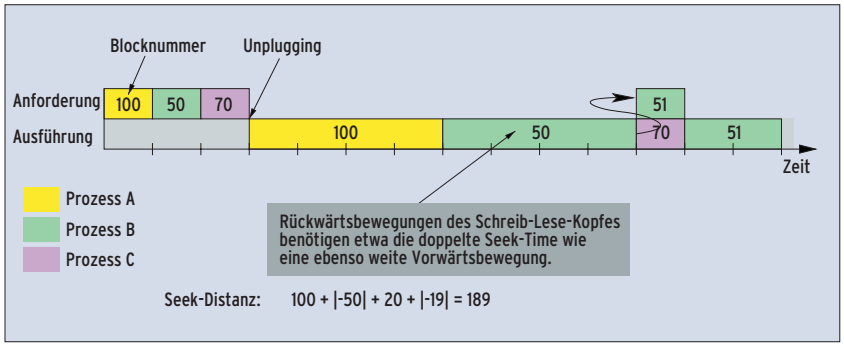


Abbildung 6: Der No-Op-IO-Scheduler reicht die Aufträge in der Reihenfolge ihres Eintreffens an den Gerätetreiber weiter. Das führt zu aufwändigen Rückwärtsbewegungen des Schreib-Lese-Kopfes der Festplatte, die Seek-Distanz ist also relativ groß (Darstellung qualitativ).

nimmt den nächsten Auftrag der Dispatch-Queue. Ist sie leer, prüft er zunächst, ob in der »fifo_list« ein Auftrag steht, dessen Deadline abgelaufen ist. Wenn ja, entnimmt er diesen Auftrag zusammen mit einer Anzahl benachbarter Requests aus der »sort_list« und der »fifo_list« und verschiebt ihn in die Dispatch-Queue (in Abbildung 6 sind dies drei zusätzliche Requests).

Kernelhacker nennen eine solche Gruppe von IO-Aufträgen Batch. Zu ihr gehören auch Aufträge, deren Deadline nicht abgelaufen ist. Falls noch gar keine Deadline überschritten ist, entnimmt der Scheduler seiner »sort_list« den zum vorherigen Request benachbarten Auftrag (in Richtung aufsteigender Sektornummern) und verschiebt ihn in die Dispatch-Queue. Sie führt die bisher getrennt behandelten Lese- und Schreibaufträge wieder zusammen. Dabei bevorzugt sie allerdings normalerweise die Leseaufträge, da diese für die Interaktivität des Systems von größerer Bedeutung sind. Standardmäßig kommt auf zwei Gruppen Leseaufträge eine Gruppe Schreibaufträge (siehe »deadline_dispatch_requests()« in »drivers/block/deadline_iosched.c«).

Abbildung 8 überträgt das Beispiel aus Abbildung 5 auf den Deadline-Scheduler. Bei dieser Variante ist die Seek-Distanz kürzer. Außerdem vermeidet der Algorithmus Rückwärtsbewegungen des Schreib-Lese-Kopfes, was sich positiv auf die Seek-Time auswirkt. Die entsprechende Theorie geht davon aus, dass

IO-Scheduler in Kernel 2.6.10

Zu den Features von Kernel 2.6.10 gehört, dass der IO-Scheduler mit Hilfe des Sys-Filesystems für jedes Blockgerät gesondert ausgewählt werden kann. Wer beispielsweise erfahren möchte, welche IO-Scheduler der Kernel für die erste IDE-Festplatte anbietet, sieht in der Datei »/sys/block/hda/queue/scheduler« nach:

```
# cat /sys/block/hda/queue/scheduler
[noop] anticipatory deadline cfq
```

Im Beispiel stehen der »noop«-, »anticipatory«-, »deadline«- und der »cfq«-IO-Scheduler zur Auswahl, der gerade aktive Scheduler ist in Klammern aufgeführt. Um ihn zu wechseln, genügt es bereits, seinen Namen in die Datei zu schreiben:

```
# echo deadline > /sys/block/hda/queue/➥
  scheduler
# cat /sys/block/hda/queue/scheduler
noop anticipatory [deadline] cfq
```

Im Verzeichnis »/sys/block/hda/queue/« befindet sich das Unterverzeichnis »iosched«. Es exportiert die zum ausgewählten IO-Scheduler gehörigen Attribute jeweils als eine Datei. Der hier ausgewählte Deadline-Scheduler stellt in Kernel 2.6.10 insgesamt fünf Attribute zur Auswahl.

Diese Sequenz liest das Attribut »fifo_batch« aus und ändert es danach auf den Wert »8«:

```
# cd /sys/block/hda/queue/iosched/
# ls
fifo_batch  front_merges  read_expire➥
  write_expire  writes_starved
# cat fifo_batch
16
# echo 8 >fifo_batch
# cat fifo_batch
8
```

Das Attribut »writes_starved« lässt sich in Kernel 2.6.10 aufgrund eines Bugs allerdings nicht verändern.

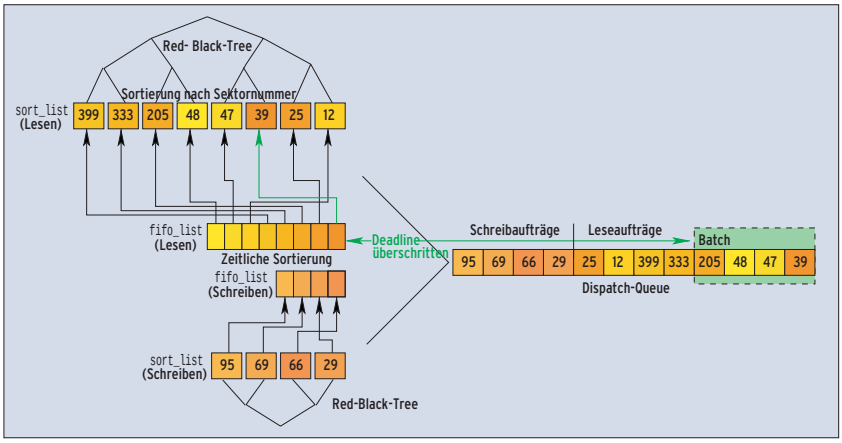


Abbildung 7: Neue Aufträge verwaltet der Scheduler in der »sort_list« nach ihrer Sektornummer, in »fifo_list« nach der Reihenfolge. Beim Unplugging verschiebt er die Aufträge in die Dispatch-Queue und übergibt sie so dem Gerätetreiber. In der Queue sind zuvor getrennte Lese- und Schreibaufträge wieder vereint.

Rückwärtsbewegungen doppelt so aufwändig sind wie gleich weite Vorwärtsbewegungen des Kopfes.

Nicht nur der Algorithmus als solcher, auch die Implementierung ist für Programmierer interessant: Neben doppelt verketteten Listen kommen Red-Black-Trees [4] und Hashfunktionen zum Einsatz. Die doppelt verketteten Listen (»fifo_list«) dienen der Einsortierung nach der Auftragszeit – jeweils eine Liste für Leseaufträge und eine für Schreibaufträge. Red-Black-Trees (»sort_list«) sind optimal für die Sortierung nach der Sektornummer. Dabei verwaltet ebenfalls jeweils ein Tree die Leseaufträge, ein anderer die Schreibaufträge. Die Hashfunktion fasst schließlich die Aufträge zusammen.

Feineinstellung über SysFS

Die Wahl der Algorithmen sorgt für eine weitgehend konstante Laufzeit des Algorithmus – abgesehen vom Einsortieren der Aufträge. Die Red-Black-Trees garantieren allerdings auch hier einen überschaubaren Aufwand mit der Komplexität O(Log N).

Der Kasten „IO-Scheduler in Kernel 2.6.10“ beschreibt, wie man den Deadline-IO-Scheduler einstellt (siehe auch [2]). Er besitzt folgende Parameter:

- »fifo_batch«: Dieser Parameter legt fest, wie viele benachbarte Aufträge der Scheduler in die Dispatch-Queue zur Übergabe an den Gerätetreiber übernimmt, wenn die Deadline eines Auftrags abgelaufen ist. Standardmäßig bilden 16 Aufträge eine Auftragsgruppe (Batch).
- »read_expire«: Der Wert bestimmt die Länge der Deadline für Leseaufträge in Millisekunden. Standardmäßig beträgt sie 500 Millisekunden.
- »write_expire«: Das Attribut legt die Länge der Deadline für Schreibaufträge in Millisekunden fest. Standardmäßig sind es fünf Sekunden.
- »writes_starved«: Das Attribut legt fest, wie viele Gruppen der Scheduler für Leseaufträge bearbeitet, bevor er eine Gruppe Schreibaufträge angeht. Gibt es keine Lese-, sondern nur Schreibaufträge, übernimmt er diese nach Ablauf der Deadline direkt. Normalerweise verarbeitet er vor einem Schreib-Batch zwei Lese-Batches.
- »front_merges«: Dieser Parameter legt fest, ob der Deadline-IO-Scheduler überhaupt Frontmerges durchführt. Üblicherweise erlaubt man Frontmerges, sodass der Parameter von Haus aus den Wert »1« besitzt.

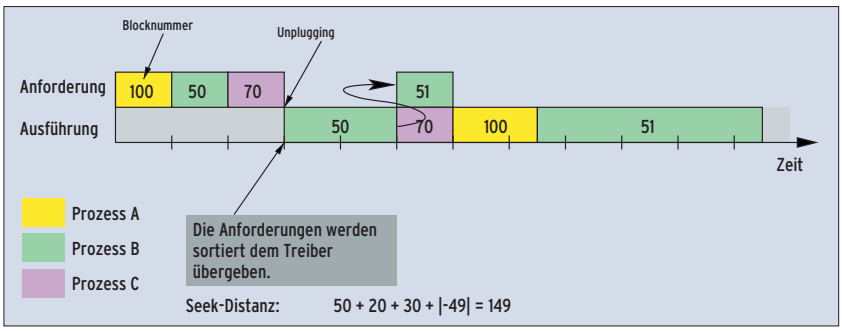


Abbildung 8: Der Deadline-IO-Scheduler sortiert die Anfragen der Rechenprozesse. Später eingehende Aufträge werden nicht mehr einsortiert (Darstellung qualitativ).

Vorschau

Für jede IO-Scheduling-Strategie wäre es am besten, in die Zukunft schauen zu können. Denn wer die folgenden Lese- und Schreibaufträge kennt, kann am besten entscheiden, wie er mit den aktuellen verfährt. Einen Schritt in diese Richtung macht der Anticipatory-Scheduler, der zu erraten versucht, welche Ordnung der Aufträge am günstigsten ist. Wer erfahren möchte, wie der Anticipatory-Scheduler arbeitet und was ihn vom CFQ-Scheduler unterscheidet, findet alle Details in der kommenden Folge der Kern-Technik. *(ofr)* ■

Infos

[1] Jens Axboe, „Linux Block IO – present and future“: Proceedings of the Linux Symposium, Volume One, Ottawa Juli 2004, [http://www.finux.org/Reprints/Reprint-Axboe-OLS2004.pdf]

[2] Jens Axboe, „Deadline IO scheduler tunables“: [http://lxr.linux.no/source/Documentation/block/deadline-iosched.txt?v=2.6.8.1]

[3] IO-Scheduler-Diskussion auf der Kernel-Mailingliste: [http://kerneltrap.org/node/431]

[4] Wikipedia, Red-Black-Tree: [http://en.wikipedia.org/wiki/Red-black_tree]

[5] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 8: Linux-Magazin 3/04, S. 87

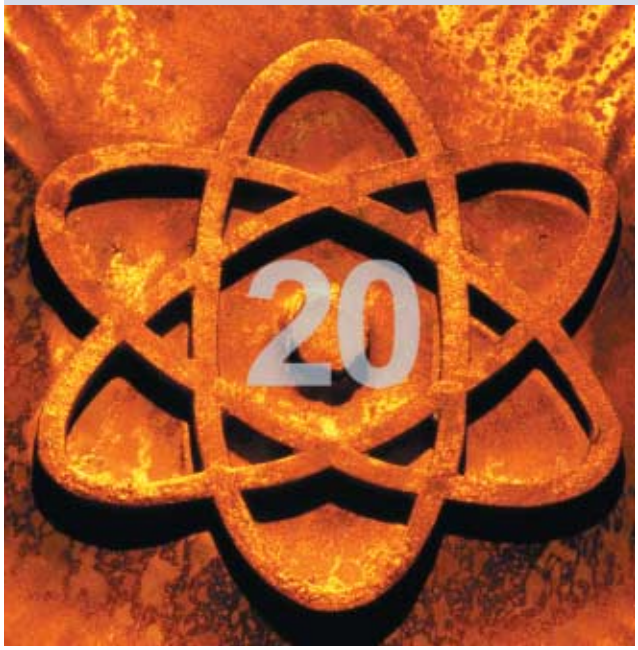
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie gemeinsam ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 20

Kern-Technik

Mit dem Anticipatory-IO-Scheduler greift der Linux-Kernel vorausschauend und damit recht effektiv auf Festplatten zu. Der brandneue CFQ-IO-Scheduler tritt als ambitionierter Konkurrent auf. Der Artikel erklärt die Arbeitsweise beider Zugriffsstrategien und zeigt, worin sie sich unterscheiden. Eva-Katharina Kunst, Jürgen Quade



Die vorige Folge der Kern-Technik-Reihe stellte die Arbeitsweise der No-Op- und Deadline-IO-Scheduler vor [1]. Beide sind Beispiele für die neue IO-Architektur in Kernel 2.6, die es erlaubt, IO-Scheduler zur Laufzeit auszutauschen. Dabei demonstriert der untätige No-Op-Scheduler auf einfache Weise, wie sich eine solche Komponente ins Gesamtsystem integriert. In echten Arbeitsumgebungen liefern der Deadline- und die im Folgenden beschriebenen Anticipatory- und CFQ-IO-Scheduler weitaus bessere Ergebnisse, denn sie arbeiten mit ausgefeilten Algorithmen.

In die Zukunft sehen

Der Anticipatory-IO-Scheduler (vorausschauend, anticipatory) kann seine Verwandtschaft zum Deadline-IO-Scheduler nicht leugnen [2]. Wie dieser sortiert er die Aufträge nach aufsteigenden Sektornummern in einem Red-Black-Tree und verhindert über eine mit Fifos implementierte Zeitüberwachung das Verhungern einzelner Aufträge. Auch er kennt die Dispatch-Queue, in die er – zum Beispiel nach dem Unplugging – Aufträge verschiebt und aus der der Plattentreiber sich die Aufträge nach dem Fifo-Prinzip herausholt.

Zusätzlich berücksichtigt der Anticipatory-IO-Scheduler jedoch auch den Rechenprozess, in dessen Auftrag er lesen und schreiben soll. Das ist durchaus sinnvoll, denn üblicherweise verteilen sich die Daten einer Applikation nicht kreuz und quer über die gesamte Festplatte. Vielmehr legen moderne Filesysteme die Daten in möglichst sequenziell hintereinander liegenden Sektoren auf der Festplatte ab, man spricht von einer Lokalität der Daten.

Die Sache hat nur einen Haken: In Unix-Systemen greifen Applikationen synchron zu. Sie starten einen neuen Leseauftrag erst, wenn der letzte Auftrag abgeschlossen ist (siehe Abbildung 1). Bei Verwendung des Deadline-Schedulers [1] bedeutet dies, dass der Schreib-Lese-Kopf der Festplatte sich bereits von der gewünschten Plattenposition entfernt hat. Der nachfolgende Auftrag müsste also wohl knapp 500 Millisekunden warten – das ist viel zu lange für ein interaktives System.

Anders als der Deadline- verschiebt der Anticipatory-Scheduler normalerweise nicht gleich einen ganzen Batch in die Dispatch-Queue, sondern nur einen einzelnen Auftrag. Holt der Gerätetreiber einen Auftrag beim IO-Scheduler ab, überprüft dieser den nächsten Kandidaten. Stammt der Auftrag vom gleichen Rechenprozess, reicht der Scheduler ihn an den Gerätetreiber weiter. Das tut er auch, wenn es sich um einen benachbarten Auftrag handelt, also einen, dessen Speicherort auf der Festplatte sich in der Nähe befindet.

Aufträge bearbeiten

In allen anderen Fällen zieht der IO-Scheduler eine Zugriffsstatistik zu Rate, die er für jeden Prozess getrennt führt. Ist zu erwarten, dass ein Prozess bald einen neuen Auftrag erteilen wird, bleibt seine Dispatch-Queue zunächst leer – der Treiber und damit das Gerät bekommen keinen neuen Auftrag und warten. Trifft ein neuer Auftrag ein, reicht der Scheduler ihn an den Gerätetreiber durch. Bleibt er aus, startet der IO-Scheduler einen Timer. Er holt sich entweder aus der Fifo-Liste (abgelaufene Deadline) oder aus dem Red-Black-Tree den nächsten Auftrag.

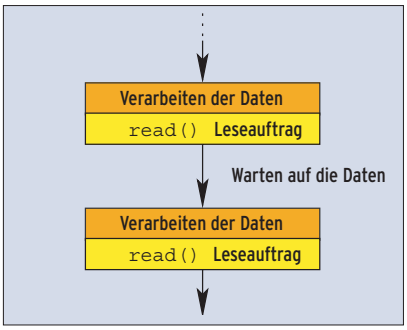


Abbildung 1: Synchrones Lesen: Typischerweise erteilen Applikationen erst dann den nächsten IO-Auftrag, wenn der vorhergehende abgeschlossen ist.

Abbildung 2 zeigt, wie der Anticipatory-Scheduler auf Requests wartet. Trifft während der Wartezeit kein neuer Auftrag vom selben Prozess (in der Abbildung nach Sektor 70) ein, übergibt der Scheduler den nächsten, bereits vorhandenen Auftrag (hier Sektor 100) an den Gerätetreiber.

Dass sich das Warten lohnt, zeigt ein Blick auf die für das Beispiel optimale Seek-Distanz. Für den Deadline-IO-Scheduler betrug sie im analogen Fall 189 (siehe [1]). Auch die Seek-Time, also die Zugriffszeit als solche, verkürzt sich trotz Wartens erheblich.

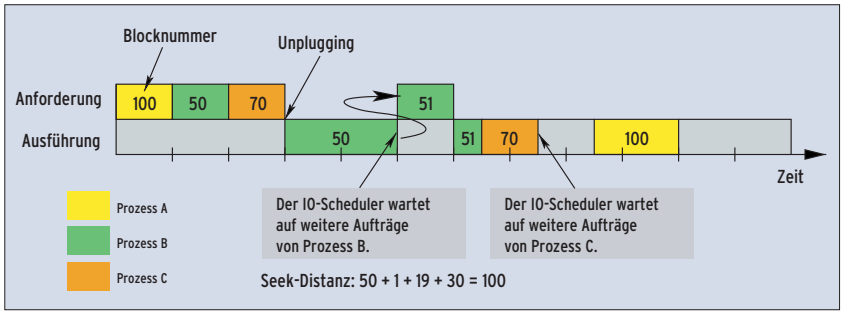


Abbildung 2: Der Anticipatory-IO-Scheduler sortiert die Anfragen und wartet zusätzlich auf später ankommende Aufträge. Rückwärtsbewegungen des Schreib-Lese-Kopfes werden damit weitgehend vermieden.

Der Scheduler führt Buch

Woran erkennt der Anticipatory-Scheduler, dass es besser ist, noch zu warten und den Auftrag eines anderen Prozesses nicht sofort dem Gerätetreiber zu übergeben? Dazu berechnet er für jeden Rechenprozess die bereits erwähnte Statistik, bei der er den durchschnittlichen zeitlichen Abstand zwischen zwei Leseaufträgen misst. In der Sprache des Anticipatory-Schedulers ist dies die Denkzeit (Thinktime). Zudem führt er eine Statistik darüber, wie nahe die jeweiligen Aufträge beieinander liegen (Seek-Distanz). Wegen dieses Faktors findet ein Rechenprozess kaum Berücksichtigung, der zwar häufig Requests erteilt, die aber weit auseinander liegen.

Wie seine einfacher gestrickten Kollegen unterscheidet auch der Anticipatory-IO-Scheduler zwischen Lesen und Schreiben. Allerdings richtet er sich nicht nach der Anzahl der übergebenen Batches, sondern beachtet eine einstellbare Zeitdauer. Dieser Wert ist für Lesen und Schreiben getrennt einzustellen (Parameter »read_batch_expire« und »write_batch_expire«). Die Schreibaufträge übergibt man wie beim Deadline-IO-Scheduler als Batch.

Und noch ein Unterschied zu den IO-Schedulern aus der vorigen Folge: Der Anticipatory-Scheduler gibt unter Umständen auch Aufträge mit niedrigeren Sektornummern als der aktuellen an den Gerätetreiber weiter. Er erlaubt also eine Rückwärtsbewegung des Schreib-Lese-Kopfes. Das macht er, wenn der nächsthöhere Sektor mehr als doppelt so weit entfernt ist wie der niedrigere.

Der Anticipatory-IO-Scheduler ist der Standard-IO-Scheduler im aktuellen Linux-Kernel. Die Tabelle „Anticipatory-IO-Scheduler einstellen“ beschreibt die damit verfügbaren Tuning-Parameter im Einzelnen.

Completely Fair Queueing

Einen etwas anderen Ansatz als der Anticipatory-IO-Scheduler verfolgt der CFQ-IO-Scheduler (CFQ, Completely Fair Queueing). Sein Grundprinzip besteht darin, die vorhandene Bandbreite für den Plattenzugriff möglichst gerecht zwischen den einzelnen Rechenprozessen aufzuteilen.

Aufträge legt er prinzipiell nach Threadgruppen (TGID) getrennt ab. Abbildung 3 zeigt, dass dazu für jede Threadgruppe (»struct cfq_queue«) ein eigener Red-Black-Tree (»sort_list«) und eine ei-

Tabelle 1: Anticipatory-IO-Scheduler einstellen

Parameter	Beschreibung
»antic_expire«	Maximale Denkzeit eines Rechenprozesses in Millisekunden. Besitzt »antic_expire« den Wert 0, wartet der Scheduler überhaupt nicht (ähnliches Verhalten wie beim Deadline-IO-Scheduler). Defaultwert: 6
»read_expire«	Deadline eines Read-Requests in Millisekunden. Da nur alle »read_expire« Millisekunden ein Auftrag aus der »fifo_list« berücksichtigt wird, handelt es sich um eine weiche Zeitschranke. Lese- und Schreibaufträge behandelt der IO-Scheduler alternierend, sodass er das Überschreiten einer Read-Request-Deadline nur dann überprüft, wenn er auch Leseaufträge verarbeitet (im Sync-Modus). Defaultwert: 125
»write_expire«	Deadline für einen Write-Request in Millisekunden. Das Überschreiten einer Write-Request-Deadline verarbeitet der Scheduler nur bei Schreibaufträgen (im Async-Modus). Defaultwert: 250
»read_batch_expire«	Innerhalb der Zeit »read_batch_expire« verarbeitet der IO-Scheduler Leseaufträge (Angabe in Millisekunden). Danach schaltet er auf Schreiben um. Defaultwert: 500
»write_batch_expire«	Zeitspanne in Millisekunden, in der Schreibaufträge verarbeitet werden. Nach Ablauf dieser Zeit bekommt der Gerätetreiber Leseaufträge übergeben. Defaultwert: 125
»est_time«	Bei »est_time« handelt es sich nur um eine Statusinformation des Schedulers. Insgesamt lassen sich im Kernel 2.6.10 dieser Datei die im Folgenden aufgeführten drei Informationen entnehmen: »exit probability«: Die Wahrscheinlichkeit, dass ein Prozess endet oder in absehbarer Zeit keinen neuen Auftrag generiert. Ist sie größer als 50 Prozent, bricht der Scheduler das Warten zugunsten eines anderen Auftrags ab. »new thinktime«: Die Wartezeit, die der Scheduler einem Rechenprozess zugesteht, der erstmalig einen Request in Auftrag gibt, bevor er andere, bereits schedulte Requests dem Gerätetreiber übergibt. »sectors new seek distance«: Die durchschnittliche Seek-Distanz bei Leseaufträgen. Falls während des Wartens ein neuer (unerwarteter) Auftrag eintrifft und der angeforderte Sektor kleiner als dieser Wert ist, bricht der Scheduler das Warten zugunsten des neuen Auftrags ab.

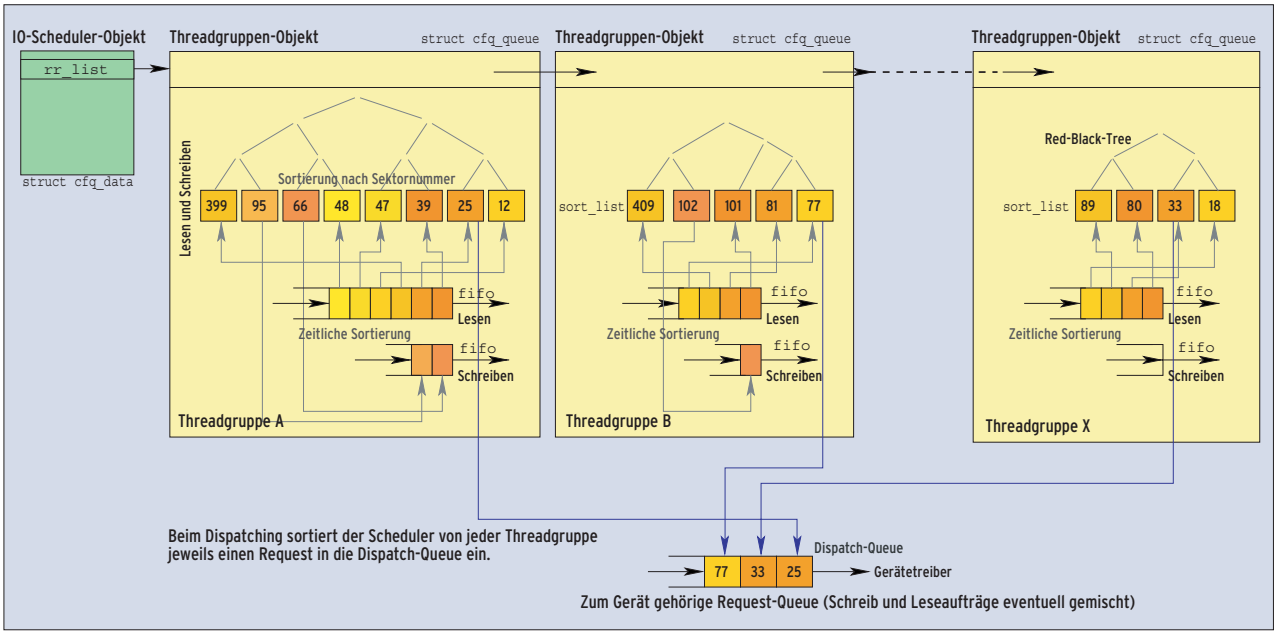


Abbildung 3: Der CFQ-IO-Scheduler besitzt für jede Threadgruppe einen eigenen Satz an Datenstrukturen. Um Fairness zu garantieren, übernimmt er von jeder Threadgruppe jeweils einen Auftrag in die Dispatch-Queue. Die so geordneten Aufträge wandern über eine Warteschlange zum Gerätetreiber.

gene Fifo-Liste (»fifo«) zur Verfügung stehen. Innerhalb der »sort_list« gibt es übrigens keine Unterscheidung zwischen Lese- und Schreibaufträgen, wohl aber in der Fifo-Liste.

Dass der CFQ-Scheduler nicht selbst zwischen Lese- und Schreibaufträgen unterscheidet, spielt keine Rolle, denn Schreibaufträge erfolgen ohnehin selten im Kontext des Prozesses, zu dem die Daten gehören. Vielmehr initiiert der Systemprozess »pdflush« das Sichern der Daten aus dem Pagecache, sodass die Trennung aufgrund der Threadgruppe gleichzeitig auch Schreib- und Leseaufträge unterscheidet.

Das Dispatching, also das Verteilen und Übergeben der Aufträge an den Gerätetreiber, geschieht in einer Funktion namens »cfq_dispatch_requests()«. Für jede Threadgruppe verschiebt der IO-Scheduler jeweils einen Auftrag in die Dispatch-Queue. Aufträge einer Threadgruppe bleiben von ihm unberücksich-

Tabelle 2: CFQ-IO-Scheduler einstellen

Parameter	Beschreibung
»fifo_expire_sync«	Deadline für einen Read-Request in Millisekunden. Da der Scheduler jedoch nur alle »fifo_batch_expire« Millisekunden einen Auftrag aus der Fifo-Liste »fifo« berücksichtigt, handelt es sich um eine weiche Zeitschranke. Defaultwert: 500
»fifo_expire_async«	Deadline für einen Write-Request in Millisekunden. Da der Scheduler jedoch nur alle »fifo_batch_expire« Millisekunden einen Auftrag aus der Fifo-Liste »fifo« überprüft und Leseaufträge zudem bevorzugt behandelt werden, handelt es sich um eine weiche Zeitschranke. Defaultwert: 5000
»fifo_batch_expire«	Dieser Parameter legt die Zeit in Millisekunden fest, ab der der Scheduler das erste Mal wieder die Fifo-Liste »fifo« überprüft, nachdem er ihr einen Auftrag entnommen hat. Defaultwert: 125
»find_best_crq«	Ist »find_best_crq« gesetzt, nimmt der IO-Scheduler den nächstgünstigsten Request (also den dem letzten Auftrag benachbarten). Steht »find_best_crq« auf 0, sucht er den Request mit der kleinsten Sektornummer aus. Defaultwert: 1
»quantum«	Dieser Parameter legt fest, wie viele Requests (aus beliebig vielen Threadgruppen) minimal bei einem Durchlauf in die Dispatch-Queue verschoben werden. Defaultwert: 4
»back_seek_penalty«	Je weniger Bewegungen der Schreib-Lese-Kopf der Festplatte ausführen muss, desto besser ist die Performance. Entsprechend sucht der CFQ-IO-Scheduler den Request, dessen Sektornummer möglichst nahe dem aktuellen ist. Allerdings werden hierbei die Sektornummern, die vor dem aktuellen Sektor liegen (also die kleineren Sektornummern) zusätzlich noch mit der »back_seek_penalty« bewertet (multipliziert). Ist die »back_seek_penalty« beispielsweise 2, bevorzugt der Scheduler einen Request, dessen Sektor maximal doppelt so weit entfernt ist wie der nächste Sektor in Vorwärtsrichtung. Defaultwert: 2
»back_seek_max«	Die maximale Zahl an Sektoren, die sich der Scheduler für einen Auftrag rückwärts bewegt. Liegt der Sektor des Requests weiter als »back_seek_max« vom letzten Request entfernt, bevorzugt er eine Bewegung nach vorne. Defaultwert: 16384
»key_type«	Normalerweise wandern die Requests einer Threadgruppe (»tgid«) in eine Queue. Die Gruppierung kann aber auch nach Prozessgruppen (»pgid«), einzelnen Benutzern (»uid«) oder einzelnen Gruppen (»gid«) geschehen. Defaultwert: »tgid«
»queued«	Anzahl der Requests pro Threadgruppe, die zu einem Zeitpunkt in die Dispatch-Queue eingehen. Kommt in der Threadgruppe ein weiterer Request an, wartet der Scheduler, bis die vorhergehenden verarbeitet sind. Defaultwert: 8
»clear_elapsed«	Dieser (nur schreibbare) Parameter setzt interne Accounting-Variablen zurück. Intern führt das System nämlich Buch darüber, wie lange ein Request bis zu seiner wirklichen Bearbeitung im IO-Scheduler verweilt. Der Parameter löscht die Variablen, die den längsten Verbleib speichern. Allerdings hat das keinerlei Bedeutung für den Anwender, denn weder verarbeitet der Code die Accounting-Variablen intern, noch lassen sie sich in der im Kernel 2.6.10 eingebauten Variante auslesen.

tigt, wenn sich einer ihrer Aufträge in Bearbeitung (»in_flight«) befindet. Der Dispatch-Queue (»queuelist«) entnimmt der Gerätetreiber nach dem Fifo-Prinzip die Aufträge zur Bearbeitung.

Insgesamt versucht der Dispatcher eine Anzahl »quantum«-Requests in die »queuelist« zu übernehmen. Gibt es wenige Threadgruppen mit IO-Aufträgen, verschiebt der IO-Scheduler auch schon mal mehrere Aufträge aus einer einzigen Threadgruppe.

Timer sorgen für Gerechtigkeit

Normalerweise verschiebt der Scheduler den in der »sort_list« folgenden Auftrag einer jeden Threadgruppe – abhängig also vom zuletzt erteilten Auftrag – nach dem Elevator-Prinzip. Um dabei das Verhungern (Starvation) einzelner Aufträge zu verhindern, überwacht auch der CFQ-IO-Scheduler über die Fifo-Liste die abgelaufene Zeit. Die Werte »fifo_sync_expire« (fürs Lesen; die entsprechende interne Variable heißt »cfq_fifo_expire_r«) sowie »fifo_async_expire()« (fürs Schreiben; die interne Variable heißt »cfq_fifo_expire_w«) parametrisieren die zugehörigen Deadlines.

Steht an erster Stelle in der Fifo-Liste ein Auftrag, dessen Deadline abgelaufen ist, bevorzugt der Scheduler ihn beim Verschieben in die Dispatch-Queue. Gibt es einen solchen Auftrag sowohl in der Lese- als auch in der Schreib-Fifo-Liste, besitzt die Lese-Fifo Priorität. Allerdings überwacht der IO-Scheduler die Fifo-Listen nicht ständig. Hat er sich einmal aus der Fifo-Liste bedient, wartet er mindestens die Zeit »fifo_batch_expire« ab, bevor er erneut die Fifo-Listen der entsprechenden Queue überprüft. Die Tabelle „CFQ-IO-Scheduler einstellen“ beschreibt die Parameter und ihre Verwendung im Detail.

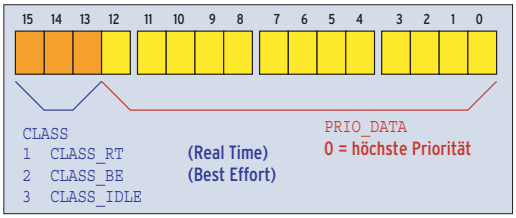


Abbildung 4: Die IO-Priorität wird mit 16 Bit kodiert. Die obersten 3 Bits wählen eine von drei definierten Klassen aus. Die unteren 13 Bits spezifizieren im Fall von Best Effort die Priorität (von 0 bis 7).

Blick in die Glaskugel

Im Vergleich zum Anticipatory-IO-Scheduler kann der CFQ bei Durchsatz und maximaler Latenzzeit nicht überzeugen. Wegen seiner Vorzüge in puncto Interaktivität aktivieren einige bekannte Distributionen CFQ jedoch bereits beim Booten. Entscheidender für die Kernelentwickler ist hingegen das Potenzial, das sie in den verwendeten Grundprinzipien sehen. Anders als der Anticipatory-IO-Scheduler ist der CFQ-IO-Scheduler nämlich noch nicht ausgereizt.

In der Tat arbeitet Jens Axboe, der verantwortlich für das Blockgeräte-Subsystem im Linux-Kernel ist, kräftig am CFQ. Auf der Kernel-Mailingliste kursierten bereits Patches für einen Time-sliced CFQ-IO-Scheduler mit Prioritäten [3]: Statt von jedem Rechenprozess genau einen IO-Auftrag in die Dispatch-Queue zu übernehmen, stellt diese Scheduler-Variante jedem Rechenprozess eine Zeitscheibe (Time Slice) zur Verfügung. Währenddessen übernimmt er Aufträge in die Dispatch-Queue und bearbeitet sie auch gleich. Wie beim Anticipatory-IO-Scheduler eröffnet das die Möglichkeit, eine kurze Zeit (maximal die einstellbare Zeit »slice_idle«) auf weitere Aufträge des gleichen Prozesses zu warten. Diese Aufträge sind wegen des Lokalität-Prinzips mit großer Wahrscheinlichkeit benachbart zum vorherigen.

So gut wie möglich

Damit nicht genug: Der neue Time-sliced CFQ-IO-Scheduler berücksichtigt zusätzlich noch IO-Prioritäten. Jens Axboe definierte zunächst drei Prioritätenklassen: »Idle«, »Best Effort« (BE) und »Realtime«. Die normalerweise verwendete Klasse »Best Effort« bietet insgesamt acht Prioritäten, wobei die mit »0« bezeichnete Priorität die höchste dieser Klasse ist (siehe Abbildung 4). Ohne weiteres Zutun teilt der IO-Scheduler ei-

Tabelle 3: Parameter des Time-sliced CFQ-Schedulers mit IO-Prioritäten

Parameter	Beschreibung
»back_seek_max«, »back_seek_penalty«, »fifo_expire_async«, »fifo_expire_sync«, »fifo_batch_expire«, »quantum«, »queued«	Diese Parameter unterscheiden sich in ihrer Bedeutung nicht von denen des Standard-CFQ-IO-Schedulers (siehe Tabelle „CFQ-IO-Scheduler einstellen“).
»max_depth«	Gibt an, wie viele Aufträge einer Threadgruppe der Scheduler zu einem Zeitpunkt in die Dispatch-Queue übernimmt. Default: 1
»slice_sync«	Länge der Basiszeitscheibe für Best-Effort-Leseaufträge in Millisekunden. Die wirkliche Länge der Zeitscheibe ergibt sich aufgrund der IO-Priorität des zugehörigen Rechenprozesses. Je höher die Priorität, desto länger ist die Zeitscheibe. Leseaufträge von Rechenprozessen der niedrigsten Priorität bekommen genau eine Basiszeitscheibe; bei acht Prioritätsebenen entspricht die tatsächliche Zeitscheibe bei der höchsten IO-Priorität acht Basiszeitscheiben. Defaultwert: 22
»slice_async«	Länge der Basiszeitscheibe in Millisekunden für Schreibaufträge der Prioritätsklasse »Best Effort«. Siehe »slice_sync« für die Herleitung der realen Zeitscheibenlänge aufgrund der Priorität. Defaultwert: 10
»slice_idle«	Zeit in Millisekunden, die der Scheduler während einer Zeitscheibe maximal wartet, bis der zugehörige Thread den nächsten Auftrag anfordert (Abbildung 6). Verstreicht »slice_idle«, bricht er die Zeitscheibe ab und sortiert Aufträge des nächsten Thread in die Dispatch-Queue ein. Defaultwert: 10
»slice_async_rq«	Wert, der indirekt die maximale Anzahl von Schreibaufträgen bestimmt. In die Berechnung dieser Anzahl fließt die IO-Priorität des Thread ein. Je höher die Priorität, desto mehr Schreibaufträge pro Scheduler-Umlauf sind möglich. Defaultwert: 2

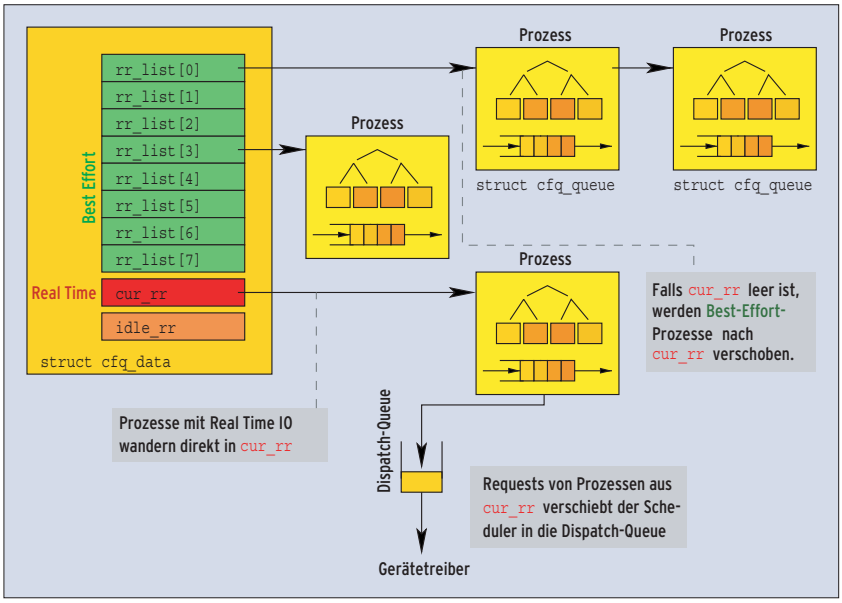


Abbildung 5: Die Requests eines Thread sind nach Prioritäten sortiert. Fordert der Gerätetreiber einen Request an, übergibt der IO-Scheduler einen Auftrag aus der »cfq_queue« an die Real-Time-Queue »cur_rr«. Ist diese leer, fügt er stattdessen einen Best-Effort-Prozess hinzu.

nem Prozess eine Best-Effort-Priorität gemäß seinem Nice-Wert zu. Um diesen Wert zu beeinflussen oder um einem Prozess gar eine Echtzeitpriorität zu geben, führte Axboe die neuen, aber nur vorläufigen Systemcalls »ioprio_set« und »ioprio_get« ein. Sollte der Code Eingang in den offiziellen Kernel finden – so die Warnung – sind Änderungen zu erwarten. Ein Beispiel dafür, wie man diese Systemcalls nutzt, findet sich in [4].

Zeitscheiben

Abbildung 5 zeigt die wesentlichen Datenstrukturen des Time-sliced CFQ-IO-Schedulers mit Prioritäten. Das Objekt »cfq_data« repräsentiert das Gerät respektive den IO-Scheduler. Die Zeitscheibenvariante des CFQ-IO-Schedulers sortiert die Aufträge nicht auf Basis der Threadgruppen (TGID), sondern nach Threads (PID). Jeder Prozess findet eine Entsprechung in einem Objekt des Typs »cfq_queue« – allerdings nicht nur in einer Liste, sondern gleich in mehreren gemäß der IO-Priorität.

Die ersten acht Listen »rr_list« entsprechen den acht Prioritäten für Best-Effort-Aufträge. Aufträge, die über die Dispatch-Queue an den Gerätetreiber gehen, werden allerdings nur den Request-Queues entnommen, die in der Liste »cur_rr« stehen. Immer wenn »cur_rr« leer ist, entnimmt der Scheduler ein »cfq_queue«-Objekt gemäß IO-Priorität den Best-Effort-Listen. Prozessobjekte, deren IO-Aufträge der Real-Time-Klasse angehören, hängt der Scheduler direkt in die Liste »cur_rr« ein und bedient sie somit zuerst.

Die Funktion »cfq_dispatch_requests()« verschiebt nach bekanntem Schema maximal »quantum« Aufträge aus der Request-Queue in die Dispatch-Queue. Falls der zur Request-Queue gehörige Prozess seine IO-Zeitscheibe verbraucht hat, berücksichtigt der Scheduler die Request-Queue des nächsten Prozesses aus der Schlange »cur_rr«. Das »struct cfq_queue«-Objekt verwaltet die eigentlichen IO-Aufträge (»cfq_request«) in Red-Black-Trees und in Fifo-Listen getrennt fürs Lesen und Schreiben.

Tests mit der Zeitscheiben-Variante des CFQ zeigen, dass dieser bezüglich Durchsatz und Latenzzeiten zum Anticipatory-IO-Scheduler aufschließt [5]. Übertrumpft hat er ihn allerdings bislang noch nicht.

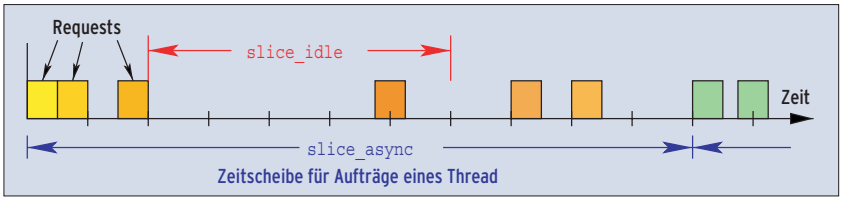


Abbildung 6: Jedem Thread steht eine Zeitscheibe zur Verfügung, während der er exklusiv Aufträge an den Gerätetreiber übergeben kann. Die Länge der Zeitscheibe hängt von der IO-Priorität ab. Wartet er nach einem Auftrag länger als »slice_idle«, wird die Zeitscheibe abgebrochen.

IO-Scheduling-Alternativen

Jens Axboe ist nicht konkurrenzlos, wenn es um die Optimierung der Festplattenzugriffe geht. Philips in Eindhoven beispielsweise entwickelt zusammen mit Werner Almesberger das Active Block IO Scheduling System (ABISS), das den Applikationen garantierte Lese- und Schreibraten bereitstellt (Realtime Streams). Eine erste Implementierung gibt es bereits [6]. Ob und wann allerdings Code in den Standardkernel übernommen wird, steht in den Sternen. *(ofr)*

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 19: Linux-Magazin 2/05, S. 88

[2] Jens Axboe, „Linux Block IO – present and future": Proceedings of the Linux Symposium, Volume One, Ottawa Juli 2004, [http://www.finux.org/Reprints/Reprint-Axboe-OLS2004.pdf]

[3] Patch: Time sliced CFQ IO-Scheduler with priorities: [http://www.kernel.org/pub/linux/kernel/people/axboe/patches/v2.6/2.6.11-rc1/]

[4] Quellcode zum Programm »ionice«: [http://lwn.net/Articles/116496/]

[5] Jens Axboe: Benchmarks zum Time Sliced CFQ-IO-Scheduler: [http://lwn.net/Articles/113869/]

[6] Active Block IO Scheduling System: [http://abiss.sf.net]

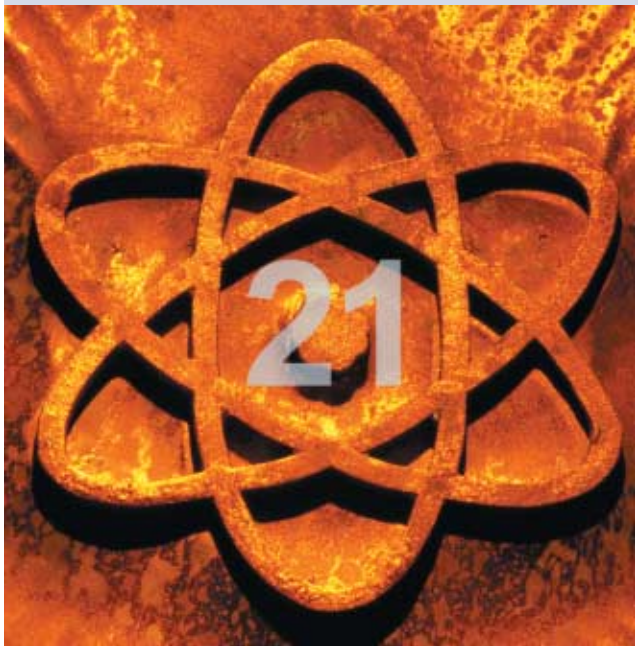
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 21

Kern-Technik

Kernelprogrammierer können auf die effiziente dynamische Speicherverwaltung von Linux bauen. Doch auch für Applikationsentwickler sind Kenntnisse des Buddy-Systems und des Slab-Allokators interessant, denn es stecken trickreiche Algorithmen dahinter. Eva-Katharina Kunst, Jürgen Quade



Das dynamische Management realen und virtuellen Speichers ist eine komplexe Angelegenheit. Linux verwaltet ihn, wie andere Betriebssysteme auch, primär mit so genannten Seiten. Dabei handelt es sich um Speicherbereiche gleicher Größe, die auf einem 32-Bit-System normalerweise 4 KByte, auf einem 64-Bit-System 8 KByte umfassen.

Nicht alle Speicherbereiche besitzen die gleichen Eigenschaften. Nur ein Teil des Speichers ist beispielsweise für ISA-DMA nuztbar [1], ein anderer Teil entzieht sich auf einem x86-System dem direkten Zugriff (Highmem). Aus diesem Grund untergliedert Linux den gesamten, physisch adressierbaren Speicher in die »ZONE_DMA«, »ZONE_NORMAL« und »ZONE_HIGHMEM« und verwaltet den Speicher jeder Zone separat.

Der Kernel bietet es seinen Subsystemen wie dem Pagecache oder dem Sys-Filesystem an, die Speicherseiten einer Zone für den eigenen Gebrauch zu reservieren. Benötigt eine Komponente mehr Speicher als mit einer Page zur Verfügung steht, erhält sie mehrere, direkt aufeinander folgende Seiten. Braucht sie den Speicher nicht mehr, muss sie ihn zurückgeben.

Fragmentierung verhindern

Meist fragmentiert das ständige Reservieren und Freigeben den verfügbaren Speicher: Fordert eine Komponente einen größeren an, gibt es vielleicht nicht mehr genügend direkt aufeinander folgende Speicherseiten. Diese Fragmentierung versucht das Buddy-System (siehe Kasten „Buddy-System“) zu verhindern. Einen Überblick über Funktionen, mit denen Speicher auf Basis der Speicherseiten reserviert wird, gibt der Kasten „Speicherseiten allozieren“.

Das Buddy-System hat im Wesentlichen zwei Nachteile: Erstens ist das Reservieren und Freigeben von Seiten aufwändig. Zweitens verschwendet das Allozieren einer ganzen Page Speicher, wenn das darin abgelegte Objekt nur wenige Bytes groß ist. Kein Wunder also, dass das Buddy-System nur als Basis für eine Speicherverwaltung dient, die auch effizient mit kleineren Objekten umgehen kann. Eine solche Verwaltung bietet zudem die Chance, zusätzlich noch Objekt-Caching zu implementieren. Programmierer sprechen von Objekt-Caching, wenn ein Objekt nicht gleich vollständig zerstört wird, nur weil es aktuell keine Verwendung findet.

Stattdessen verbleibt das Objekt in seinem vorinitialisierten Zustand in einem Cache. Die Vorgänge „Speicher reservieren“ und „Objekt vorinitialisieren“ sind bei einer erneuten Anforderung damit überflüssig, die Vorgänge „Objekt deinitialisieren“ und „Speicher freigeben“ können je einmal eingespart werden (siehe Abbildung 1).

Schneller mit Cache

Objekt-Caching ist nur mit typisierten Objekten zu realisieren, schließlich besitzt jeder Typ spezifische Variablen. Das Speichermanagement kennt also seine Objekte. Die zugreifenden Instanzen fordern nicht mehr eine bestimmte Menge Speicher, sondern das Objekt an. In Linux sorgt eine Speicherverwaltung mit

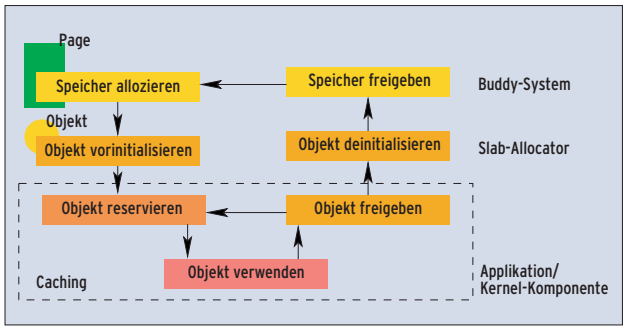


Abbildung 1: Einsatz eines Cache: Der Zugriff auf Objekte ist wesentlich effizienter, wenn diese nach Gebrauch nicht komplett deinitialisiert werden und der zugehörige Speicher reserviert bleibt.

dem Namen Slab-Allocator für den effizienten Umgang mit den kleinen Objekten und für das Objekt-Caching.

Mit Hilfe der Funktion »kmem_cache_create()« (siehe Kasten „Cache-Funktionen für den Kernelprogrammierer“) erzeugt der Programmierer den Cache für einen neuen Objekttyp. Neben der Objektgröße übergibt er zusätzlich die jeweiligen Adressen einer Konstruktor- und einer Destruktor-Funktion. Legt der Cache dann initial ein neues Objekt an, ruft er den Konstruktor auf. Der vom Entwickler programmierte Konstruktor initialisiert die eventuell im Objekt befindliche Spinlocks, Referenzzähler und Variablen.

Zum Löschen eines Cache mit all seinen Objekten ruft der Programmierer die Funktion »kmem_cache_destroy()« auf. Bevor er die Objekte aber wirklich zerstört, ruft der Allokator den Destruktor auf. Dieser gibt zum Beispiel die bei der Initialisierung angeforderten Spinlocks wieder frei. Da der Slab-Allocator immer mehrere Objekte gleichzeitig anlegt beziehungsweise zerstört, ergeben sich Laufzeitvorteile: Der Code des Konstruktors und der des Destruktors muss nicht mehrfach in den Instruktions-Cache des Prozessors geladen werden.

Zwischen dem Aufruf des Konstruktors und dem des Destruktors kann das Objekt beliebig oft vom Cache angefordert (durch Aufruf der Funktion »kmem_cache_alloc()«) und dem Cache zurückgegeben werden (Aufruf der Funktion »kmem_cache_free()«). Der Objekttyp wird hierbei in Form der Cache-Referenz

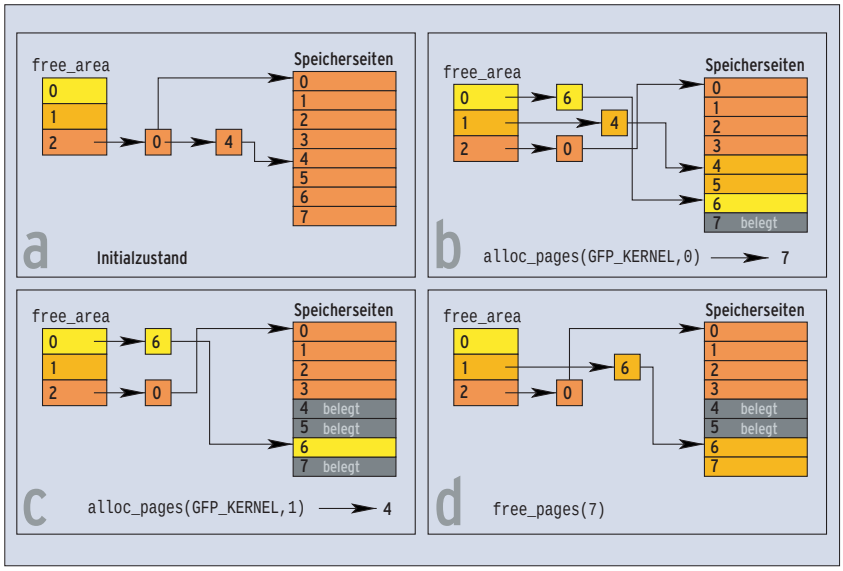


Abbildung 2: Gibt es keinen freien Speicherbereich in der angeforderten Größe, spaltet der Buddy-Algorithmus einen größeren Speicherbereich so lange auf, bis geeignete Speicherbereiche vorliegen.

Buddy-System

Das Buddy-System verwaltet den freien Speicher im Linux-Kernel. Kernelkomponenten fordern zusammenhängenden Speicher nicht durch Angabe der gewünschten MByte, sondern durch Angabe einer so genannten Order an. Der Ausdruck 2^{order} spezifiziert die Anzahl aufeinander folgender Speicherseiten. Benötigt der Programmierer genau eine Speicherseite, übergibt er der Funktion »alloc_pages()« eine Order von 0, denn $2^0 = 1$. Für vier aufeinander folgende Seiten übergibt er 2, für 1024 die 10 und so weiter.

Das Buddy-System setzt über die Konstante »MAX_ORDER« dem Größenwahn ein Limit. Bei einer Order von »MAX_ORDER - 1« ist Schluss – mehr zusammenhängender Speicher ist auf diese Art nicht zu reservieren. Auf einem x86-System liegt diese Grenze bei 1024 Speicherseiten, was bei 4 KByte Seitengröße einem Speicher von 4 MByte entspricht (siehe Kasten „Vmalloc“).

Ordnung im Array

Basis der Implementierung ist ein Array, das für jede Speichergröße »order« die freien zusammenhängenden Speicherbereiche listet. Ein Speicherbereich ist dabei über die erste Page und den Feldindex – dieser entspricht der Order – spezifiziert. Mit jeder Allokation wählt »order« die Liste aus, aus der ein Element (freier Speicherbereich) entnommen wird. Ist die Liste leer, erzeugt das Buddy-System neue Einträge. Das passiert, indem es ein Element des nächstgrößeren Speicherbereichs entnimmt und halbiert.

Speicher vom Kumpel

Abbildung 2 demonstriert an einem einfachen Beispiel das Grundprinzip der Allokation und der Freigabe: Das Buddy-System verwaltet acht Pages, maximal vier aufeinander folgende Seiten sind reservierbar. Im Anfangszustand sind alle Speicherbereiche (zweimal vier Pages) verfügbar (Abbildung 2a).

Fordert jetzt eine Kernelkomponente eine Page an, stellt das Buddy-System fest, dass die Liste für »order = 0«-Requests leer ist. Da am Index 1 ebenfalls keine Einträge vorhanden sind, entnimmt es an Index 2 einen der beiden Einträge (Speicher ab Seite 4) und splittet ihn. Von den beiden neuen Einträgen (4 und 6) wird wiederum einer (hier 6 in 6 und 7) gesplittet und an Index 0 gehängt. Daraufhin kann am Index 0 der angeforderte Speicherbereich entnommen werden (hier 7), ein Eintrag bleibt übrig, in Abbildung 2b ist das Eintrag 6.

Ein zwei Seiten großer, zusammenhängender Speicherbereich lässt sich jetzt im Beispiel direkt anfordern. Der am zugehörigen Index 1 befindliche Bereich wird der anfordernden Komponente übergeben (Abbildung 2c). Bei der Freigabe versucht das Buddy-System sequenzielle Speicherseiten wieder zusammenzufügen. Gibt der Kernel die zuvor reservierte einzelne Page (Nummer 7) wieder frei, verbindet das Buddy-System Page 7 mit der noch freien Page 6 und fügt diese wieder an die Liste mit dem Index 1 ein (Abbildung 2d).

Infos im Proc-Filesystem

Die Proc-Datei »/proc/buddyinfo« gibt für jede Zone die Anzahl der verfügbaren Speicherbereiche an. Diese Bereiche sind dabei gemäß ihrer »order« aufgeschlüsselt:

```
Node 0, zone   DMA      3    5    4    4 ...
Node 0, zone   Normal   180  49   99   76 ...
```

In diesem Beispiel gibt es noch 99 Speicherbereiche mit jeweils einer Länge von $2^{(3-1)} = 4$ Pages (dritte Nummernspalte).

Das Buddy-System ist ein einfacher und effizienter Algorithmus. Da er Fragmentierung dennoch nicht gänzlich verhindern kann, arbeiten die Kernelhacker an seiner Verbesserung. So hat die Erweiterung des Zonenmodells um jeweils eigene Zonen gemäß der Speicheranforderung (»GFP_ATOMIC«, »GFP_KERNEL«, »GFP_USER«) bereits zu einer spürbar besseren Verfügbarkeit großer, zusammenhängender Speicherblöcke geführt [7].

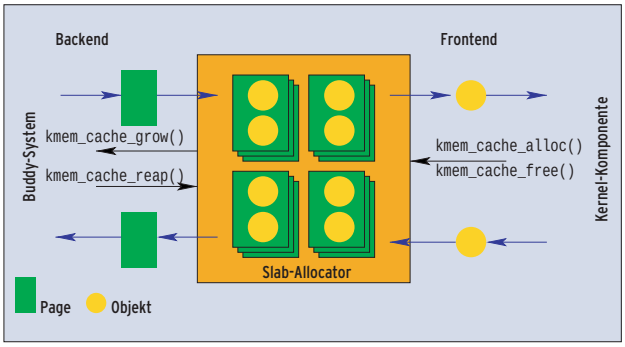


Abbildung 3: Der Slab-Allocator nutzt das Buddy-System als Backend. Die Funktionsaufrufe sind in schwarz dargestellt, der Objektfluss blau.

Speicherseiten allozieren

Mit Hilfe der beiden folgenden Funktionen fordern Kernelprogrammierer vom Buddy-System Speicherseiten an:

- »struct page *alloc_pages(unsigned int gfp_mask, unsigned int *order)«: Der Aufruf der Funktion fordert beim Buddy-System 2^{order} zusammenhängende Speicherseiten an. Eine Verwendung der Maske »gfp_mask« von »GFP_ATOMIC« erlaubt es, die Funktion auch im Interrupt-Kontext zu verwenden.
- »void free_pages(unsigned long addr, unsigned int order)«: Diese Funktion gibt die Speicherseiten ab der Adresse »addr« wieder frei. Alles zusammen umfasst damit der freizugebende Speicherbereich 2^{order} Pages.

beim Aufruf der Funktionen festgelegt. Die Cache-Referenz wiederum hat der Programmierer beim Aufruf von »kmem_cache_create()« in Form des Rückgabewerts erhalten.

Der Slab-Allocator nutzt das Buddy-System als Backend (siehe Abbildung 3). Die beiden Mechanismen sind dabei so verzahnt, dass Objekte nicht nur durch »kmem_cache_destroy()« zerstört werden, sondern auch dann, wenn das Buddy-System „unter Druck steht“. Hat nämlich das Backend für eine Anforderung keinen ausreichend großen Speicherbereich, sammelt es die vom Slab-Allocator gerade nicht benötigten Speicherbereiche wieder ein. Das reicht bereits oft, um die ausstehende Anforderung bedienen zu können. Übrigens sammelt das Buddy-System die Speicherseiten auch dann wieder ein, wenn für einige Sekunden niemand die Objekte im Cache verwendet.

Gut organisiert

Den Einstieg in den Slab-Allocator findet der Kernel über das Objekt »cache_cache« (siehe Abbildung 4). Es stellt den Anfang einer Liste von Objekt-Caches dar. Jeder Cache (»struct cache«) ist für genau einen Objekttyp verantwortlich. Er verwaltet die so genannten Slabs, in denen sich die Objekte befinden. Ein Slab (Kachel) ist ein aus Seiten bestehender Speicherbereich mit jeweils mehreren Objekten.

Ein Cache verwaltet drei Arten solcher Slabs: erstens die Slabs, deren Objekte alle in Gebrauch sind, zweitens die Slabs, deren Objekte partiell verwendet werden, und schließlich die Slabs, deren Objekte zwar initialisiert, aber unbenutzt sind. Durch diese Dreiteilung beschleunigt sich die Reservierung und Freigabe der Objekte.

Wie viele Seiten einen Slab bilden, hängt vom Speicherbedarf eines einzelnen Ob-

Listing 1: »slab.c« – Nutzung des Slab-Allocator

```
01 #include <linux/module.h>
02 #include <linux/completion.h>
03 #include <linux/slab.h>
04 ...
07 static int ThreadID=0;
08 static DECLARE_COMPLETION(OnExit);
09 static kmem_cache_t *cache;
10
11 struct linobj {
12    int dummy_i, dummy_j, dummy_k;
13    char dummy_feld[250];
14 };
15
16 static void linobj_destructor(void *objp,
   kmem_cache_t *cache,
17         unsigned long flags)
18 {
19    printk("linobj_destructor(%p)\n", objp);
20    return;
21 }
22
23 static void linobj_constructor(void *objp,
   kmem_cache_t *cache,
24         unsigned long flags)
25 {
26    if(flags&SLAB_CTOR_VERIFY)
27          return;
28    printk("linobj_constructor(%p)\n", objp);
29    return;
30 }
31
32 static int ThreadCode(void *data)
33 {
34    unsigned long timeout;
35    int i;
36    struct linobj *obj;
37
38    daemonize("linobj-test");
39    allow_signal(SIGTERM);
40    for(i = 0; i < 5; i++) {
41       obj = (struct linobj
   *)kmem_cache_alloc(cache, GFP_KERNEL);
42       printk("objadr=%p\n", obj);
43       timeout=HZ;
44       set_current_state(TASK_INTERRUPTIBLE);
45       timeout=schedule_timeout(timeout);
46       kmem_cache_free(cache, obj);
47       if(timeout)
48          break;
49    }
50    complete_and_exit(&OnExit, 0);
51 }
52
53 static int __init slabInit(void)
54 {
55    cache = kmem_cache_create("linobj",
   sizeof(struct linobj),
56          0, 0, linobj_constructor,
   linobj_destructor);
57    if(!cache)
58       return -EFAULT;
59    ThreadID = kernel_thread(ThreadCode, NULL,
   CLONE_KERNEL);
60    if(ThreadID == 0)
61       return -EIO;
62    return 0;
63 }
64
65 static void __exit slabExit(void)
66 {
67    kill_proc(ThreadID, SIGTERM, 1);
68    wait_for_completion(&OnExit);
69    if(cache)
70       kmem_cache_destroy(cache);
71 }
72
73 module_init(slabInit);
74 module_exit(slabExit);
```

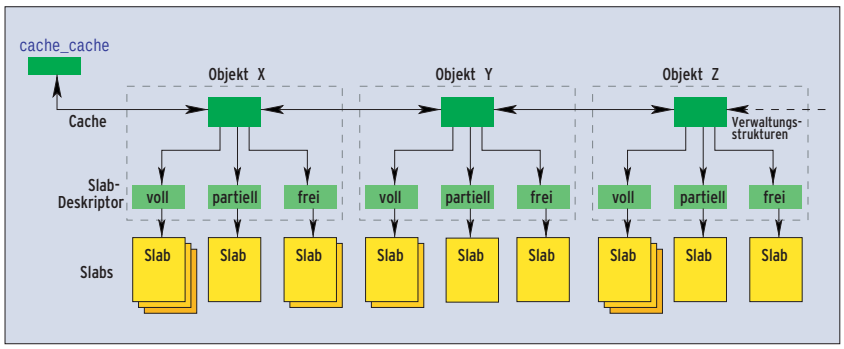


Abbildung 4: Der Slab-Allocator verwaltet mehr als 100 unterschiedliche Kernelobjekte in jeweils eigenen Cache. Jeder Cache besteht aus drei Gruppen von Slabs, in denen sich die Objekte befinden.

Vmalloc

Über das Buddy-System lassen sich Speicherbereiche von maximal 4 MByte reservieren. Wer größere zusammenhängende Speicherbereiche benötigt, verwendet die Funktion »vmalloc()«. Sie hat allerdings den Nachteil, dass die Memory Management Unit (MMU) physisch nicht aufeinander folgende Speicherseiten zu einem zusammenhängenden Speicherbereich umsetzt, was das Umschreiben der Pagetables erfordert. Das wiederum ist jedoch zeitaufwändig und daher möglichst zu vermeiden.

jekts ab. Für kleine Objekte (weniger als 512 Byte) reserviert der Allokator grundsätzlich nur eine Speicherseite. Für größere Objekte blockt er dagegen so viele Seiten, dass der Verschnitt nicht mehr als 12,5 Prozent beträgt (1/8 Page). Verschnitt entsteht allerdings schnell: Würde ein Objekt beispielsweise 2052 Byte belegen, könnte man in einer Page von 4 KByte genau ein Objekt unterbringen. Damit blieben 2044 Byte des Speichers ungenutzt.

Abbildung 5 zeigt, dass bei Verwendung von zwei Pages der ungenutzte Platz pro Objekt mit knapp 679 Byte wesentlich geringer ausfällt. Vom Verschnitt hängt es auch ab, ob die Verwaltungsstruktur für den Slab (der Slab-Descriptor) im Slab selbst steht oder separat gehalten werden muss.

Konformisten

Der Objekt-Cache kommt auch für nicht typisierte Objekte, also für normalen Speicher zum Einsatz. Die bekannten Funktionen »kmalloc()« und »kfree()« basieren ebenfalls auf dem Slab-Allocator. 26 Caches stellen generische Objekte in den Größen 32, 64, 128, 256 und so weiter bis zu maximal 131072 Byte (je einmal aus der »ZONE_NORMAL« und einmal aus der »ZONE_DMA«) bereit. Der Unterschied zwischen den generischen und den typisierten Objekten besteht darin, dass Erstere keine Konstruktor- und keine Destruktor-Funktion zur Verfügung stellen.

Die fixen Objektgrößen der generischen Objekte deuten aber an, dass ein Programmierer, der 132 Byte anfordert, damit zugleich 124 Byte Verschnitt produziert. Benötigt der Entwickler häufiger einen Speicherbereich dieser krummen Größe, ist das Anlegen eines eigenen Objekt-Cache empfehlenswert. Listing 1

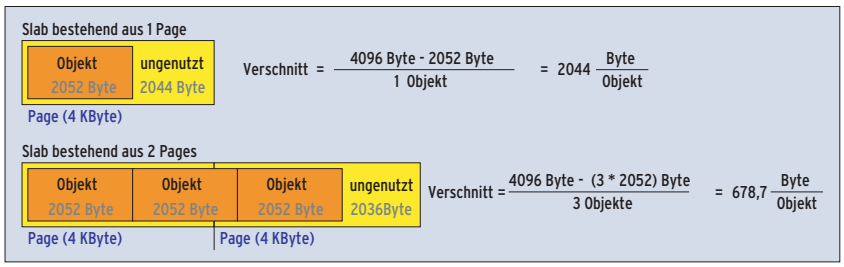


Abbildung 5: Um den Verschnitt klein zu halten, nutzt der Slab-Allocator für Objekte, die größer als 512 Byte sind, mehrere Pages für eine Kachel (ein Slab).

Listing 2: Makefile

```
01 TARGET=slab.o
02
03 ifneq ($(KERNELRELEASE),)
04 obj-m       := ${TARGET}
05
06 else
07 KDIR    := /lib/modules/$(shell uname -r)/build
08 PWD     := $(shell pwd)
09
10 default:
11         $(MAKE) -C $(KDIR)  SUBDIRS=$(PWD) modules
12 endif
```

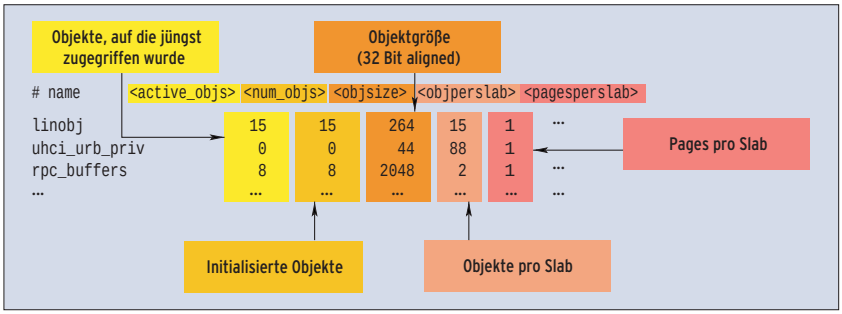


Abbildung 6: Die Proc-Datei »/proc/slabinfo« informiert über Objekttypen, ihren Aufbau und ihre Verwendung.

zeigt, wie man das macht. Insbesondere demonstriert es die Verwendung von Konstruktor und Destruktor.

Der Parameter »objp« ist der Zeiger auf den Objektspeicher, »cache« spezifiziert den Cache und »flags« übergibt die Zugriffsflags. Die Datei » < linux/slab.h > « definiert drei Flags : »SLAB_CTOR_CONSTRUCTOR« ist beim Aufruf des Konstruktors gesetzt, nicht aber beim Aufruf des Destruktors. »SLAB_CTOR_ATOMIC« bedeutet, dass der Konstruktor nicht schlafen darf, und »SLAB_CTOR_VERIFY« bedeutet, dass der Aufruf des Konstruktors nur der Überprüfung dient. Ist dieses Flag gesetzt, kehrt der Konstruktor im Regelfall direkt zurück (siehe Listing 1, Zeile 27).

Mit Hilfe eines Makefile in Listing 2 lässt sich aus dem Code aus Listing 1 ein Kernelmodul bauen. Nach dem Laden per »insmod slab.ko« legt der Slab-Allocator den neuen Cache »linobj« an. Gleichzeitig startet er einen Thread, der fünf Sekunden lang jeweils ein Objekt anfordert und danach wieder freigibt. Spätestens beim Entladen (»rmmod slab«) ruft er den Destruktor auf.

Ein Blick in »/proc/slabinfo« zeigt, dass das Objekt »linobj« 264 Bytes Speicher benötigt. Damit passen 15 dieser Objekte in einen Slab, wobei ein Slab aus einer Page besteht (Abbildung 6).

Solaris' Erbe

Die anderen, hier nicht erwähnten Parameter in »/proc/slabinfo« deuten an, dass der Slab-Allocator noch weitere Raffinessen bietet. Neugierige sollten einen Blick in die Papers des Slab-Erfinders Jeff Bonwick ([2], [3]) werfen, der 1994 diese Technologie in Solaris eingebaut hat. Wer sich mehr für die Linux-Implementierung interessiert, wird in [4], [5] und [6] fündig. Bleibt zum Abschluss noch der Hinweis, dass sich die hier beschriebenen Verfahren nicht nur in diversen Betriebssystemen finden. Auch Applikationen profitieren von der effizienten Technik, zum Beispiel das Speichermanagement von Perl. *(ofr)* ■

Infos

[1] „Kern-Technik", Folge 3: Linux-Magazin 10/03, S. 81
[2] Jeff Bonwick, „The Slab Allocator": Usenix 1994 [http://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.ps]
[3] Jeff Bonwick, „Extending the Slab Allocator to Many CPUs and Arbitrary Resources": Usenix 2001 [http://www.usenix.org/event/usenix01/full_papers/bonwick/bonwick.pdf]
[4] Wolfgang Mauerer, „Linux Kernelarchitektur": Hanser-Verlag 2004
[5] Mel Gorman, „Understanding the Linux Virtual Memory Manager": Prentice Hall 2004
[6] Sven Krohlas, „Der Slab Allocator": [http://www.krohlas.de/computing/slab_allocator.pdf]
[7] Listings und Makefile online: [http://www.linux-magazin.de/Service/Listings/2005/06/Kern-Technik/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Cache-Funktionen für den Kernelprogrammierer

Die folgende Funktion legt ein neues Objekt mit dem Namen »name«, dem Speicherbedarf »size«, der Ausrichtung »align« und den Flags »flags« an:

```
kmem_cache_t * kmem_cache_create (const ➥
char *name, size_t size, size_t align, ➥
unsigned long flags, void (*ctor)(void*,➥
kmem_cache_t *, unsigned long), void ➥
(*dtor)(void*, kmem_cache_t *, unsigned ➥
long))
```

Sobald der Slab-Allocator für diesen Objekttyp einen Slab anlegt, ruft der Kernel für jedes Objekt den Constructor »ctor« auf. Den Destructor »dtor« ruft er für jedes Objekt auf, wenn der Slab aufgelöst wird.

Folgende Flags »flags« steuern das Anlegen der Objekte im Cache:

»SLAB_POISON«: Der Speicher wird mit dem Wert »0xa5« gefüllt. Initialisierungsfehler sind damit leichter zu identifizieren.

»SLAB_RED_ZONE«: Vor und nach jedem Objekt werden Magics (spezielle Zeichenketten) geschrieben. Damit lassen sich Speicherüberläufe und auch der Initialisierungszustand des Objekts überwachen.

»SLAB_NO_REAP«: Das Flag verhindert, dass der Slab-Allocator dem Buddy-System freien Speicher zurückgibt, auch wenn Speicher knapp wird.

»SLAB_HWCACHE_ALIGN«: Die Ausrichtung der Objekte erfolgt gemäß den Hardware-Cache-Reihen. Unter Umständen lässt sich damit unnötiges Flushen des Translation Lookaside Buffers (TLB) verhindern.

Die Verwendung von »kmem_cache_create()« innerhalb einer Interrupt Service Routine (ISR) ist verboten. Weiterhin wichtig: Der Speicher für »name« muss über die komplette Existenz des Objekts hinweg bereitstehen. Die Funktion darf nicht innerhalb einer ISR aufgerufen werden, sie gibt im Erfolgsfall einen Zeiger auf den erzeugten Cache zurück, sonst »NULL«. Diese Funktion löst den Cache »cachep« auf:

```
int kmem_cache_destroy (kmem_cache_t * ➥
cachep)
```

Der Slab-Allocator ruft für jedes Objekt im Cache die Destructor-Funktion auf. Im Erfolgsfall gibt die Funktion »0« zurück, sonst »1«. Die Funktion

```
void * kmem_cache_alloc (kmem_cache_t ➥
*cachep, int flags)
```

reserviert ein Objekt aus dem Cache »cachep«. »flags« kann die Werte »GFP_ATOMIC«, »GFP_KERNEL«, »GFP_USER« und »GFP_DMA« annehmen. Bei Erfolg gibt sie die Adresse des Objekts zurück, sonst »NULL«. Die Funktion

```
vvoid kmem_cache_free (kmem_cache_t ➥
*cachep, void *objp)
```

gibt das Objekt »objp« dem »cachep« zurück.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 22

Kern-Technik

Für Fehlersuche und Performance-Messungen brachte der Linux-Kernel bislang nur wenige praktische Instrumente mit. Mit KProbes und JProbes lassen sich bei Kernel 2.6 Halte- und Messpunkte ohne großen Aufwand in das laufende Betriebssystem einfügen. Eva-Katharina Kunst, Jürgen Quade



Bösen Bugs rückt der Applikations-Programmierer mit einem Debugger zu Leibe. Kernelhackern steht diese Möglichkeit nicht ohne weiteres offen. Zwar gibt es auch für den Linux-Kernel einen Debugger – doch weigert sich Linus Torvalds bis heute standhaft ihn offiziell in den Kernel zu übernehmen. Sein Hauptargument: Der Debugger lenke von der eigentlichen Fehlerursache ab, was dazu führe, dass eben nur Symptome, nicht jedoch das eigentliche Problem gefixt würden. Da zudem der Umgang mit dem extra erhältlichen Kernel-Debugger umständlich ist, haben sich die Programmierer daran gewöhnt, mit »printk()« Debug-Ausgaben einzubauen.

Wenn es um ladbare Module geht, ist Printk praktisch und einfach zu benutzen. Neue Printk-Statements sind schnell eingebaut und wieder entfernt. Das Kompilieren und Neuladen eines Moduls geht ebenfalls sehr schnell. Anders sieht es jedoch bei Code aus, der fester Bestandteil des Kernels ist. Hier muss der Kernel neu generiert und installiert werden. Noch schlimmer: Ein zeitaufwändiger Neustart des Systems ist notwendig.

Automatisierte Breakpoints

Findige Programmierer haben sich bereits vor drei Jahren eine alternative Debug-Methode für den Kernel ausgedacht: so genannte Kernel Probes (KProbes), die aus Kernel-Sicht automatisierte Breakpoints darstellen (auch Probe Points genannt). Damit kann offensichtlich auch Linus Torvalds leben: Er hat die KProbes jedenfalls fest in Kernel 2.6 aufgenommen.

Stößt die CPU bei der Abarbeitung auf einen solchen Punkt, ruft der Kernel dem Probe Point zugeordnete, vom Programmierer zur Verfügung gestellte Funktionen auf. Diese Funktionen protokollieren zum Beispiel durchgeführte Aktionen per »printk()« oder geben per »dump_stack()« einen Stacktrace aus. Allerdings ist es nicht möglich, die von einem Debugger bekannten interaktiven Kommandos zu verwenden. Innerhalb des Kernels wäre das ohnehin problematisch, da das Zeitverhalten des Kernels massiv durcheinander geriete.

Der Informatiker betrachtet ein Kernel Probe als Objekt. Es beherbergt neben der Breakpoint-Adresse (dem Probe Point) vier Methoden (Abbildung 1): die Pre-, Post-, Fault- und Break-Handler. Mit der Instanziierung des Objekts durch die Funktion »register_kprobe()« tauscht der Kernel den Maschinenbefehl (Opcode) an der Probe-Point-Adresse gegen den eines Debugger-Breakpoint aus.

Einstieg über Debug-Interrupt

Auf einer x86-Architektur lautet dieser Maschinenbefehl »INT 3« mit dem Opcode »0xcc«. Es ist der übliche Interrupt, um in einen Debugger zu wechseln (siehe Abbildung 2). Sobald der Prozessor auf diesen Breakpoint trifft, ruft er die Kernelfunktion »do_int3()« auf (Abbildung 3). Diese aktiviert den »pre_handler()«, schaltet Single-Stepping ein und setzt den Instruction Pointer auf den zuvor geretteten Maschinenbefehl, damit nach Beendigung des INT-3-Interrupts der ursprüngliche Maschinenbefehl abgearbeitet wird.

Dank Single-Stepping ruft der Kernel direkt danach die Funktion »do_debug()« auf, die schließlich den »post_handler()« aktiviert. Dann schaltet er das Single-

Bestimmen einer Kerneladresse

In [1] sind insgesamt vier Methoden beschrieben, die die Kerneladresse einer Funktion bestimmen (hier vorgeführt am Beispiel des Systemcall »sys_setuid«):

- Adresse der System-Map entnehmen: »grep sys_setuid /usr/src/linux/System.map«
- Adresse mittels »nm« aus dem Objektfile lesen: »nm /usr/src/linux/vmlinux | grep sys_setuid«
- Adresse über das Proc-Filesystem bestimmen: »cat /proc/kallsyms | grep sys_setuid«
- Adresse mit der Funktion »kallsyms_lookup_name()« beziehen

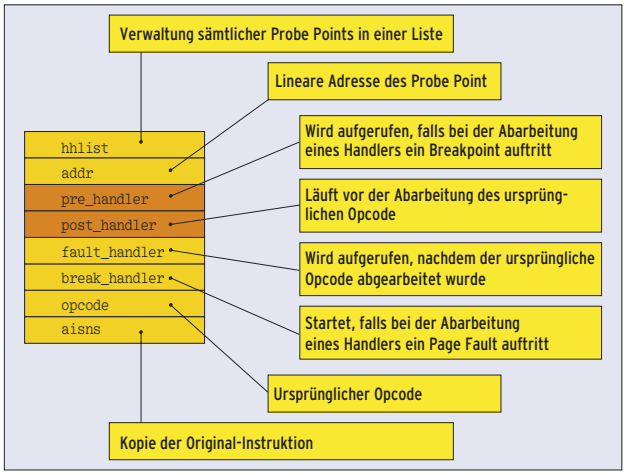


Abbildung 1: Das KProbe-Objekt speichert verschiedene Komponenten, die den Ablauf des Programms beeinflussen, zum Beispiel den »pre_handler« und den »post_handler«, die vor respektive nach der Unterbrechung ablaufen.

Stepping wieder aus und setzt den Instruction Pointer auf den nächsten Opcode der ursprünglich abgearbeiteten Funktion. Tritt während des Ablaufs von Pre- oder Post-Handler ein Page Fault auf, ruft der Kernel den zu KProbe gehörigen Fault-Handler auf. Damit der Kernel diesen Page Fault unbeachtet lässt, sollte »fault_handler()« den Wert »1« zurückgeben.

Die Break-Funktion »break_handler()« rundet die Methodensammlung ab. Stößt der Kernel innerhalb der Probe-Funktion auf einen weiteren Breakpoint, ruft er diese Funktion auf. Normalerweise gibt sie eine »1« zurück: das Zeichen für den Kernel, den Breakpoint unbeachtet zu lassen. Der Post-Handler besitzt übrigens keinen Rückgabewert (»void«). Dagegen sollte der Rückgabewert der Pre-Handler-Funktion grundsätzlich »0« sein.

Um als Entwickler oder als neugieriger Administrator KProbes zu nutzen, muss die entsprechende Option beim Kernel eingestellt sein. Ob dies der Fall ist, lässt sich – ist »Kernel .config support« vorhanden – mit Hilfe von »zcat /proc/config.gz | grep CONFIG_KPROBES« leicht feststellen: Wer als Ergebnis »CONFIG_KPROBES is not set« erhält, muss seinen Kernel neu bauen (Abbildung 4). Er sollte außerdem sicherstellen, dass die Option »CONFIG_CALLSYMS« eingeschaltet ist.

Speicheradressen finden

Sind KProbes einkompiliert, lässt sich bereits mit wenigen Zeilen Code ein Kernel-Probe implementieren. In Listing 1 ist zu sehen, wie man zu Beginn des Systemcall »sys_setuid« einen Pre-Handler installiert. Allerdings läuft der Code unverändert nur auf einem recht aktuellen Kernel (ab Version 2.6.12-rc1). Die Kerneladresse der zu überprüfenden Funktion erfragt das Modul nämlich durch den Aufruf von »kallsyms_lookup_name()«. Ab Version 2.6.12-rc1 exportiert der Kernel diese Funktion standardmäßig für GPL-konforme Module.

Weitere Methoden, um die Kerneladresse zu bestimmen, beschreibt der Kasten „Bestimmen einer Kerneladresse“. Wer Kernel-Probes bei einem älteren Kernel einsetzen möchte, muss entweder auf eine dieser alternativen Methoden zurückgreifen oder den Kernel patchen: Dazu ist das Modul »/usr/src/linux/kernel/kallsyms.c« um die Zeile »EXPORT_SYMBOL_GPL(kallsyms_lookup_name)« zu ergänzen.

Ist für das übergebene Symbol keine Adresse bestimmbar, beendet sich das Modul mit einem Fehlercode. Andernfalls instanziiert der Kernel das Probe-Objekt. Jedes Mal, bevor er den Systemcall »sys_setuid« aufruft, kommt also die hier implementierte Funktion »pre_probe()« zum Zuge (siehe Listing 1). Sie meldet sich im Syslog und erhöht einen Zähler, dessen Stand das Modul beim Laden ausgibt. Dazu muss der Syslog-Daemon natürlich richtig konfiguriert sein, siehe [2].

Zum Test ist das Modul mit Hilfe des Makefile (Listing 2) zu übersetzen und danach mit »insmod ksetuid.ko« zu laden. Die kurze Beispielapplikation aus Listing 3 ruft zum Testen den Systemcall auf. Das Programm lässt sich mit »make setuid« kompilieren.

Ruft man das generierte Programm mit »./setuid 1000« auf, erscheint im Syslog eine entsprechende Nachricht:

```
tail -1 /var/log/messages
May 17 21:22:37 mobil kernel: pre_probe 1
```

Unabhängig davon, ob es sich um die Pre- oder die Post-Funktion handelt, ist innerhalb der Kernel-Probe-Funktion der

Listing 1: »ksetuid.c«

```
01 #include <linux/module.h>
02 #include <linux/kprobes.h>
03 #include <linux/kallsyms.h>
04
05 static int call_count = 0;
06
07 static int pre_probe(struct kprobe *p, struct
   pt_regs *regs)
08 {
09    ++call_count;
10    printk("pre_probe %d\n", call_count);
11    return 0;
12 }
13
14 static struct kprobe kp = {
15    .pre_handler = pre_probe,
16 };
17
18 static int __init probe_init(void)
19 {
20    kp.addr = (kprobe_opcode_t *) kallsyms_
   lookup_name("sys_setuid");
21
22    if (kp.addr == NULL) {
23       printk("kallsyms_lookup_name could not
   find address"
24              "for the specified symbol name\
   n");
25       return 1;
26    }
27    if( register_kprobe(&kp)==-ENOSYS ) {
28       printk("kprobes not supported\n");
29       return 1;
30    }
31    printk("kprobe registered address %p\n",
   kp.addr);
32    return 0;
33 }
34
35 static void __exit probe_exit(void)
36 {
37    unregister_kprobe(&kp);
38    printk("probe-breakpoint called %d times.\
   n", call_count);
39 }
40
41 module_init( probe_init );
42 module_exit( probe_exit );
43 MODULE_LICENSE("GPL");
```

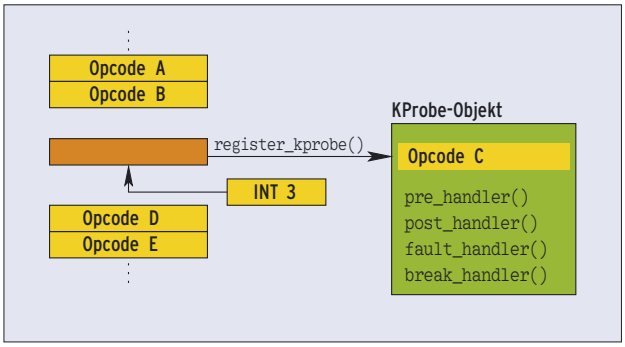


Abbildung 2: Der Maschinenbefehl an der zu untersuchenden Kernelcode-Adresse wird für KProbes mit dem Opcode »INT 3« ausgetauscht, der für einen Debugger-Breakpoint steht. Das KProbe-Objekt nimmt den alten Opcode auf.

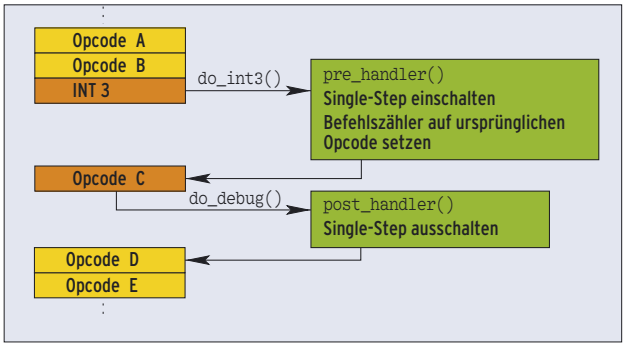


Abbildung 3: Ablauf bei Verwendung eines Kernel-Probe: Zu Beginn schaltet der Kernel im »pre_handler()« den Prozessor in den Einzelschrittmodus (Single-Step). Der »post_handler()« schaltet den Modus wieder aus.

Zugriff auf alle globalen Daten möglich, auf lokale Variablen allerdings nicht. Immerhin erhält der Programmierer mit den JProbes Zugriff auf die Übergabeparameter einer Funktion. Dazu implementiert er nur einen einzigen Handler, dessen Übergabeparameter jenen der zu untersuchenden Funktion entsprechen. Einen Rückgabewert hat diese Handler-Funktion nicht. Stattdessen muss sie mit einem Aufruf von »jprobe_return()« abschließen.

Listing 2: Makefile zu »ksetuid.c«

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := ksetuid.o
03 else
04 KDIR    := /lib/modules/$(shell uname -r)/build
05 PWD     := $(shell pwd)
06
07 default:
08         $(MAKE) -C $(KDIR)  SUBDIRS=$(PWD) modules
09 endif
10
11 clean:
12         rm -f *.mod.c *.o *.ko
```

Test mit Systemcall

Als Beispiel für die auf den KProbes basierenden JProbes dient wieder »sys_setuid«. Diese Kernelfunktion bekommt die jeweils zu setzende UID als Integerwert übergeben. Listing 4 zeigt den Quellcode, der ein JProbe auf diesen Systemcall setzt. Ruft man nach dem Laden des Moduls mit »insmod jsetuid.ko« das Kommando »setuid« mehrmals mit unterschiedlichen Parametern auf, protokolliert Syslog das mit:

```
# ./setuid 1000
setting to uid 1000
# tail -1 /var/log/messages
May 23 20:34:24 mobil kernel: jsetuid(1000)
# setuid 1001
# tail -1 /var/log/messages
May 23 20:34:57 mobil kernel: jsetuid(1001)
```

Während JProbes nur zu Beginn einer Funktion gesetzt werden können, lassen sich mit KProbes beliebige Codestellen instrumentieren. Die gesuchte Stelle zu lokalisieren erfordert – wenn auch nur in sehr beschränktem Umfang – Assembler-Kenntnisse: Man findet den Ort im disassemblierten Code der entsprechenden Funktion.

Listing 3: »setuid.c«

```
01 #include <stdio.h>
02
03 int main( int argc, char **argv )
04 {
05    int uid;
06
07    if( argc != 2 ) {
08       printf("usage: %s uid\n", argv[0] );
09       return -1;
10    }
11    uid = atoi( argv[1] );
12    printf("setting to uid %d\n", uid );
13    if( setuid( uid )!=0 )
14       perror("setuid");
15    return 0;
16 }
```

Ein bisschen Assembler

Auch dieses Verfahren ist am besten anhand eines Beispiels zu erklären: Ein Probe Point in »sys_setuid()« soll nur dann aufgerufen werden, wenn jemand ohne ausreichende Berechtigung diesen Systemcall ausführt.

Listing 4: »jsetuid.c«

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <linux/kprobes.h>
04 #include <linux/kallsyms.h>
05
06 static void jsetuid( uid_t uid )
07 {
08    printk("jsetuid( %d )\n", uid );
09    jprobe_return();
10 }
11
12 static struct jprobe jp = {
13    .entry   = (kprobe_opcode_t *) jsetuid,
14 };
15
16 static int __init probe_init(void)
17 {
18    jp.kp.addr = (kprobe_opcode_t *) kallsyms_
   lookup_name("sys_setuid");
19    if( jp.kp.addr == NULL ) {
20       printk("kallsyms_lookup_name could not
   find address"
21           "for the specified symbol name\n");
22       return 1;
23    }
24    register_jprobe(&jp);
25    printk("jprobe registered\n");
26    return 0;
27 }
28
29 static void __exit probe_exit(void)
30 {
31    unregister_jprobe(&jp);
32    printk("jprobe unregistered\n");
33 }
34
35 module_init( probe_init );
36 module_exit( probe_exit );
37 MODULE_LICENSE("GPL");
```

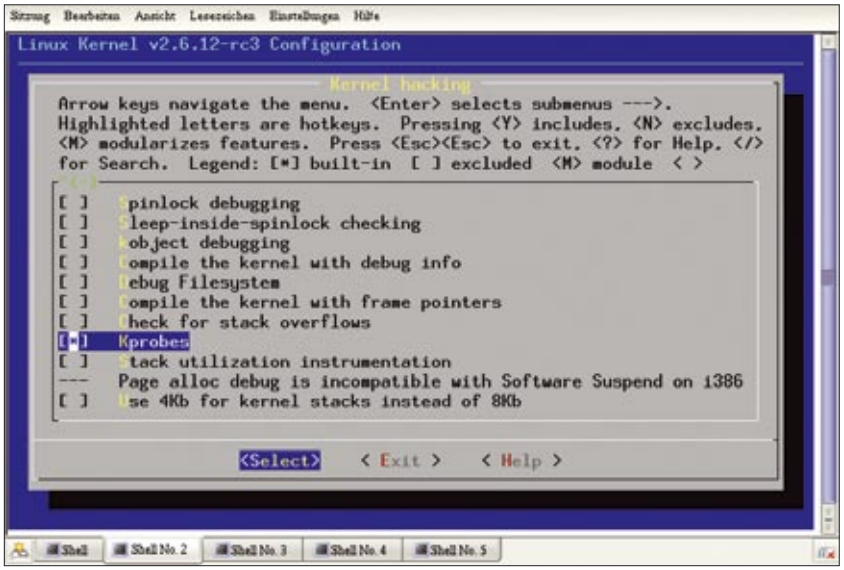


Abbildung 4: Um Kernel-Probes für eigene Zwecke zu nutzen, muss in der Kernelkonfiguration unter dem Menüpunkt »Kernel-Hacking« die Option »KProbes« aktiviert sein.

Der Quellcode der Funktion »sys_setuid()« findet sich in der Datei »kernel/sys.c«. Listing 5 zeigt den relevanten Ausschnitt daraus. Falls beim Aufruf der Funktion die Zugriffsreche nicht ausreichen, gibt der Kernel »-EPERM«, also »-1« zurück. In diesem Fall soll er den Pre-Handler ausführen. Zum Setzen des Probe Points muss der Entwickler die Adresse dieser markanten Stelle suchen, indem er das zugehörige Assembler-Listing zu Rate zieht. Folgender Befehl disassembliert das Objektfile:

```
objdump -d /usr/src/linux/kernel/sys.o >➥
/tmp/sys.asm
```

Listing 6 zeigt den relevanten Teil der dissassemblierten Funktion, wie er in der Datei »/tmp/sys.asm« zu finden ist. An Adresse »d9d« findet sich der Wert »-1« (Hexadezimal 0xffffffff): Das ist also die Adresse für den Probe Point.

Probe Points in ladbaren Modulen

Vorsicht ist bei Probe Points in dynamischen Kernelmodulen geboten. Entlädt man nämlich das zu untersuchende Modul vor dem Modul mit dem Probe Point (das Modul mit dem KProbe), kommt es unweigerlich zu einem Crash (Oops). Schließlich greift das Kernel-Probe-Modul auf Code zu, der schon längst wieder entladen ist.

Doch Abhilfe ist möglich. Dazu muss das KProbe-Modul dem Kernel mitteilen, dass es auf das zu untersuchende Modul zugreift. Das geschieht mit der Funktion »try_module_get()«. Allerdings braucht man dazu die Adresse des Modulobjekts »struct file_operations«. Stammt das Modul aus der eigenen Feder, lässt sich diese Adresse am einfachsten per »EXPORT_SYMBOL_GPL()« exportieren. Das KProbe-Modul kann sie dann einfach wieder importieren.

Natürlich müssen nach getaner Arbeit die Ressourcen wieder freigegeben werden. Das geschieht durch den Aufruf von »module_put()«. Listing 7 zeigt die wesentlichen Codestücke im KProbe-Modul.

Der nächste Schritt bestimmt den Offset dieser Adresse gegenüber dem Anfang der Funktion »sys_setuid«. Dazu subtrahiert man von »0xd9d« die Anfangsadresse der Funktion »0xce0«. Das Ergebnis (»0xb7«) – also die Offsetadresse – wird zur Adresse von »sys_setuid« im Kernel (»kp.addr«) addiert. Der Anfang der Funktion »probe_init()« sieht dann so aus (Ausschnitt aus »ksetuid2.c«):

```
static int __init probe_init(void)
{
    kp.addr = (kprobe_opcode_t *) ➥
    kallsyms_lookup_name("sys_setuid");
    kp.addr+= 0xb7;
    if (kp.addr == NULL) {
```

Der vollständige Code des Moduls »ksetuid2.c« steht unter [9]. Wer das Modul ausprobieren möchte, sollte nicht vergessen im Quellcode die Offsetadresse »0xb7« der Funktion gegen jene des eigenen Systems auszutauschen.

Nach dem Kompilieren des modifizierten Moduls und dem Laden mit »insmod ksetuid2.ko« folgt der Test. Ruft der Superuser über den Befehl »su« intern den Systemcall »sys_setuid« auf, sind die Zugriffsrechte korrekt und der Probe

Listing 5: Ausschnitt aus »sys_setuid()«

```
01 asmlinkage long sys_setuid(uid_t uid)
02 {
03    int old_euid = current->euid;
04    int old_ruid, old_suid, new_ruid, new_suid;
05    int retval;
06
07    ...
08    if (capable(CAP_SETUID)) {
09        if (uid != old_ruid && set_user(uid, old_euid
   != uid) < 0)
10                return -EAGAIN;
11        new_suid = uid;
12    } else if ((uid != current->uid) && (uid !=
   new_suid))
13        return -EPERM;
14    ...
15 }
```

Listing 6: »sys_setuid« disassembliert

```
01 00000ce0 <sys_setuid>:
02      ce0:    55                      push   %ebp
03      ce1:    b8 00 e0 ff ff          mov    $0xffffe000,%eax
04      ce6:    57                      push   %edi
05      ce7:    56                      push   %esi
06      ce8:    53                      push   %ebx
07      ce9:    83 ec 10                sub    $0x10,%esp
08      cec:    8b 5c 24 24             mov    0x24(%esp),%ebx
09 ...
10      d95:    5d                      pop    %ebp
11      d96:    c3                      ret
12      d97:    39 eb                   cmp    %ebp,%ebx
13      d99:    74 a3                   je     d3e <sys_setuid+0x5e>
14      d9b:    39 f3                   cmp    %esi,%ebx
15      d9d:    ba ff ff ff ff          mov    $0xffffffff,%edx
16      da2:    74 9a                   je     d3e <sys_setuid+0x5e>
17      da4:    eb e7                   jmp    d8d <sys_setuid+0xad>
18      da6:    8d 76 00                lea    0x0(%esi),%esi
19      da9:    8d bc 27 00 00 00 00    lea    0x0(%edi),%edi
```

Point wird nicht aufgerufen. Erst wer mit einer anderen UID versucht die User-ID zu verändern, erhält eine Meldung vom Probe Point:

```
# ./setuid 1000
setting to uid 1000
# su mail
sh-3.00$ ./setuid 1000
setting to uid 1000
setuid: Operation not permitted
...
# tail -1 /var/log/messages
May 17 21:34:45 mobil kernel: pre_probe(1)
```

Die Beispiele zeigen, wie über KProbes bereits mit wenigen Zeilen Code wertvolle Debuginformationen zu gewinnen sind. Da die Technik – abhängig von der Anzahl und der Komplexität der eingesetzten Probe Points – im Normalfall kaum Performance-Einbußen mit sich bringt, eignet sie sich auch für den Einsatz in einem Produktivsystem.

In diesem Fall lassen sich auch statistische Informationen gewinnen, zum Beispiel die Aufrufreihenfolge und -häufigkeit von spezifischen Kernelfunktionen. So helfen KProbes auch bei der Optimierung von Kernelalgorithmen und Applikationen. Leider laufen KProbes und JProbes (noch) nicht auf jeder Hardware. Zu den unterstützten Plattformen zählen x86 (sowohl 32 als auch 64 Bit), Sparc und PowerPC (jeweils nur 64 Bit).

Ergänzungen zu KProbes

Inzwischen sind drei Projekte entstanden, die den Umgang mit den Kernel-Probes vereinfachen wollen: DProbes [5], System Tap [6] und das noch junge KProbes-HERE [7]. Während mit KProbes-HERE das Debugging direkt per Kommandozeile erfolgt, definiert der Entwickler bei den beiden anderen Projekten die Probe Points über eine eigene Skriptsprache. Diese übersetzt ein Compiler in ein ladbares Modul – eine Technik, die übrigens dem DTrace-Mechanismus in Solaris 10 ähnelt [8]. *(ofr)*

Infos

[1] Prasanna Panchamukhi, „Kernel debugging with KProbes“: [http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnxw04Kprobe]
[2] Eva-Katharina Kunst, Jürgen Quade, Kern-Technik, Folge 3: Linux-Magazin 10/03, S. 81
[3] William Cohen, „Gaining insight into the Linux kernel with KProbes“: [http://www.redhat.com/magazine/005mar05/features/kprobes/]
[4] Sudhanshu Goswani, „An introduction to KProbes“: [http://lwn.net/Articles/132196/]
[5] Linux Dynamics Probes: [http://dprobes.sf.net]
[6] System Tap: [http://sources.redhat.com/systemtap]
[7] Tejun Heo, KProbes-HERE [http://home-tj.org/kphere]
[8] Suns DTrace: [http://www.sun.com/bigadmin/content/dtrace]
[9] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2005/08/Kern-Technik].

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 7: »ksetuid.c« mit Module Usage Count

```
01 ...
02 #include <linux/fs.h>
03 ...
04 extern struct file_operations fops; // "fops" muss exportiert sein
05 ...
06
07 static int __init probe_init(void)
08 {
09    ...
10    kp.addr = (kprobe_opcode_t *) kallsyms_lookup_name("driver_read");
11    if (kp.addr == NULL) {
12       printk("kallsyms_lookup_name could not find address"
13              "for the specified symbol name \n");
14       return 1;
15    }
16    try_module_get( fops.owner );
17    register_kprobe(&kp);
18    printk("kprobe registered\n");
19    return 0;
20 }
21
22 static void __exit probe_exit(void)
23 {
24    module_put( fops.owner );
25    unregister_kprobe(&kp);
26    printk("kprobe unregistered\n");
27    printk("generic_make_request() called %d times.\n", call_count);
28 }
29 ...
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 23

Kern-Technik

Inodes und Dentries entschlüsselt: Das virtuelle Filesystem im Kernel lässt Applikationen auf unterschiedlichste Dateisysteme einheitlich zugreifen. Dieser Artikel erklärt die Grundlagen, entschlüsselt Inodes und Dentries und zeigt, wie man ein eigenes Filesystem implementiert. Eva-Katharina Kunst, Jürgen Quade



In Sachen Dateisysteme ist Linux ein echtes Multitalent: Die Anzahl der von Linux unterstützten Filesysteme ist riesig, wie der Befehl »cat /proc/filesystems« auf der Konsole offenbart. Neben den Klassikern wie Ext 3, Reiser, JFS, NTFS oder FAT32 finden sich diverse virtuelle Kollegen. Proc-FS, Sys-FS und UBS-FS sind dabei nur die bekanntesten. Hinzu kommen noch Dateisysteme zur Verschlüsselung oder das „Filesystem in der Datei“ (Loop).

Ordnung durch Abstraktion

Eine einheitliche Schnittstelle im Kernel macht diese Vielzahl für Programmierer beherrschbar. Die technische Lösung für eine solche Anforderung lautet Abstraktion. Die Kernelprogrammierer haben das Modell eines universalen Filesystems definiert, auf das sich die unterschiedlichsten Dateisysteme abbilden lassen.

Dieses Modell definiert, dass eine Datei aus Datenblöcken zusammengesetzt ist und eine Dateigröße, einen Besitzer, eine Gruppe und Zugriffsrechte hat (Abbildung 1). Außerdem gibt es unterschiedliche Dateiarten wie normale Dateien, Gerätedateien oder Verzeichnisdateien (Directories). Hinzu kommen noch Daten über die Zeitpunkte des Anlegens, der letzten Modifikation sowie des letzten Zugriffs auf das File.

Alle diesen Informationen stehen im so genannten Inode, der zentralen Datenstruktur jedes Unix-Dateisystems. In diesem Modell ist jedes Directory selbst eine Datei, deren Inhalt den Datei- oder Verzeichnisnamen einem Inode zuordnet (Abbildung 2).

Inodes und Dentries

Wie die Daten auf die einzelnen Blöcke eines Dateisystems verteilt werden oder wo die Inode-Informationen liegen, ist Sache des jeweiligen Filesystems (Abbildung 3). Die Implementierung bildet also das reale Dateisystem auf das VFS ab. Ein Ext-2-Filesystem orientiert sich dabei mit seinen Meta-Informationen sehr am VFS-Modell (genau genommen haben sich die Entwickler am realen FS orientiert). Ein FAT-Filesystem kennt dagegen beispielsweise keine Inodes. Also generiert der Filesystem-Code die Inodes allein auf Basis der vorhandenen Informationen. Spezifische Zugriffsrechte, die das FAT-Filesystem nicht kennt, ersetzt er dabei durch Defaultwerte.

Um auf ein Dateisystem zugreifen zu können, muss zunächst der Inode des Root-Verzeichnisses bekannt sein. Zumindest im Modell ordnet er die Subdirectories und die im Wurzelverzeichnis abgelegten Dateien den jeweils zugehörigen Inodes zu. Eine einzelne derartige Zuordnung zwischen Datei- respektive Verzeichnisname und Inode wird als Dentry (Directory Entry) bezeichnet (Abbildung 4).

Schneller mit Dcache

Erst die Kenntnis des Inode erlaubt den Zugriff auf die Daten der Datei. Die wichtigste Operation des VFS ist daher das Auffinden eines Inode, wenn ein vollständiger Dateiname gegeben ist. Dazu zerlegt es den gegebenen Dateinamen zunächst in seine Bestandteile. Aus »/usr/bin/vim« wird dabei »/«, »usr«, »bin« und »vim«. Im Folgenden sucht das VFS dann für die Pfadkomponente »usr« im Wurzelverzeichnis »/« (dessen Inode ja bekannt ist) nach dem zugehörigen Inode.

Da das VFS-Modell nicht festlegt, wie (und ob überhaupt) die Daten in einer Verzeichnisdatei abgelegt sind, muss jedes Dateisystem die Suchfunktion »lookup()« zur Verfügung stellen. Sie gibt den zur Pfadkomponente gehörigen »dentry« oder einen Fehlercode zurück. Wenn es den Dentry kennt, sucht das Filesystem im Beispielfall die Komponente »bin« im Verzeichnis »usr«. Mit jeder weiteren Pfadkomponente wiederholt es diesen Vorgang, bis es den gesuchten Dentry (und damit den Inode) gefunden hat.

Aus Performancegründen sucht der Kernel nicht mit jeder Pfadkomponente

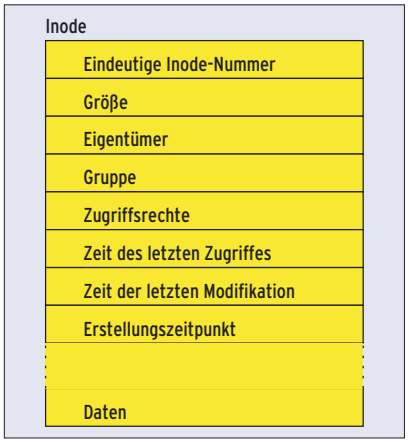


Abbildung 1: Neben den eigentlichen Daten gehören zu jeder Datei vielfältige Attribute wie Größe, Eigentümer, Gruppe und so weiter.

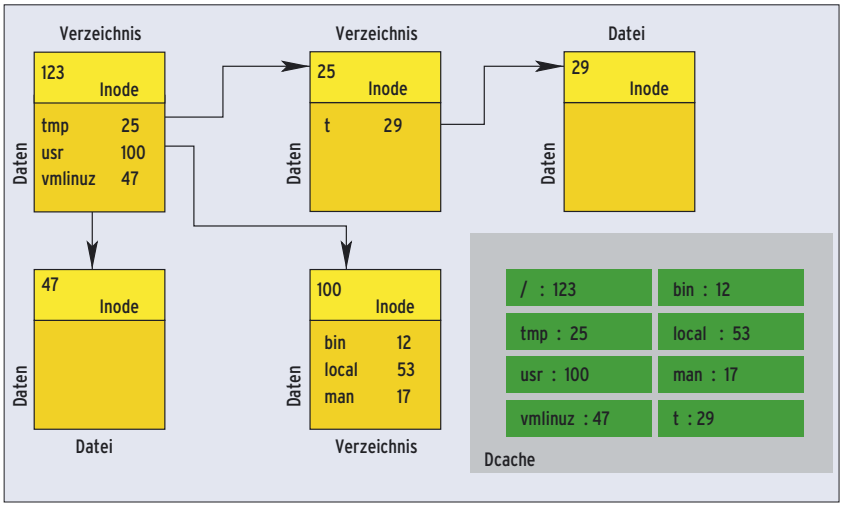


Abbildung 2: Inodes und Dentries realisieren die Filesystem-Hierarchie.

den zugehörigen Inode aufs Neue. Er speichert die Suchanfrage – bestehend aus dem darüber liegenden Inode und der Pfadkomponente – zusammen mit dem Ergebnis (dem zugehörigen Inode) in einem Dentry-Objekt. Die Dentries wiederum speichert er im so genannten Dcache »dentry_cache«.

Bei einer erneuten Anfrage sucht er zuerst im Cache. Erst wenn er hier keinen Dentry findet, beauftragt er den Filesystem-Code mit der Suche. Natürlich lassen sich im Dcache nicht unendlich viele Dentries ablegen. Nach dem Least-Recently-Used-Algorithmus (LRU) ersetzt der Kernel die jeweils ältesten Einträge durch neuere.

Hashes statt Suche

Den für die Suche notwendigen Vergleich zwischen der Pfadangabe im Verzeichnis und der gesuchten Pfadkomponente beschleunigt ein Hash-Algorithmus. Vereinfacht ausgedrückt ersetzt ein solches Verfahren einen teuren Suchalgorithmus durch eine einfache und damit schnelle Berechnung. Dazu trägt der Kernel im Dentry sowohl den realen Namen als auch dessen Länge sowie den zugehörigen Hashwert ein.

Im Detail ist das hier vorgestellte VFS-Modell natürlich etwas komplexer, schließlich unterstützt es auch noch Gerätedateien, Named Pipes und Links. Hardlinks sind dabei sehr einfach zu realisieren. Für sie legt der Kernel einfach einen weiteren Dentry an. Softlinks wiederum sind Pfadnamen, die nicht einem Inode, sondern einem anderen Pfadnamen zugeordnet sind.

Die beschriebenen Grundlagen reichen bereits aus, um praktische Erfahrungen durch die Implementierung eines eigenen virtuellen Filesystems zu sammeln. Der Betriebssystemkern unterstützt den Programmierer bei der Erstellung seines Filesystems nicht nur durch das skizzierte Modell, sondern zusätzlich mit einigen Hilfsfunktionen. Die meisten von ihnen sind in der Bibliothek »libfs.a« (in den Kernelquellen in »/usr/src/linux/fs/libfs.c«) zu finden. ▶

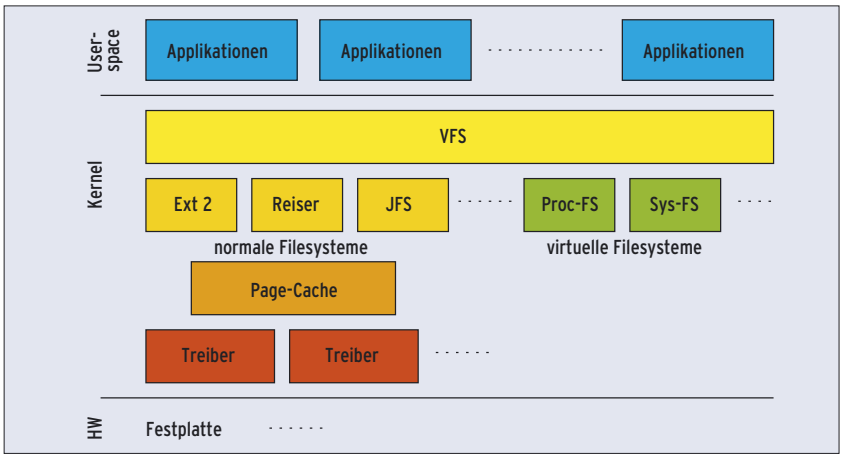


Abbildung 3: Beim Zugriff auf Daten sind mehrere Komponenten des Kernels beteiligt, die schichtartig aufeinander aufbauen. Unten im Kernel befinden sich die Gerätetreiber, ganz oben das VFS.

Das hier vorgeführte Kernelmodul in Listing 1 implementiert ein Filesystem-Objekt, für jedes dann wirklich gemountete Filesystem einen Superblock und für jede Datei ein Inode-Objekt. Zusätzlich braucht das Beispiel die beschriebenen Dentry-Objekte, die Inodes und Dateinamen einander zuordnen.

Ein neues Dateisystem registriert sich beim Betriebssystemkern über »register_filesystem()«. Parameter dieser Funktion ist erwartungsgemäß das Filesystem-Objekt, welches das neue Dateisystem typisiert: Es legt also den Namen (zum Beispiel »ext2«) fest und stellt eine Methode zur Verfügung, mit der der Kernel den Superblock für eine Instanz des Filesystems erhält.

Zeile 109 in Listing 1 ordnet dem zugehörigen Feld ».get_sb« die Funktion »get_superblock()« (Zeile 99) zu. Diese Methode wird typischerweise beim Mounten des Filesystems aufgerufen. Sie bekommt unter anderem den Gerätenamen übergeben, also zum Beispiel »/dev/hda1«. Bei virtuellen Dateisystemen ist der Gerätename nicht relevant, es hat sich daher eingebürgert, beim Mounten dafür das Schlüsselwort »none« zu verwenden.

Die vom Kernel bereitgestellte Funktion »get_sb_single()« reserviert den Speicher für den Superblock und führt die Grundinitialisierung durch. Die Beispielfunktion »fill_superblock()« schließt die Initialisierung ab (Listing 1, Zeile 70).

Listing 1: »fs.c«

```
001 #include <linux/module.h>
002 #include <linux/init.h>
003 #include <linux/pagemap.h>
004 #include <linux/fs.h>
005
006 static ssize_t read_hello_world(struct file *, char *, size_t , loff_t *);
007
008 static struct super_operations superblock_ops = {
009     .statfs     = simple_statfs,
010     .drop_inode = generic_delete_inode,
011 };
012
013 static struct file_operations file_ops = {
014     .read = read_hello_world,
015 };
016
017 static ssize_t read_hello_world(struct file *instanz, char *buf,
018     size_t tocopy, loff_t *offset)
019 {
020     char *text = "hello world\n";
021     int c, len=strlen(text)+1;
022
023     if( tocopy<len )
024         tocopy=len;
025     if( (c=copy_to_user(buf,text,len))<0 )
026         return -EFAULT;
027     return tocopy-c;
028 }
029
030 static struct inode *make_new_inode(struct super_block *sb, int mode)
031 {
032     struct inode *ret = new_inode(sb);
033
034     if (ret) {
035         ret->i_mode = mode;
036         ret->i_uid = ret->i_gid = 0;
037         ret->i_blksize = PAGE_CACHE_SIZE;
038         ret->i_blocks = 0;
039         ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
040     }
041     return ret;
042 }
043
044 static struct dentry *create_file(struct super_block *sb,
045     struct dentry *parent, char *name)
046 {
047     struct qstr qname;
048     struct inode *inode;
049     struct dentry *dentry;
050
051     inode = make_new_inode(sb, S_IFREG | 0644);
052     if( !inode )
053         return 0;
054     inode->i_fop = &file_ops;
055     inode->i_size = 14; // strlen("hello world\n")+1;
056
057     qname.name = name;
058     qname.len = strlen(name);
059     qname.hash = full_name_hash(name, qname.len);
060
061     dentry = d_alloc(parent, &qname);
062     if( !dentry ) {
063         iput( inode );
064         return 0;
065     }
066     d_add(dentry, inode);
067     return dentry;
068 }
069
070 static int fill_superblock(struct super_block *sb, void *data, int silent)
071 {
072     struct inode *root;
073     struct dentry *root_dentry;
074
075     sb->s_blocksize = PAGE_CACHE_SIZE;
076     sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
077     sb->s_magic = 0x20040804;
078     sb->s_op = &superblock_ops;
079
080     // Inode fuer das Root-Verzeichnis anlegen...
081     root = make_new_inode(sb, S_IFDIR | 0755);
082     if( !root )
083         return -ENOMEM;
084     root->i_op = &simple_dir_inode_operations;
085     root->i_fop = &simple_dir_operations;
086     root_dentry = d_alloc_root(root);
087     printk("root_dentry->name: %s\n", root_dentry->d_iname );
088     if( !root_dentry ) {
089         iput( root );
090         return -ENOMEM;
091     }
092     sb->s_root = root_dentry;
093
094     create_file( sb, root_dentry, "hello" );
095
096     return 0;
097 }
098
099 static struct super_block *get_superblock(struct file_system_type *fst,
100     int flags, const char *devname, void *data)
101 {
102     printk("get_superblock (%s, %s)...\n", fst->name, devname);
103     return get_sb_single(fst, flags, data, fill_superblock);
104 }
105
106 struct file_system_type hello={
107     .owner              = THIS_MODULE,
108     .name               = "hello",
109     .get_sb             = get_superblock,
110     .kill_sb    = kill_litter_super
111 };
112
113 static int __init mod_init(void)
114 {
115     return register_filesystem(&hello);
116 }
117
118 static void __exit mod_exit(void)
119 {
120     unregister_filesystem(&hello);
121 }
122
123 module_init( mod_init );
124 module_exit( mod_exit );
125 MODULE_LICENSE("GPL");
```

Beim Unmounten des Filesystems ruft der Kernel die Methode »kill_sb()« auf. Wenn das Filesystem keine besonderen De-Initialisierungen durchführen muss, kann der Programmierer einfach auf die ebenfalls vorhandene Funktion »kill_litter_super()« zurückgreifen.

Jede Filesystem-Instanz benötigt nicht nur den Superblock, sondern auch einen Inode und ein »dentry«-Objekt für das Wurzelverzeichnis. Beides legt das Beispiel beim Mounten an. Das Filesystem soll eine (virtuelle) Datei enthalten, die beim Lesen den String »Hello World« zurückgibt. »struct file_operations« ordnet dem Inode dieser Datei die Lesefunktion »read_hello_world()« zu (Zeile 13). Die Implementierung der »struct file_operations«-Funktionen hatte die zweite Kern-Technik-Folge vorgestellt [1]. Im Ernstfall müsste man für die Inodes noch die beschrieben »lookup()«-Funktionen implementieren. Stattdessen erzeugt das Beispiel direkt das »dentry«-Objekt und legt es über die Funktion »d_add()« im Dcache ab (Zeile 66).

Listing 1 zeigt den vollständigen Code des virtuellen Dateisystems. Der Name des Filesystemtyps lautet »hello« (Zeile 108). Der Code und das zugehörige Makefile ist unter [2] zu finden. Ist das kompilierte Modul geladen, mountet der Befehl »mount -t hello none /mnt« das virtuelle Filesystem.

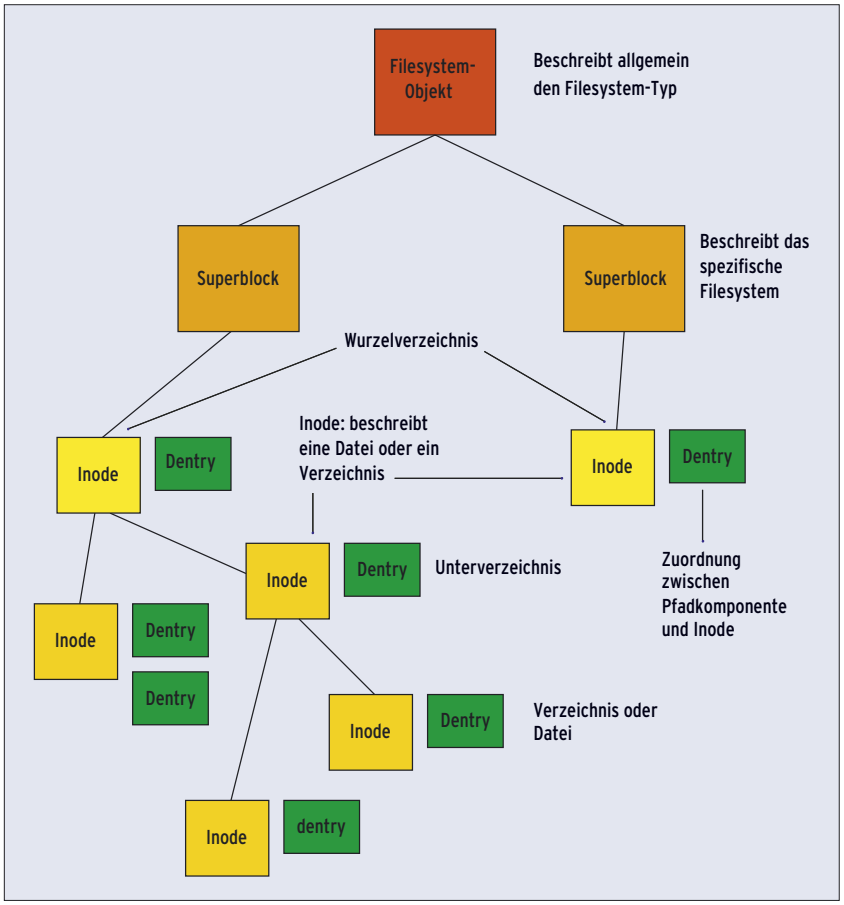


Abbildung 4: Das VFS-Modell hat vier verschiedene Objekte, Inodes und Dentries realisieren die Hierarchie.

Ein »ls -l /mnt« zeigt die Datei »hello«, die sich per »cat« lesen lässt. Der Einfachheit halber ist »hello« als unendliche Datei realisiert, also als eine, die beim Lesen nie ein End-of-File liefert. Der Befehl »cat /mnt/hello« lässt sich mit [Strg] + [C] abbrechen.

Nicht stehen bleiben

Wer nach dieser Einführung auf den Geschmack gekommen ist, kann versuchen die Moduldatei »hello« um eine Schreibfunktion zu erweitern. Danach könnte die Implementierung eines Unterverzeichnisses auf dem Plan stehen. Hilfestellung dazu leisten [3], [4] und [5]. Ein weiterer wichtiger Ausgangspunkt für eigene Dateisystem-Projekte ist der offene Quellcode des Kernels, vor allen Dingen das von Linus Torvalds selbst erstellte Template »/usr/src/linux/fs/ramfs/inode.c«. *(ofr)*

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 2: Linux-Magazin 9/03, S. 86
[2] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2005/10/Kern-Technik/]
[3] Robert Love, „Linux Kernel Development, Second Edition": Novell Press 2005
[4] Wolfgang Mauerer, „Linux Kernelarchitektur": Hanser Verlag 2004
[5] Jonathan Corbet, „Creating Linux virtual Filesystems": [http://lwn.net/Articles/57369/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 24

Kern-Technik

Wer per Konsole Dateien manipuliert, erwartet auch, dass das korrespondierende Konqueror-Fenster daneben die mitbekommt. Konqueror pollt dazu nicht das Verzeichnis, sondern setzt das Filemonitoring des Kernels ein. Besonders effizient geht das mit dem neuen Inotify. Eva-Katharina Kunst, Jürgen Quade



Neue Kernel, neue Features. Zwei Monate dauerten die Arbeiten am Kernel 2.6.13, der mit einem ganzen Reigen überarbeiteter und neuer Funktionen auftrumpft: von A wie Alsa bis Z wie Zeitbasis. Eine wichtige Novität ist Inotify (Inode Notify, [1]). Es ersetzt das aus älteren Kerneln bekannte Dnotify. Beide sind für das Filesystem-Monitoring zuständig, bei dem der Kernel die Applikationen über Modifikationen im Dateisystem informiert.

Anwendungen hierfür gibt es reichlich, wenn beispielsweise Nautilus oder Konqueror die grafische Anzeige zu aktualisieren haben, sobald jemand im Hintergrund Dateien löscht oder neu anlegt. Oder ein Security-Monitor erkennt bösartige Manipulationen an Dateien und das neue Suchprogramm Beagle [2] aktualisiert Suchanfragen on the fly. Auch ein Backup-System könnte nicht zu festen Zeitpunkten, sondern immer dann sichern, sobald sich Dateien geändert haben. Solche Applikationen pollen dazu nicht im Dateisystem nach Veränderungen, sondern lassen sich effizienter durch den Kernel – genauer durch Inotify – informieren.

Nicht nur Verzeichnisse

Ein Ersatz für Dnotify war aus mehreren Gründen notwendig. Zunächst ermöglicht Dnotify nur die Überwachung ganzer Verzeichnisse, nicht jedoch von einzelnen Dateien. Jedes per Dnotify überwachte Verzeichnis benötigt zudem einen eigenen Filedeskriptor. Bei vielen überwachten Verzeichnissen ist das Limit der für einen Rechenprozess erlaubten Filedeskriptoren schnell erreicht.

Operationen auf die überwachten Verzeichnisse, beispielsweise Löschen oder Verschieben, bekommt die Applikation per Signal kundgetan. Signale mit ihren Tücken sind bei den Entwicklern nicht sonderlich beliebt. Die Liste der Kritikpunkte ließe sich fortsetzen und gipfelt in der Unmöglichkeit, das überwachte Verzeichnis zu mounten. Der überwachende Rechenprozess greift wie beschrieben über seinen Filedeskriptor auf das Verzeichnis zu und belegt es damit. Im Zeitalter von USB und Wechselmedien ist das nicht tragbar.

Inotify dagegen fußt darauf, dass die Applikation den Kernel ein Inotify-Objekt instanzieren lässt. Das Objekt bekommt die Überwachungspunkte (also Verzeichnisse und Dateien) – die so genannten Watches – übergeben. Ob sich an einem der Watches eine Veränderung ergeben hat, erfährt die Applikation über die bekannten Datei-Zugriffsfunktionen »read()«, »select()« oder »poll()«.

Damit schmiegt sich der Mechanismus harmonisch in das Unix-System ein und ermöglicht gerade deshalb einen problemlosen Unmount-Event. Hängt nämlich jemand ein Dateisystem mit einer überwachten Datei aus, empfängt die Applikation den Unmount-Event und der Kernel entfernt den Watch.

Inotify mit nur vier Funktionen handhaben

So nützlich, wie Inotify für Applikationen ist, so einfach ist es einsetzbar. Neben »read()« hält der Kernel drei weitere neue Systemcalls bereit:

- Mit »int inotify_init(void)« fordert die Applikation einen Filedeskriptor zur Überwachung von Verzeichnissen und Dateien an. Der Systemcall instanziert das Inotify-Objekt. Rückgabewert ist der Filedeskriptor, mit dem »read()«-, »select()«- und »poll()«-Aufrufe möglich sind.
- »int inotify_add_watch(int *fd*, const char **Path*, unsigned int *Mask*)« übergibt dem Kernel die zu überwachenden Verzeichnisse und Dateien (»*Path*«) mitsamt der Eventmaske (»*Mask*«). Die Eventmaske spezifiziert, welche Ereignisse der Applikation zu melden sind, zum Beispiel das Öffnen oder Schließen einer Datei. Tabelle 1 listet die möglichen Operationen auf. Der Systemcall gibt einen

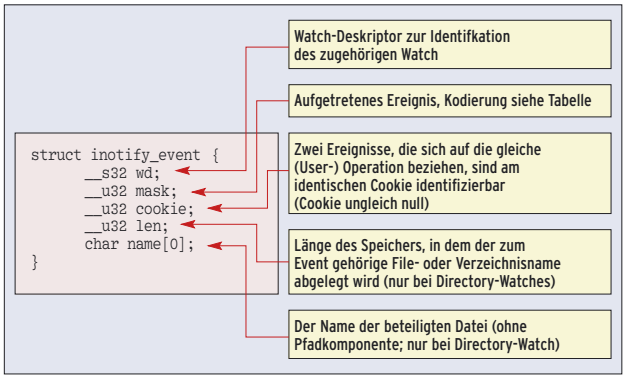


Abbildung 1: Der Kernel schickt der Applikation ein nach dieser Struktur aufgebautes Inotify-Event-Objekt.

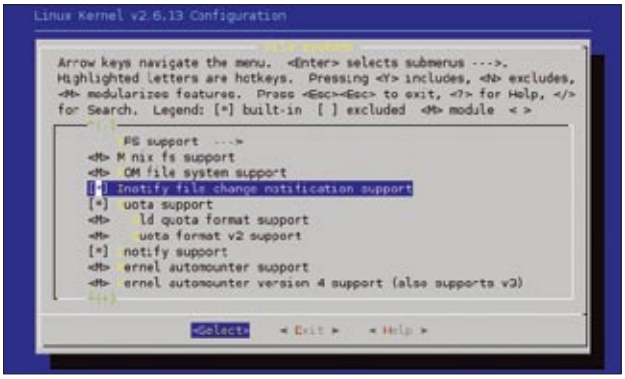


Abbildung 2: Die Option für Inotify findet sich in der Kernelkonfiguration unterhalb »File systems«.

Watch-Deskriptor zurück. Dieser referenziert den Watch.

- »int inotify_rm_watch (int *fd*, int *wd*)« entfernt den über »*wd*« referenzierten Watch.
- Der mehrfach erwähnte Systemcall »int read(int *fd*, char **buf*, int *Maxsize*)« holt die einzelnen Ereignisse ab. Die Anwendung übergibt dazu einen ausreichend dimensionierten Speicherbereich »*buf*«. Wenn niemand die Zugriffsart durch Aufruf der Funktion »fcntl()« von „blockierend“ auf „nicht blockierend“ geändert hat, schläft der Rechenprozess, bis einer oder auch mehrere der Kernel-Events eintreffen. Der Kernel speichert jeden Event in einer Struktur vom Typ »struct inotify_event«, deren Aufbau Abbildung 1 zeigt. Im übergebenen Speicher »*mbuf*« liegen so viele derartige Strukturen, wie Events vorhanden sind respektive solange noch Platz im Speicher ist. Da direkt am Ende der Struktur »struct inotify_event« unter Umständen ein an der Operation beteiligter Dateiname hängt, ist die Länge nicht fix (siehe Listing 1, Zeile 90).

Tabelle 1: Standard-Inotify-Flags und -Events

Define	Bedeutung
IN_ACCESS	Zugriff auf die Datei
IN_MODIFY	Modifikation der Datei
IN_ATTRIB	Veränderung von Meta-Information
IN_CLOSE_WRITE	Ein zum Schreiben geöffnetes File ist geschlossen
IN_CLOSE_NOWRITE	Ein zum Lesen geöffnetes File ist geschlossen
IN_OPEN	Öffnen der Datei
IN_MOVED_FROM	Eine Datei ist aus dem Verzeichnis verschoben
IN_MOVED_TO	Eine Datei ist ins Verzeichnis verschoben
IN_CREATE	Anlegen einer Datei
IN_DELETE	Löschen einer Datei im Verzeichnis
IN_DELETE_SELF	Löschen der Datei oder des Verzeichnisses
IN_MOVE_SELF	Verschieben der Datei oder des Verzeichnisses
Events, die an alle Inotify-User geschickt werden	
IN_UNMOUNT	Filesystem ist ausgehängt
IN_Q_OVERFLOW	Zu viele Events
IN_IGNORED	Ignorierte Datei
Spezielle Flags	
IN_ISDIR	Der Event bezieht sich auf ein Verzeichnis
IN_ONESHOT	Nur einmal verschickter Event
Hilfsdefinitionen	
IN_CLOSE	IN_CLOSE_WRITE | IN_CLOSE_NOWRITE
IN_MOVE	IN_MOVED_FROM | IN_MOVED_TO
IN_ALL_EVENTS	Die Oder-Verknüpfung der Standard-Events

Neben »read()« lassen sich selbstverständlich auch die Aufrufe »select()« und »poll()« verwenden.

Bei der Glibc ist zurzeit nichts zu holen

Die aktuelle Version der Standard-C-Bibliothek kennt die nagelneuen Systemcalls noch nicht. Für den regelmäßigen Leser der Kern-Technik ist das aber kein Problem. Schließlich hat bereits die Folge 13 [3] gezeigt, wie Systemcalls auch ohne Glibc funktionieren. Einzig die Headerdateien des Kernels 2.6.13 müssen zum Kompilieren greifbar sein – hier findet der Compiler die Systemcall-Nummern. Den Pfad zu den Kernel-Headerdateien reflektiert das zugehörige Makefile.

Wer einen Kernel 2.6.13 oder jünger hat, in dem »Inotify« einkompiliert ist (siehe Abbildung 2), kann das File-Monitoring mit den wenigen Zeilen Code von Listing 1 ausprobieren. Zum Übersetzen eignet sich folgendes Makefile:

```
CFLAGS=-g -Wall -I/usr/src/linux-2.6.13/➥
include
all: inotify
```

Die entstandene Applikation startet auf der Konsole unter Angabe des Verzeichnisses oder der Datei, die sie überwachen soll. Wer dies beispielsweise für das Temp-Directory tun will, gibt

```
./inotify /tmp/ &
```

ein. Abbildung 3 zeigt das Programm in Aktion. Die drei gelb markierten Zeilen sind Benutzereingaben, die eine Datei kopieren oder bewegen. Die roten Markie-

rungen betreffen die ausgelösten Events, wie sie Tabelle 1 bezeichnet. Hier ist der Unterschied zwischen dem Kopieren zwischen Verzeichnissen und dem Verschieben innerhalb gut zu sehen.

Drei neue Objekte-Klassen im Kernel

Inotify ist im Kernel geradlinig implementiert. Insgesamt gibt es drei neue Objekte (Datenstrukturen): Inotify-Device, Inotify-Watch und Inotify-Event. Der Kernel legt beim Aufruf von »init_inotify()« ein Inotify-Device-Objekt für die Applikation an. Es dient als Container für die Watch-Objekte, die der Aufruf von »inotify_add_watch()« erzeugt. Eine zusätzliche Referenz auf das Watch-Objekt entsteht in der Inode der Datei respektive des Verzeichnisses, die der Watch durch die Angabe des Pfades überwachen soll.

Wenn eine Applikation eine Datei-Operation bemüht, ruft der VFS [4] die Funktion »inotify_inode_queue_event()« auf. Sie prüft, ob es einen Watch zur Inode gibt. Falls ja, erzeugt Inotify ein Event-Objekt und übergibt es dem Inotify-Device-Objekt. Sollte ein Rechenprozess auf dem Inotify-Device-Objekt schlafen, weckt der Kernel ihn auf. Der Aufruf »read()« übergibt schließlich die gequeuten Inotify-Events der Applikation.



Abbildung 3: Das Testprogramm meldet jede Datei-Operation. Das Verschieben von Dateien (»IN_MOVE_*«) machen (hier grüne) Cookies deutlich.

Außer den VFS-Funktionen ist auch die »generic_shutdown_superblock()« manipuliert. Bei ihrem Aufruf aktiviert sich die Funktion »inotify_super_block_umount()«, die alle Inodes des zugehörigen Filesystems durchgeht. Ist dort ein Watch referenziert, erzeugt sie einen Watch-Event vom Typ »IN_UMOUNT« und zerstört bei der Gelegenheit gleich das Inotify-Watch-Objekt. Wenn der

Listing 1: File-Monitoring mit »inotify.c«

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <sys/syscall.h>
05 #include <linux/inotify.h>
06 #include <asm/unistd.h>
07
08 // Funktions-Wrapper fuer die Systemcalls
09 static inline int inotify_init (void)
10 {
11     return syscall(__NR_inotify_init);
12 }
13
14 static inline int inotify_add_watch (int fd,
   const char *name, __u32 mask)
15 {
16     return syscall(__NR_inotify_add_watch, fd,
   name, mask);
17 }
18
19 static inline int inotify_rm_watch (int fd,
   __u32 wd)
20 {
21     return syscall(__NR_inotify_rm_watch, fd,
   wd);
22 }
23
24 // Eigentlicher Programmcode
25 static char *watch_event( int event )
26 {
27     switch( event ) {
28     case IN_ACCESS:
29         return "IN_ACCESS";
30     case IN_MODIFY:
31         return "IN_MODIFY";
32     case IN_ATTRIB:
33         return "IN_ATTRIB";
34     case IN_CLOSE_WRITE:
35         return "IN_CLOSE_WRITE";
36     case IN_CLOSE_NOWRITE:
37         return "IN_CLOSE_NOWRITE";
38     case IN_OPEN:
39         return "IN_OPEN";
40     case IN_MOVED_FROM:
41         return "IN_MOVED_FROM";
42     case IN_MOVED_TO:
43         return "IN_MOVED_TO";
44     case IN_CREATE:
45         return "IN_CREATE";
46     case IN_DELETE:
47         return "IN_DELETE";
48     case IN_DELETE_SELF:
49         return "IN_DELETE_SELF";
50     case IN_MOVE_SELF:
51         return "IN_MOVE_SELF";
52     case IN_UNMOUNT:
53         return "IN_UNMOUNT";
54     case IN_Q_OVERFLOW:
55         return "IN_Q_OVERFLOW";
56     case IN_IGNORED:
57         return "IN_IGNORED";
58     }
59     return "unknown";
60 }
61
62 int main( int argc, char **argv )
63 {
64     int fd, watch_descriptor, len, ret;
65     struct inotify_event *ie;
66     char buf[1000];
67
68     if( argc != 2 ) {
69         printf("usage: %s path\n", argv[0] );
70         exit( -1 );
71     }
72     fd = inotify_init();
73     if( fd<0 ) {
74         exit( -2 );
75     }
76     watch_descriptor = inotify_add_watch(fd,
   argv[1],
77         IN_ALL_EVENTS|IN_UMOUNT);
78     while( 1 ) {
79         len = read( fd, buf, sizeof(buf) );
80         ie = (struct inotify_event *)&buf;
81         while( len>0 ) {
82             printf("wd=%04x, %16.16s (0x%08x),"
83                 " cookie=%04x, len=%04x ",
84                 ie->wd, watch_event(ie->mask),
85                 ie->mask, ie->cookie, ie->len);
86             if( ie->len )
87                 printf("name=\"%s\"",
   ie->name);
88             putchar( '\n' );
89             len -= sizeof(struct inotify_
   event)+ie->len;
90             ie = (struct inotify_event *)
   ((char*)ie + sizeof(struct inotify_event)
   +ie->len);
91         }
92     }
93     ret = inotify_rm_watch (fd, watch_
   descriptor);
94     return 0;
95 }
```

File-Monitoring-Applikationen

Desktop-Systeme verlangen schon lange nach effizienten Methoden, mit denen sich der Zustand von Geräten, Verzeichnissen und Dateien nahezu in Echtzeit verfolgen lässt. Dnotify und insbesondere das neue Inotify machen das sonst fällige Polling überflüssig, lösen aber nicht das mögliche Problem, dass mehrere Applikationen auf dasselbe Objekt einen Watch setzen. Zu viele Watches führen zu einem Overhead im Kernel.

Die erste Lösung dieses Problems gelang SGI mit dem File Alteration Monitor. FAM besteht aus einem Server, der unter Linux auf Dnotify (auf Irix heißt der Mechanismus Imon) aufsetzt, und der Client-Bibliothek Libfam. Die Client-Bibliothek realisiert transparent für den Programmierer die Kommunikation mit dem Server [5]. Weiterer Vorteil von FAM: Im Gegensatz zu Dnotify ist die FAM-Schnittstelle plattformunabhängig.

Eine Entwicklergruppe rund um den Gnome-Hacker Daniel Veillard entwickelte FAM weiter zu Gamin [6]. Sie legten bei ihrer Entwicklung besonderen Wert auf Sicherheit und eine vereinfachte Codebasis. Zudem nutzt der Gamin-Server den vorgestellten Inotify-Mechanismus. Doch mit der Einfachheit des Interface zu Inotify schmelzen die Vorteile von FAM und Gamin dahin. Erste Applikationsprogrammierer setzen direkt auf Inotify auf und degradieren damit Gamin & Co. zu Kandidaten für die Rumpelkammer.

Inotify ist übrigens nicht in der Lage, neue Geräte zu erkennen. Hierfür sind weiterhin Hotplug und der Hardware Abstraction Layer (HAL) mit dem Hal-Daemon »hald« zuständig [7].

Kernel ein Inotify-Objekt erzeugt, überprüft er zugleich, ob das Flag »IN_ONESHOT« gesetzt ist. Wenn ja, löscht der Betriebssystemkern das zugehörige Watch-Objekt. Dieses Feature hatten sich die Samba-Entwickler erbeten.

Zum Schluss ein Wermutstropfen: Im Kernel 2.6.13 funktionieren leider (noch) nicht alle Features. Das betrifft den »IN_UMOUNT«-Event ebenso wie das »IN_ONESHOT«-Flag. *(jk)* ■

Infos

[1] Dokumentation zu Inotify in den Kernelquellen: »/usr/src/linux-2.6.13/Documentation/filesystems/inotify.txt«
[2] Beagle: [http://beaglewiki.org/Main_Page]
[3] E.-K. Kunst und J. Quade: „Kern-Technik", Folge 13, Linux-Magazin 08/04, S. 92
[4] E.-K. Kunst und J. Quade: „Kern-Technik", Folge 23, Linux-Magazin 10/05, S. 90
[5] FAM: [http://oss.sgi.com/projects/fam/]
[6] „Gamin the File Alteration Monitor": [http://www.gnome.org/~veillard/gamin/]
[7] Hardware Abstraction Layer HAL: [http://freedesktop.org/Software/hal]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Linux-Anfängen Open-Source-Fans. Unter dem Titel „Linux Treiber entwickeln" haben sie ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 25

Kern-Technik

Langes Warten bei Linux-Neustarts kann ziemlich lästig sein. Schuld sind die vielen Stufen des Bootprozesses. Der Kexec-Trick des Kernels umgeht sie und spart damit etwas Bootzeit ein. Als zusätzliches Schmankerl macht er Crash-Dumps möglich. Eva-Katharina Kunst, Jürgen Quade



Zwei Anwendergruppen profitieren besonders von der Beschleunigung des Reboots: erstens Entwickler, die ihren neuen Kernel zum Testen booten, und zweitens jene Admins, die einen neuen Kernel installiert und mit möglichst kurzer Downtime gebootet haben wollen. Hierbei können rund 15 einsparbare Sekunden durchaus ein gewichtiges Argument sein, wenn jeder Stillstand des Servers eine Menge Geld kostet. Mit dem Kexec-Mechanismus [1] bietet Linux eine simple Möglichkeit an, einen Reboot schneller zu gestalten.

Restart im Kernel

Die Kexec-Patches gibt es bereits seit über drei Jahren. Mit Kernel 2.6.13 hat Linus Torvalds den Mechanismus endlich in den Standardkernel übernommen. Wurde ursprünglich nur die I-386-Architektur unterstützt, so beherrscht Linux diesen Mechanismus mittlerweile auch auf den AMD64- sowie PPC-, PPC64-, IA64- und S390-Architekturen.

Die hier vorgestellte Technik lässt sich außerdem nutzen, um im Fall eines Kernelabsturzes einen Crash-Dump zu schreiben. Diesen Speicherabzug erstellt ein im Speicher geparkter Kernel, der die Daten entweder auf die lokale Platte schreibt oder per Netzwerk auf einen anderen Rechner überträgt.

Mit Hilfe solcher Daten identifizieren die Linux-Entwickler die Ursache für den Absturz und steigern längerfristig die Stabilität des Kernels. Diese Technik firmiert in der Linux-Welt unter dem Namen Kdump (siehe Kasten „Crash-Dumps“).

Das Bios beim Booten übergehen spart bei jedem Neustart etwas Zeit. Die obere Hälfte der Abbildung 1 zeigt die Abläufe im Rechner beim normalen Boot- respektive Reboot-Vorgang. Dabei lädt das Bios den ersten Block eines bootbaren Mediums – meist den ersten Block der Festplatte – und startet das dort in den ersten 422 Bytes abgelegte Programm, den First Stage Bootloader. Der interpretiert die ebenfalls im ersten Block abgelegte Partitionstabelle.

Die Partitionstabelle enthält unter anderem die Information, in welcher Partition sich ein weiterer Bootloader befindet, nämlich jener, der das zu startende System lädt. Dieser so genannte Second Stage Bootloader lädt schließlich den Betriebssystemkern und übergibt ihm die Programmkontrolle.

Der untere Teil in Abbildung 1 skizziert den verkürzten Reboot-Vorgang. Das Bios sowie die beiden Bootloader sind nicht mehr beteiligt. Stattdessen parkt der Kexec-Mechanismus den zu ladenden Kernel an einem freien Speicherbereich im Hauptspeicher.

Kommando-Übergabe

Soll ein Reboot das geparkte Betriebssystem aktivieren, übernimmt Kexec das Kommando und überschreibt den Code des zuletzt aktiven Betriebssystemkerns mit dem geparkten Kernel. Ist dieser Vorgang abgeschlossen, übergibt Kexec ihm die Kontrolle: Das neue System bootet. Der Umweg über die Parkposition ist notwendig, weil der eigentlich für den Kernelcode vorgesehene Speicher vom gerade aktiven Kernel belegt ist.

Vor dem Genuss des schnellen Reboots muss der Admin zunächst das System für den schnellen Kernelwechsel vorbereiten. Ist dies erledigt, muss Kexec vor dem eigentlichen Reboot noch einen neuen Kernel im Speicher ablegen und danach aktivieren. Abbildung 2 zeigt die Vorbereitung des Systems.

Der beschleunigte Reboot setzt einen aktuellen Kernel voraus, der das Kexec-Feature unterstützt. Bei der Kernelkonfiguration sind die folgenden drei Optionen zu setzen:

- »CONFIG_KEXEC = y«: Diese Option integriert den Kexec-Systemcall in den Kernel. Sie findet sich in der Konfiguration unter dem Menüpunkt »Processor type and features« (siehe Abbildung 3).
- »CONFIG_PHYSICAL_START = 0x 100000« ist ebenfalls unter »Processor type and features« zu finden. Sie legt

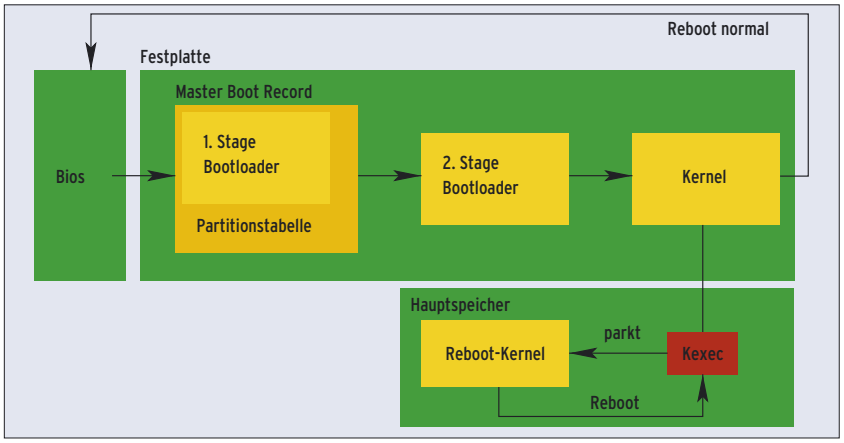


Abbildung 1: Am gewöhnlichen Bootvorgang auf dem PC sind neben dem Bios zwei Bootloader beteiligt, bevor der Linux-Kernel die Kontrolle übernimmt. Kexec übergeht das Bios und die beiden Bootloader-Stages, indem es einen im Speicher geparkten Kernel umschichtet und dann startet.

die Adresse fest, ab der der Kernel geladen wird. Üblicherweise ist die Option richtig konfiguriert (siehe Abbildung 3).

- »CONFIG_SYSFS = y«: Diese unter »Pseudo filesystems« befindliche Einstellung aktiviert das Gerätemodell und die Dateisystem-Schnittstelle (das Sys-Filesystem).

Nach der Kernelkonfiguration erfolgen das Generieren und das Installieren des Kernels. Wer sich dabei unsicher fühlt, findet die für Linux 2.6 passenden Informationen unter [4].

Nach der Installation des Kernels erfolgt im zweiten Schritt die Installation des Userspace-Programms »kexec« [5]. Eine Hilfe für die Generierung und Installation sucht man im Paket allerdings vergeblich, doch die bekannte Befehlsfolge »./configure && make && make install« führt auch hier zu dem gewünschten Ergebnis, das sich anschließend in »/usr/local/sbin/« findet.

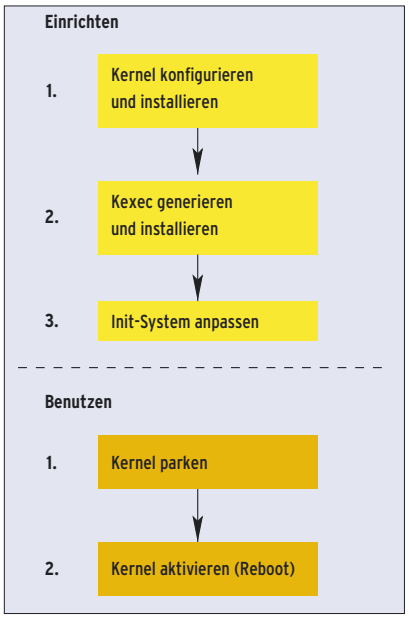


Abbildung 2: Schritt für Schritt zum Kexec-Reboot. Vor dem eigentlichen Neustart steht die gründliche Vorbereitung.

Reboot-Skript anpassen

Der letzte Einrichtungsschritt ist notwendig, damit nach dem Herunterfahren des Systems statt des Bios der geparkte Kernel startet. Hierzu muss der Sysop das Init-System anpassen. Die Änderung ist minimal und betrifft allein das Skript »/etc/init.d/reboot«. Bei einer Ubuntu-Distribution beispielsweise baut er den Kexec-Aufruf folgendermaßen vor dem Kommando »reboot« ein:

```
log_begin_msg "Rebooting... "
/usr/local/sbin/kexec -e
reboot -d -f -i
log_end_msg $?
```

Kernel in Parkposition

Hatte Kexec vor dem Reboot einen Kernel geparkt, ruft es ihn vor dem sonst stattfindenden Übergang zum Bios auf. Liegt im Speicher kein nachzuladender Betriebssystemkern, kommt der bekannte Bootprozess über das Bios und die Bootloader zum Einsatz. Der Aufruf von »kexec -e« startet den geparkten Kernel. Der zum Beispiel mit dem Kommando »reboot« eingeleitete Neustart hat bis zum Aufruf des Programms seine Aufräumarbeiten abgeschlossen.

Nach Abschluss dieser Modifikationen steht dem Reboot nichts mehr im Weg. Hierzu ist als Erstes der neue Kernel inklusive der Bootparameter und einer eventuell notwendigen RAM-Disk zu parken. Auch bei dieser Arbeit hilft das Programm »kexec« mit dem Schalter »-l«. Der Parameter »--append« legt die Bootparameter fest, die Option »--initrd« die initiale RAM-Disk.

Wer sein System über den Bootloader »grub« startet, findet die Konfiguration bei den meisten Distributionen in der Datei »/boot/grub/menu.lst«. Dort ste-

Crash-Dumps

Ein im Speicher geparkter Kernel beschleunigt nicht nur einen normalen Reboot. Der Nutzen für Kernelentwickler besteht darin, bei einer Kernelpanic mit Hilfe des intakten geparkten Kernels noch einen Speicherabzug schreiben zu lassen. Die hierfür bereits existierenden Lösungen hielt Linus Torvalds bis dato für unausgereift, keine von ihnen hat es in den Standardkernel geschafft.

Der Ansatz mit Kexec und Kdump ändert die Situation. Bei einem frischen Kernel ist die Gefahr korrumpierter Datenstrukturen gering. Die Sache hat nur einen Haken: Der geparkte Kernel benötigt selbst den Speicher, den er eigentlichen retten (dumpen) oder zumindest analysieren soll. Doch die Kernelhacker fanden eine Lösung. Zunächst sorgen sie dafür, dass der geparkte Crashkernel selbst sehr schlank ist und wenig Speicher braucht. Der Speicherbereich, auf den der Crashkernel zugreifen darf, ist sogar von vornherein in seiner Größe limitiert. Der eigentliche Kernel reserviert einen ausreichend großen Speicherbereich, in dem der Notfallkernel seine Arbeit verrichten kann.

Nur das Booten erfordert einen überlappenden Speicherbereich. Somit muss Kexec im Fall einer Panic diesen Speicherbereich zuerst retten. Danach startet es den Crashkernel, der jetzt Zugriff auf den nicht gelöschten Speicher hat. Der Admin bekommt eine Shell, mit der er entweder den Speicher direkt analysieren oder einen Speicherabzug übers Netz oder auf Festplatte durchführen kann.

Um Kdump zu nutzen, ist also neben dem Produktiv-Kernel noch der schlanke Crashkernel notwendig. Im Userspace muss außerdem die Kexec-Applikation gepatcht werden. Eine genaue Anleitung dazu findet sich in der Dokumentation [2], eine allgemeine Beschreibung des Vorgangs unter [3].

hen üblicherweise Zeilen wie etwa die folgenden für den Kernel und die initiale RAM-Disk:

```
kernel /boot/vmlinuz-2.6.13.2 ➥
root=/dev/sda2 ro
initrd /boot/initrd.img-2.6.13.2
```

Mit dem Kexec-Kommando wird daraus die Zeile:

```
kexec -l /boot/vmlinuz-2.6.13.2 ➥
--append="root=/dev/sda2 ro" ➥
--initrd=/boot/initrd.img-2.6.13.2
```

Wer den Reboot-Prozess im Detail verfolgen will, sollte auf eine Textkonsole wechseln. Nur auf diese Weise lassen sich die Ausgaben beim Bootprozess verfolgen. Das ist gerade beim ersten Reboot-Versuch wichtig, denn einzelne Treiber verlassen sich auf eine Hardware-Initialisierung durch das Bios.
So ließ sich beispielsweise ein Centrino-Notebook nicht hochfahren, solange der Kernel beim Start den Treiber »ipw2100.ko« benutzte. In solchen Fällen hilft nur noch ein harter Reset. Das System fährt dann wie gewohnt hoch. *(ofr)* ■

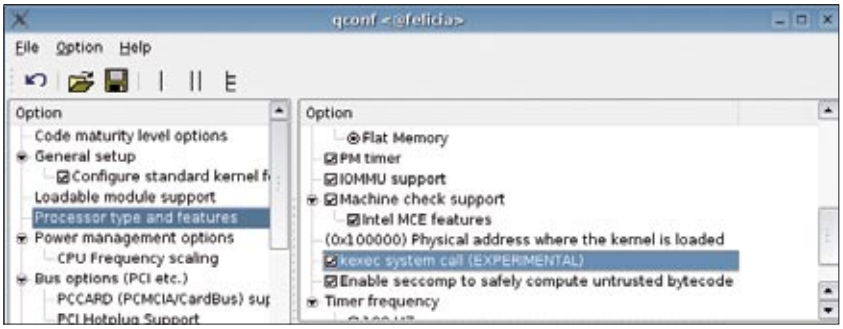


Abbildung 3: Der Kernel muss das Kexec-Feature unterstützen. Die Einstellung findet sich in der Kernelkonfiguration unter »Processor type and features«. Die Ladeadresse des Kernels liegt bei 0x100000.

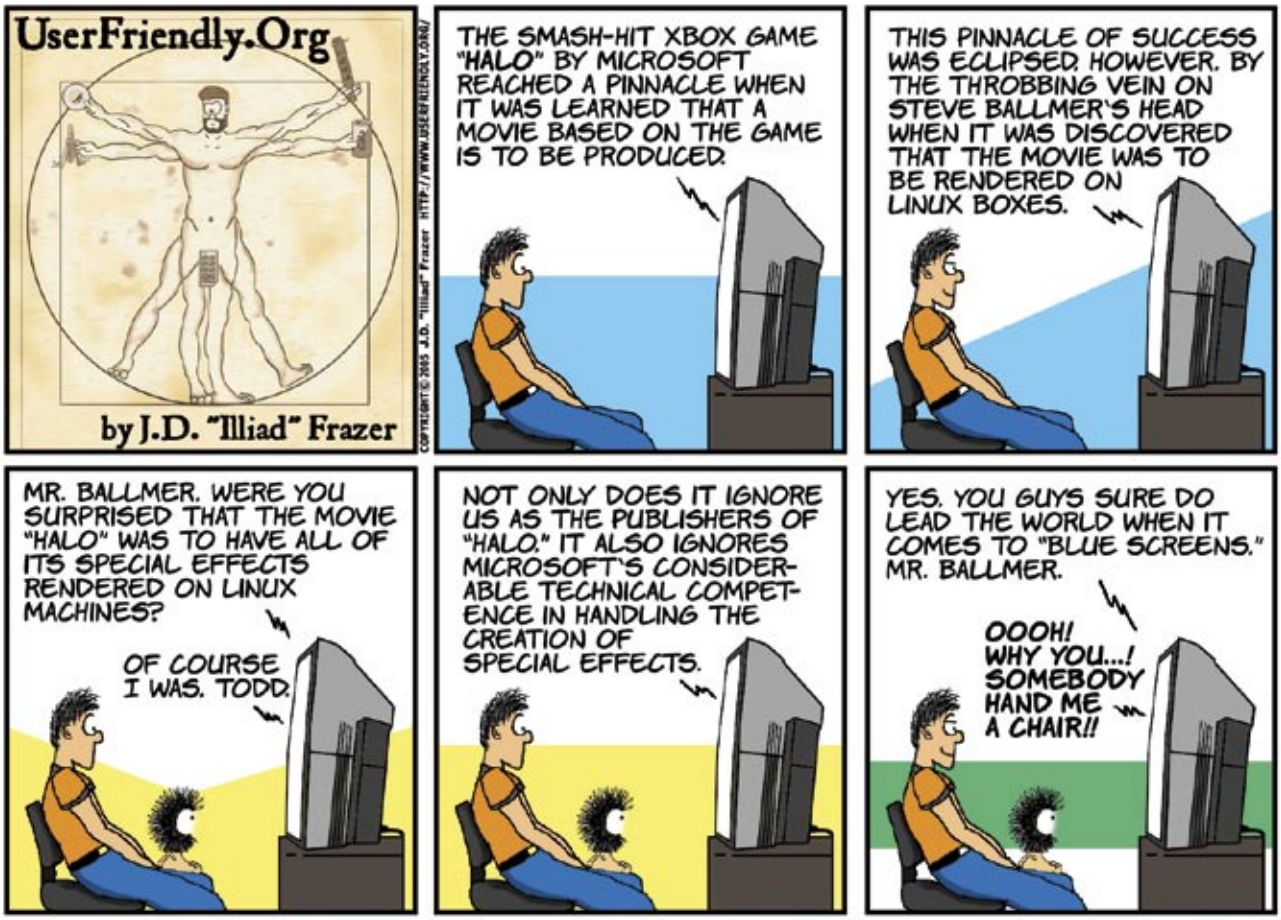
Infos

[1] Hariprasad Nellitheertha, „Reboot Linux faster using kexec“: [http://www-128.ibm.com/developerworks/linux/library/l-kexec.html?ca=dgr-lnxw07RebootFast]
[2] Dokumentation zu Kdump in den Kernelquellen: »/usr/src/linux-2.6.14/Documentation/kdump/kdump.txt«
[3] Jonathan Corbet, „Crash dumps with kexec“: [http://lwn.net/ Articles/108595/]
[4] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur“: Linux Magazin 2/04, S. 28
[5] Kexec: [http://www.xmission.com/~ebiederm/files/kexec]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Sie haben zum Kernel 2.6 das Buch „Linux Treiber entwickeln“ veröffentlicht.

User Friendly – der monatliche Comic im Linux-Magazin



Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 26

Kern-Technik

Um Ressourcenkonflikte parallel laufender Prozesse nicht durch komplizierte Locking-Strategien lösen zu müssen, versuchen es Kernelprogrammierer besser, sie gleich zu vermeiden. Die Kern-Technik zeigt, wie Per-CPU-Variablen ihnen dabei helfen. Eva-Katharina Kunst, Jürgen Quade



Der Schutz kritischer Abschnitte ist eine der höchsten Hürden bei der Programmierung parallel ablaufender Prozesse, vor allem bei echter Parallelität auf Mehrprozessormaschinen (SMP). Entsprechend umfangreich ist das Arsenal der Entwickler: atomare Variablen, Semaphore, Spinlocks und Memory-Barrier, alle vorgestellt in Folge 5 der Kern-Technik [1].

Einfacher ist es allerdings, kritische Abschnitte gleich zu vermeiden. Hierzu bedarf es der Änderung des Datenmodells: Statt den konkurrierenden Zugriff auf Daten durch Semaphor oder Spinlock zu sequenzialisieren, erhält jede parallel zugreifende Instanz (jeder Codeabschnitt) einen eigenen Datenbereich. Dies macht den gleichzeitigen Zugriff möglich und vermeidet den Konflikt.

Sieht das ursprüngliche Datenmodell beispielsweise vor, empfangene Bytes über eine Workqueue in einen einzelnen Buffer abzulegen, vervielfacht – gemäß neuem Datenmodell – der Kernelprogrammierer diesen Buffer entsprechend der Anzahl der parallel zugreifenden Instanzen. Jede dieser Instanzen arbeitet dann auf einem eigenen Datenbereich, der teure Zugriffsschutz entfällt auf Kosten eines etwas erhöhten Speicherbedarfs.

Oder: Wenn es beispielsweise darum geht, statistische Informationen über eine Zugriffshäufigkeit zu protokollieren, speichert im Sinne der Konfliktvermeidung jede Instanz die Anzahl der Zugriffe in einer eigenen Variablen. Erst wenn eine Anwendung sie für die Auswertung anfordert, werden die Daten zusammengeführt. So macht es zum Beispiel das Netzwerk-Subsystem.

Felder bevorzugt

Im Kernel greifen zu einem Zeitpunkt maximal so viele Instanzen zu, wie Prozessoren im System vorhanden sind. Jede CPU benötigt die Variablen damit genau ein Mal. Sie heißen deshalb auch Per-CPU-Variablen. Die Realisierung ist einfach. Anstatt eine Variable genau einmal zu definieren (»static int var«), legt der Programmierer die Variable als Array an (siehe Abbildung 1), und zwar mit so vielen Elementen wie Prozessoren im System vorhanden sind: »static int var[MAX_CPU]«.

Die maximale Anzahl der Prozessoren ist ein Konfigurationsparameter bei der Kernelgenerierung und somit bekannt. Für den Zugriff auf die Variable verwendet der Entwickler das Variablenfeld mit der ID der jeweiligen CPU als Index: »sum = var[smp_processor_id()]«. Für pfiffige Entwickler ist das ein alter Hut. Neu und wenig bekannt ist allerdings, dass Kernel 2.6 für die Verwendung von Per-CPU-Variablen eine komfortable Infrastruktur zur Verfügung stellt. Diese sorgt unter anderem dafür, dass der Kernel die Variablen so im Speicher ablegt, dass sie möglichst in eine Cacheline des zugehörigen Prozessors passen. Das verhindert häufige Cache-Misses und beschleunigt damit den Zugriff.

CPU-eigene Variablen

Außerdem bietet dieses Interface Funktionen, die die Kernel-Preemption ausschalten. Kernel 2.6 hat nämlich auch die Unterbrechbarkeit von Kernelfunktionen eingeführt. Findet aber während des Zugriffs auf einen Per-CPU-Speicherbereich eine Unterbrechung und möglicherweise damit verbunden sogar eine Prozessmigration statt (der Kernel entscheidet sich den aktiven Rechenprozess auf einer anderen CPU ablaufen zu lassen), liegt doch eine zu schützende Race Condition vor (siehe hierzu Kasten „Kernel-Preemption“). Durch das Ausschalten der Kernel-Preemption lässt sich dieser kritische Abschnitt aber mit wenig Aufwand entschärfen.

Tabelle 1 listet die Funktionen für Per-CPU-Variablen auf. Der Kernelprogrammierer legt eine Per-CPU-Variable mit Hilfe des Makros »DEFINE_PER_CPU(*Datentyp*, *Variablenname*)« an. Das Makro »DECLARE_PER_CPU(*Datentyp*, *Name*)« realisiert die Deklaration für den Zugriff eines anderen Moduls auf die Variable. Allerdings muss sie dazu vom definie-

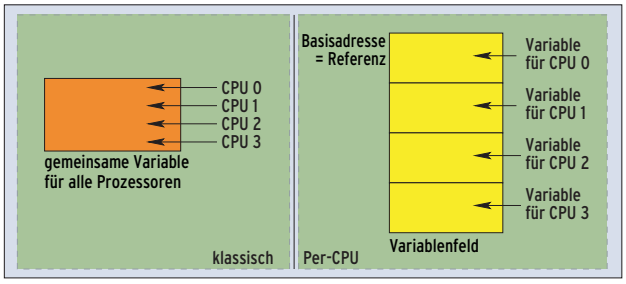


Abbildung 1: Konfliktvermeidung: Für jede CPU wird ein eigener Speicher reserviert.

renden Quellcode per »EXPORT_PER_CPU_SYMBOL_GPL()« beziehungsweise »EXPORT_PER_CPU_SYMBOL()« exportiert werden.

Das Makro »get_cpu_var(name)« übernimmt den eigentlichen Zugriff auf die Variable. Seine Handhabung ist etwas gewöhnungsbedürftig. Es schaltet die Kernel-Preemption aus und löst dann den Variablennamen als so genannten Lvalue auf. Will der Programmierer die Per-CPU-Variable zum Beispiel inkrementieren, darf das »++« direkt hinter dem Makro stehen: »get_cpu_var(stat)++«. Wenn es sich um eine Datenstruktur handelt, steht unmittelbar nach der schließenden Klammer die Elementauswahl, also zum Beispiel »get_cpu_var(statstruct).open«.

Um die Kernel-Preemption wieder zu aktivieren, ruft man »put_cpu_var(*Name*)« auf. Zwischen »get_cpu_var()« und »put_cpu_var()« können beliebig viele Zugriffe auf die Variable erfolgen, allerdings ist es bei Mehrfachzugriffen – wie Listing 2 zeigt – besser, über Pointer zuzugreifen. Ist die Kernel-Preemption ausgeschaltet, ist das Schlafenlegen trotz Prozesskontext tabu und wird vom Kernel mit einer Absturz-Meldung (Oops-Message) quittiert.

Nur mit GPL

Die bisher beschriebenen Funktionen stehen leider nicht allen Kernelkomponenten zur Verfügung. Dass jene Komponente, die Per-CPU-Funktionen benutzen will, GPL-konform sein muss, ist noch leicht zu bewerkstelligen. Schwierigkeiten bereitet eher, dass Kernelmodule die Funktionen aus technischen Gründen nicht benutzen können. Der Linker legt die Variablen nämlich in einem speziellen Segment an.

Innerhalb eines Moduls weichen die Entwickler daher auf die dynamische Allokation über »void *alloc_percpu(size_t *Datentyp*)« und »void free_percpu(const

Listing 1: »kthreadrt.c«

```
01 #include <linux/module.h>
02 #include <asm/delay.h>
03 #include <asm/atomic.h>
04
05 static int thread_id_nort=0, thread_id_rt=0;
06 static DECLARE_COMPLETION( on_exit );
07 static atomic_t active = ATOMIC_INIT(0);
08 static int interrupted = 0, not_interrupted
   = 0;
09 static int disable=0;
10 module_param( disable, int, 0666 );
11
12 static int thread_nort( void *data )
13 {
14    unsigned long remaining;
15
16    daemonize("normal-kthread");
17    allow_signal( SIGTERM );
18    while( 1 ) {
19       if( disable ) preempt_disable();
20       atomic_set( &active, 1 );
21       udelay( 1000 );
22       set_task_state( current, TASK_
   INTERRUPTIBLE );
23       atomic_set( &active, 0 );
24       if( disable ) preempt_enable();
25       remaining = schedule_timeout( 10 );
26       if( signal_pending(current) ) {
27          thread_id_nort = 0;
28          break;
29       }
30    }
31    complete_and_exit( &on_exit, 0 );
32 }
33
34 static int thread_rt( void *data )
35 {
36    unsigned long remaining;
37    struct sched_param sched_params;
38
39    daemonize("rt-kthread");
40    allow_signal( SIGTERM );
41
42    sched_params.sched_priority = 50;
43    if( sched_setscheduler(current,SCHED_
   RR,&sched_params)!=0 ) {
44       printk("can't set realtime priority
   ...\n");
45       complete_and_exit( &on_exit, -1 );
46    }
47
48    while( 1 ) {
49       set_task_state( current, TASK_
   INTERRUPTIBLE );
50       remaining = schedule_timeout( 7 );
51       if( atomic_add_return(0,&active) )
52          interrupted++;
53       else
54          not_interrupted++;
55       if( signal_pending(current) ) {
56          thread_id_rt = 0;
57          break;
58       }
59    }
60    printk("    interrupted: %d\nnot
   interrupted: %d\n",
61       interrupted, not_interrupted );
62    complete_and_exit( &on_exit, 0 );
63 }
64
65 static int __init kthread_init(void)
66 {
67 #ifndef CONFIG_PREEMPT
68    printk("KERNEL DOES NOT SUPPORT
   KERNEL-PREEMPTION!\n");
69 #endif
70    thread_id_nort=kernel_thread(thread_
   nort,NULL,CLONE_KERNEL);
71    if( thread_id_nort==0 )
72          return -EIO;
73    thread_id_rt=kernel_thread(thread_
   rt,NULL,CLONE_KERNEL);
74    if( thread_id_rt==0 ) {
75          kill_proc( thread_id_nort,
   SIGTERM, 1 );
76          wait_for_completion( &on_exit );
77          return -EIO;
78    }
79    if( disable )
80          printk("Kernel-Preemption in
   kernel-thread disabled...\n");
81    else
82          printk("Kernel-Preemption in
   kernel-thread activated...\n");
83    return 0;
84 }
85
86 static void __exit kthread_exit(void)
87 {
88    kill_proc( thread_id_nort, SIGTERM, 1 );
89    kill_proc( thread_id_rt, SIGTERM, 1 );
90    wait_for_completion( &on_exit );
91    wait_for_completion( &on_exit );
92 }
93
94 module_init( kthread_init );
95 module_exit( kthread_exit );
96 MODULE_LICENSE("GPL");
```

void *Referenz)« aus. Die Funktion »alloc_percpu()« nutzt »kmalloc()« und gibt als Referenz die Basisadresse des Variablenfelds zurück. Meist ruft man »alloc_percpu()« während der Modul-Initialisierung auf.

Manuell absichern

Mit der Referenz lässt sich die Adresse des zum gerade aktiven Prozessors gehörenden Datenbereichs bestimmen. Dazu dient die Funktion »get_cpu_ptr(*Referenz*, cpuid«, die die Adresse des zur CPU-Identifikation »cpuid« gehörenden Datenbereichs liefert. Die Parameterliste der Funktion deutet bereits an, dass ein Prozessor durch Angabe der »cpuid« auf jede beliebige Per-CPU-Variable zugreifen kann. Dabei ist aber Vorsicht geboten: Der Zugriff findet konkurrierend zur anderen CPU statt. Es handelt sich also wieder um einen kritischen Abschnitt, der gesichert werden muss.

Üblich ist es daher, die »cpuid« des gerade aktiven Prozessors mit »get_cpu()« zu bestimmen, das zuvor die Kernel-Preemption ausschaltet. Nach erfolgtem Zugriff schaltet »put_cpu()« sie wieder ein. Wird die Per-CPU-Variable nicht mehr benötigt, gibt »free_percpu()« ihren Speicher frei. Listing 3 zeigt, wie sich im Modulcode mehrere Per-CPU-Variablen verwenden lassen.

Nur wenige Kernelprogrammierer benutzen bislang in Modulen den Zugriff auf Per-CPU-Speicher – abgesehen von einigen Anwendungen im Netzwerk-Code. Dagegen finden sich im statischen Teil

Kernel-Preemption

Bis Kernel 2.6 gehorcht Linux einem überschaubaren Unterbrechungsmodell. Es sieht im Kernel drei Ebenen vor: Die eigentliche Kernelebene, die Bottom-Halves-Ebene (BH) und die Hardware-IRQ-Ebene. Dabei können Funktionen der niederen Ebene die der oberen unterbrechen. Sobald der Kernel eine Funktion der niedrigeren Ebene beendet hat, kann er die unterbrochene Funktion weiterbearbeiten.

Der Grundsatz lautet also: Funktionen im Kernel werden zwar unterbrochen, jedoch immer abgeschlossen, bevor eine Funktion auf der gleichen Ebene an die Reihe kommt. Dieses Unterbrechungsmodell reduziert auf Einprozessormaschinen die Zahl kritischer Abschnitte gewaltig. Greifen beispielsweise zwei Kernelfunktionen, die im Prozesskontext ablaufen, auf dieselbe Datenstruktur zu, ist überhaupt kein Locking notwendig.

Ohne Unterbrechung langsam

Diese eigentlich klare Struktur hat aber auch einen Nachteil: Tritt ein Ereignis ein, das eine Aktion mit hoher Priorität erforderlich macht, verzögert sich die Bearbeitung so lange, bis der gerade ablaufende Dienst mit niedriger Priorität abgearbeitet ist. Die Folge: Die Latenzzeiten (Startzeiten für die zugehörigen Aktionen) werden länger.

Da der im Artikel beschriebene kritische Abschnitt für Mehrprozessormaschinen ohnehin durch Semaphore oder Spinlocks abgesichert sein muss und damit die Vorteile des übersichtlichen Unterbrechungsmodells zunichte macht, hat Linus Torvalds mit Kernel 2.6 die Kernel-Preemption zugelassen. Kommt es jetzt im Kernel zu einer Unterbrechung (Interrupt), die eine hoch-priore Recheninstanz aktiviert, bearbeitet er diese sofort. Die ursprünglich aktive Kernelfunktion muss warten. Das macht sich für den Anwender in kürzeren Latenzzeiten positiv bemerkbar, verlangt dem Programmierer aber mehr Sorgfalt ab. Denn damit gibt es auch auf Einprozessormaschinen eine Reihe potenziell neuer kritischer Abschnitte, die der Entwickler sichern muss.

Die Unterbrechbarkeit im Kernel lässt sich übrigens auf einer Einprozessormaschine sehr einfach demonstrieren (siehe Listing 3). Dazu sind nur zwei Kernel-Threads (siehe [2]) nötig, von denen einer mit Echtzeitpriorität, der andere ohne läuft. Beide werden periodisch aktiv. Während der Thread ohne Echtzeitpriorität seine Aktivität über ein Flag signalisiert und ansonsten nur Rechenzeit verbraucht, überprüft der Echtzeit-Thread, ob er den anderen Thread unterbrochen hat. Das erkennt er anhand des Aktivitäten-Flag. Um die Wahrscheinlichkeit einer Unterbrechung zu erhöhen, lässt das Beispiel beide Threads über unterschiedliche Schlafenszeiten asynchron zueinander arbeiten.

Das Schlafenlegen ist auch wichtig, damit das System durch die CPU-hungrigen Kernel-Threads nicht zum Stillstand kommt. Schließlich bringt der Kernel – mit oder ohne Preemption – immer erst eigene Funktionen oder Threads zum Abschluss (Schlaf oder Exit), bevor User-Prozesse an die Reihe kommen.

Test mit Kernelmodul

Zum Test der Unterbrechbarkeit dient der Code aus Listing 1. Nach dem Laden des generierten Moduls mit »insmod kthreadrt.ko disable=0« entlädt »rmmod kthreadrt« es wieder. Falls der Kernel für Kernel-Preemption kompiliert ist, taucht im Syslog eine Meldung wie in Abbildung 3 auf. Beim Test wurde der niedrig-priore Kernel-Thread 25-mal unterbrochen und 251-mal nicht unterbrochen.

Nun soll das Modul ein zweites Mal geladen werden, jetzt mit dem Parameter »disable=1«. Nach zwei Sekunden soll es wieder entladen werden. Dazu dient die Befehlssequenz:

```
insmod kthreadrt.ko disable=1;
sleep 2; rmmod kthreadrt
```

Der Parameter »disable=1« bewirkt, dass der normal priorisierte Kernel-Thread die Unterbrechbarkeit im Kernel ausschaltet. Eine Zeile im Syslog wie »interrupted: 0« zeigt an, dass keine Unterbrechung stattgefunden hat (siehe Abbildung 1).

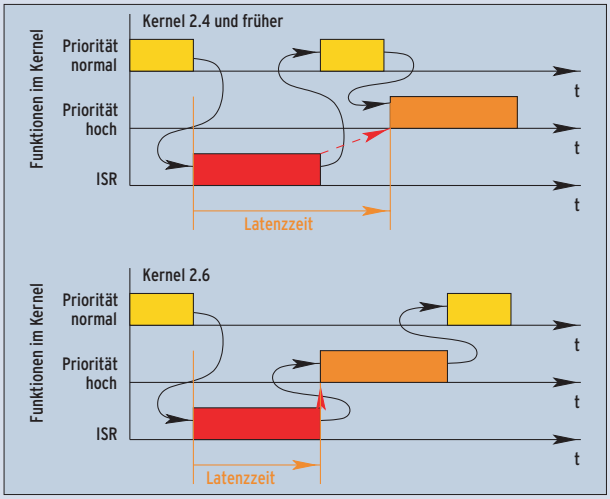


Abbildung 2: Kernel 2.4 beendet erst einen Dienst, bevor er den nächsten beginnt. Kernel 2.6 schiebt schon mal eine hoch-priore Aufgabe dazwischen.

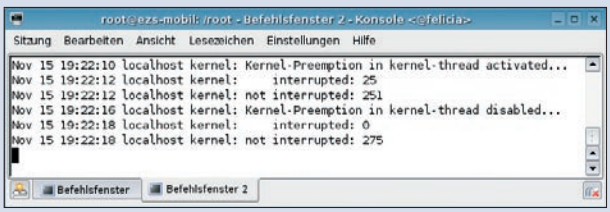


Abbildung 3: Kernel-Preemption eingeschaltet: Der hoch-priore Kernel-Thread unterbricht den normal-prioren.

des Kernels bereits rund 100 Per-CPU-Variablen. Weitere Informationen zu diesen Variablen sind in den beiden Büchern [3] und [4] zu finden. *(ofr)*

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 5: Linux-Magazin 12/03, S. 82

[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 4: Linux-Magazin 11/03, S. 96

[3] Robert Love, „Linux Kernel Development", 2. Auflage: Novell Press 2005

[4] Corbet, Rubini, Kroah-Hartman, „Linux Device Drivers", 3. Auflage: O'Reilly 2005

[5] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2006/02/Kern-Technik/]

Tabelle 1: Funktionen für Per-CPU-Variablen

Name	Kurzbeschreibung
DEFINE_PER_CPU()	Legt eine CPU-Variable an
DECLARE_PER_CPU()	Macht eine CPU-Variable bekannt
get_cpu_var()	Schaltet Kernel-Preemption aus und gibt die Variable zurück
put_cpu_var()	Schaltet Kernel-Preemption ein
per_cpu()	Zugriff auf die Variable ohne Abschaltung der Kernel-Preemption
EXPORT_PER_CPU_SYMBOL()	Exportiert die Variable
EXPORT_PER_CPU_SYMBOL_GPL()	Exportiert die Variable und erlaubt nur den Import GPL-konformer Komponenten
alloc_percpu()	Reserviert im Modul Per-CPU-Speicher
free_percpu()	Gibt reservierten Speicher wieder frei
per_cpu_ptr()	Liefert die Adresse einer per »alloc_percpu()« reservierten Per-CPU-Variablen auf der angegebenen CPU

Listing 2: Kernel-Preemption umschalten

```
01 static DEFINE_PER_CPU( unsigned long, readstat );
02 ...
03    unsigned long *readstatp;
04    ....
05    get_cpu_var(readstat)++;
06       put_cpu_var(readstat);
07    ...
08    readstatp = &get_cpu_var(readstat);
09       *readstatp++;
10       *readstatp++;
11       put_cpu_var(readstat);
```

Listing 3: Per-CPU-Variablen

```
01 void *percpu_ref;
02 int cpuid;
03 struct stat *read_stat; // Für die Referenz
04 ...
05 per_cpu_ref = alloc_percpu( struct stat );
06 ...
07 cpuid = get_cpu();
08 read_stat = per_cpu_ptr( per_cpu_ref, cpuid );
09 read_stat->count += 2; // Verwendung der Variablen
10 put_cpu();
11 ...
12 free_percpu( percpu_ref );
```

Anzeige

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 27

Kern-Technik

Trotz Taktveränderungen durch Speedstep oder Quiet'n'Cool im Kernel exakte Zeiten messen? Diese Folge der Kern-Technik stellt Kernelfunktionen für den richtigen Umgang mit der Zeit vor. Eva-Katharina Kunst, Jürgen Quade



Wie bei gesunden Menschen schlägt auch im Inneren jedes Betriebssystems das Herz regelmäßig den Takt. Dazu löst eine Timer-Hardware periodisch einen Interrupt aus, der unter anderem den Scheduler aktiviert. Fällt dieser Interrupt aus, bleibt das Betriebssystem schließlich ganz stehen. Ein hoher Systemtakt hat Vorteile: Je höher er ausfällt, desto responsiver ist das System.

Andererseits bedeutet ein höherer Systemtakt auch mehr Overhead – was auf schnellen Prozessoren allerdings prozentual gesehen vernachlässigbar ist. Im Kernel 2.4 pochte das Herz noch mit gemächlichen 100 Hz; bei Kernel 2.6 erhöhte Torvalds den Takt auf 1000 pro Sekunde. Je eine Unterbrechung pro Millisekunde führte auf einigen Systemen jedoch zu Problemen. Jetzt läuft der Kernel daher standardmäßig mit 250 Hz, also 4 Millisekunden Abstand.

Linux zählt seinen Takt in der globalen Variablen »jiffies«, die regelmäßige Leser der Kern-Technik-Serie schon kennen gelernt haben. Ein Lesezugriff gibt bereits einen ersten Anhaltspunkt über die Zeitrechnung des Kernels. Doch ist Vorsicht bei der Auswertung geboten: Beim Vergleichen von Zeitwerten kommt es leicht zu Fehlern. Da die Variable Jiffies vom Typ Unsigned Long ist, können auf einem 32-Bit-System maximal 4 Millionen Takte gezählt werden.

Bei einem Systemtakt von 1000 Hz kommt es also etwa alle 49 Tage zu einem Überlauf. In den Vergleich von Zeitstempeln vor und nach dem Überlauf mit den bekannten Operatoren > und < schleichen sich daher sehr leicht Fehler ein.

Makros für Zeitstempel

Falls die Zeitstempel nur ein paar Tage auseinander liegen, funktionieren die in Tabelle 1 aufgeführten Makros »time_after()«, »time_after_eq()«, »time_before()« und »time_before_eq()«. Der Trick: Statt zweier Zeitstempel vom Typ Unsigned Long vergleichen die Makros vorzeichenbehaftete Werte vom Typ Long. Abbildung 1 zeigt, dass damit auch bei einem Zählerüberlauf zwischen den Zeitstempeln das richtige Ergebnis herauskommt. Torvalds zwingt übrigens die Kernelprogrammierer zu sauberer Arbeit: Die Variable Jiffies ist bei Systemstart so vorbelegt, dass sie bereits nach 5 Minuten überläuft, siehe »INITIAL_JIFFIES« in »include/linux/jiffies.h«.

Natürlich lassen sich damit auch Zeitstempel miteinander vergleichen, die weiter auseinander liegen, allerdings nur über die Variable »jiffies64«. Auf einem 32-Bit-System ist der Zugriff auf eine 64-Bit-Variable ein kritischer Abschnitt, den ein Sequence-Lock schützt. Aus diesem Grund sollte der Zugriff nur über die Funktion »get_jiffies_64()« erfolgen. Auf 64-Bit-Hardware bilden Präprozessor und Compiler die Funktion auf den direkten und damit schnelleren Variablenzugriff ab.

Die Genauigkeit, mit der sich Zeiten mit Jiffies bestimmen lassen, liegt je nach gewähltem Systemtakt zwischen 1 und 10 Millisekunden. Zu ungenau für die gestellte Aufgabe? Kein Problem für Linux, denn es aktualisiert in der Kernelvariablen »xtime« die Zeit bei jedem Timertick mit bis zu Nanosekunden-Genauigkeit. Xtime zählt die Sekunden und Nanosekunden, die seit dem 1.1.1970 verstrichen sind. Da die Speicherung der Zeitinformation mehr als 32 Bit erfordert, führt der Zugriff aber zu einem kritischen Abschnitt.

Am besten eignet sich für das Auslesen des Zeitwerts die Funktion »current_kernel_time()«, sie gibt die aktuelle Zeit in einer »struct timespec« zurück:

```
struct timespec {
   time_t tv_sec;
   long tv_nsec;
}
```

Wer stattdessen lieber mit einer »struct timeval« arbeitet, ruft die Funktion »do_gettimeofday()« auf:

```
struct timeval {
   time_t tv_sec;
   suseconds_t tv_usec;
}
```

Die Verwendung der Funktionen demonstriert Listing 1.

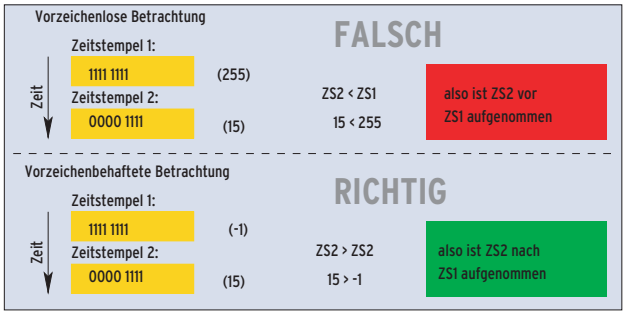


Abbildung 1: Zeitstempel vorzeichenbehaftet zu betrachten ermöglicht den korrekten Zeitvergleich (hier vereinfacht auf 8-Bit-Zeitstempel).

Obwohl »xtime« nur mit jedem Timertick neu gesetzt wird, besitzt die Variable hohe Genauigkeit. Anstatt sie bei 250 Hz Systemtakt blind jedes Mal um 4 Millisekunden zu erhöhen, misst der Kernel die zwischen den beiden Timerticks tatsächlich vergangene Zeit hochgenau. Um diesen Wert erhöht er dann »xtime«. Für die hochgenaue Messung greift der Kernel auf Hardware zurück. Moderne Prozessoren besitzen oft einen Taktzyklenzähler, den Time Stamp Counter (TSC). Mit jedem Taktsignal inkrementiert die CPU diesen Zähler. Bei 1 GHz Systemtakt erreicht der TSC eine Auflösung von Nanosekunden.

Auf diesen Zähler kann der Programmierer natürlich direkt zugreifen, sogar aus dem Userspace (siehe Kasten „Zeit im Userspace"). Beispielcode dafür – lauffähig auf einer x86-Plattform – zeigt Listing 2. Während im Userspace Assembler-Code nötig ist, hält der Kernel die Makros »rdtsc()«, »rdtscl()«, »rdtscll()« und »get_cycles()« bereit.

Da es bei der hohen Zählfrequenz schnell zu Überläufen kommt, ist der TSC auf den meisten Plattformen als 64-Bit-Zähler implementiert. Das Makro »rdtsc« übergibt in zwei Variablen einmal das höher- und einmal das niederwertige 32-Bit-Wort; »rdtscll« belegt die übergebene 64-Bit-Variable direkt mit dem Zählerwert, »rdtscl« organisiert nur das niederwertige 32-Bit-Wort.

Das Makro »get_cycles()« ist übrigens die plattformneutrale Variante, die je nach TSC einen Wert von 32- oder 64-Bit Breite zurückgibt. In Listing 3 ist der Zugriff auf den TSC vom Kernel aus zu erkennen. Doch auch mit dem TSC gibt es Probleme:

- Nicht alle Prozessoren bieten einen Taktzyklenzähler.
- Um aus dem erfassten Zählwert einen Zeitwert zu berechnen, ist die Kenntnis der Taktfrequenz notwendig.
- Auf modernen Prozessoren ist die Taktfrequenz nicht konstant.
- Ein 32-Bit-TSC läuft bei Gigahertz-Takt in wenigen Sekunden über.
- Die Verarbeitung eines 64 Bit breiten Zeitwerts ist auf einem 32-Bit-System nicht in einem Zugriff möglich.

Insbesondere die je nach Last wechselnde Taktfrequenz ist ein großes Problem. Sei es Power Now, Quiet'n'Cool oder Speedstep: Wenn sich während der Zeitmessung die Taktfrequenz ändert, ist eine Differenz-Zeitmessung nicht mit einer einfachen Differenz-Bildung getan.

Time-Shift

Doch auch in dieser Situation bietet Linux einen Ausweg. Schließlich ist es der Kernel selbst, der die CPU anweist einen Gang runter oder auch rauf zu schalten. So kann der Programmierer über den Notifier-Mechanismus vor und nach jeder Taktfrequenzänderung eine eigene Funktion aufrufen lassen. Sie bekommt die zuletzt eingestellte und die neue Taktfrequenz mitgeteilt, sodass die Berechnung der verstrichenen Zeit nicht mehr übermäßig schwer fällt. In Listing

Tabelle 1: Kernelfunktionen zur Zeitmessung

Typ	Kurzbeschreibung
int time_after(unsigned long a, unsigned long b)	Zeitvergleich
int time_after_eq(unsigned long a, unsigned long b)	Zeitvergleich
int time_before(unsigned long a, unsigned long b)	Zeitvergleich
int time_before_eq(unsigned long a, unsigned long b)	Zeitvergleich
u64 get_jiffies_64(void)	Anzahl Timerinterrupts seit Systemstart als 64-Bit-Zähler
struct timespec current_kernel_time(void)	Sekunden und Nanosekunden seit 1. 1. 1970
void do_gettimeofday(struct timeval *tv)	Sekunden und Mikrosekunden seit 1. 1. 1970
rdtsc(unsigned long low, unsigned long high)	Time Stamp Counter
rdtscl(unsigned long low)	Time Stamp Counter (niederwertiges Wort)
rdtscll(unsigned long long val)	Time Stamp Counter (64-Bit-Wort)
cycles_t get_cycles(void)	Time Stamp Counter (Hardware-unabhängig)
void ndelay(unsigned long nanoseconds)	Verzögerung in Nanosekunden
void udelay(unsigned long useconds)	Verzögerung in Mikrosekunden (maximal 20 000)
void mdelay(unsigned long mseconds)	Verzögerung in Millisekunden
long schedule_timeout_interruptible(long timeout)	Unterbrechbar schlafen legen
long schedule_timeout_uninterruptible(long timeout)	Ununterbrechbar schlafen legen
void add_timer(struct timer_list *)	Timer-Tasklet zur späteren Ausführung
int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay)	Funktion in einer Workqueue abarbeiten lassen
schedule_delayed_work(struct work_struct *work, unsigned long delay)	Funktion in der Event-Workqueue abarbeiten lassen

Listing 1: Genaue Zeit im Kernel

```
01 struct timespec spec_since_1970;
02 struct timeval val_since_1970;
03 ...
04 spec_since_1970 = current_kernel_time();
05 do_gettimeofday(&val_since_1970);
```

Listing 2: Time Stamp Counter vom Userspace

```
01 static inline unsigned long long int rdtsc()
02 {
03     unsigned long long int x;
04     __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
05     return x;
06 }
```

Listing 3: Time Stamp Counter im Kernel

```
01 unsigned long low, high;
02 cycles_t tscall;
03 ...
04 if( cpu_has_tsc )
05    rdtsc( low, high );
06 tscall=get_cycles();
```

4 ist die Realisierung eines solchen Taktfrequenz-Notifiers zu sehen.

Als Erstes definiert man ein Objekt, das die Adresse der Notifer-Callback-Funktion aufnimmt (Zeile 31). Der Aufruf von »cpufreq_register_notifier()« macht die Funktion der Zeitverwaltung des Kernels bekannt. Dieser fügt abhängig von der gewählten Parametrisierung die Callback-Funktion in eine von zwei Listen ein: Lautet der zweite Aufrufparameter »CPUFREQ_POLICY_NOTIFIER«, dann ruft der Linux-Kernel die Callback-Funktion jedes Mal auf, wenn sich die Policy der Frequenzumschaltung ändert. In Listing 4 kommt stattdessen »CPUFREQ_TRANSITION_NOTIFIER« zum Einsatz. Damit ruft der Kernel die Notifier-Funktion vor und nach der Taktfrequenzänderung auf.

Zu erwähnen bleibt noch, dass beim Entladen des Moduls das Objekt »nb« wieder aus dem Kernel entfernt werden muss (»cpufreq_unregister_notifier()«). Wer ein System mit variabler Taktfrequenz besitzt, kann den Quellcode aus Listing 4 mit dem Makefile (Listing 5) übersetzen und das erzeugte Modul per »insmod cpunotifier.ko« laden. Bei sehr unterschiedlichen Lastsituationen verändert der Kernel wie beschrieben den Takt. Das Notifier-Modul dokumentiert seine Aktivitäten im Syslog, wie Abbildung 2 zeigt.

Der Linux-Kernel bietet natürlich mehr als nur die Möglichkeit, genaue Zeiten zu erfassen und zu vergleichen. Häufig ist es notwendig, eine gewisse Zeit aktiv oder passiv zu warten. Aktives Warten ist an sich unerwünscht, schließlich kann der Prozessor nützlichere Dinge tun, als Warteschleifen drehen.

Warten in Schleifen

Wenn die Wartezeit kurz ist und sich das Umschalten (Scheduling) auf einen anderen Rechenprozess nicht lohnt oder wenn das Warten im Interruptkontext stattfindet, gibt es kaum Alternativen zur Warteschleife. Der Kernel bietet dafür die Funktionen »ndelay()«, »udelay()« und »mdelay()« an. Die erste Funktion bekommt die Wartezeit in Nano-, die zweite in Mikro- und die dritte in Millisekunden übergeben. Übrigens gibt sich der Compiler redlich Mühe festzustellen, ob »udelay()« mit mehr als 20000 parametrisiert wird. In diesem Fall erscheint beim Kompilieren die Meldung:

```
*** Warning: "__bad_udelay" ?
/tmp/treiber/udelaytest.ko] undefined!
```

Da das Symbol »__bad_udelay« an keiner Stelle im Kernel definiert ist, scheitert das Laden des Moduls.

Noch ein zweiter Punkt: Da »udelay()« (wie auch »ndelay« und »mdelay«) mit einer Zählschleife arbeitet, lässt sich damit prinzipiell sehr genau warten. Unterbrechen aber wichtigere Aktionen die gerade aktive Codesequenz, verlängert sich die Wartezeit um deren Laufzeit.

Zeit im Userspace

Wenn es um Zeit und Zeitverwaltung auf Applikationsebene geht, bietet Linux eine Reihe von Funktionen. Das Auslesen der aktuellen Zeit läuft dabei meist über den Systemcall »gettimeofday()«. Alternativ kann der Programmierer – falls seine CPU das unterstützt – auch auf den Time Stamp Counter zugreifen. Dazu bindet er auf der x86-Architektur einfach die in Listing 2 dargestellte (Assembler-)Funktion in sein Programm ein. Allerdings muss er sicherstellen, dass sich während der Messung die Taktfrequenz nicht ändert.

Das Schlafenlegen eines Rechenprozesses erfolgt typischerweise über den Systemcall »sleep()«. Er bietet aber nur eine Auflösung im Sekundenbereich. Lange Zeit haben sich Programmierer daher mit »select()« beholfen, also mit einer Funktion, der man ein Timeout übergeben kann. Mittlerweile lassen sich Sub-Sekunden-Wartezeiten über »nanosleep()« realisieren. Diese Funktion wird innerhalb des Kernels direkt auf »schedule_timeout_interruptible()« abgebildet.

Um eine Funktion zu einem späteren, definierten Zeitpunkt abarbeiten zu lassen, gibt es zwei Interfaces: Itimer und Posix-Timer. Die über »setitimer()« und »getitimer()« zur Verfügung gestellten Rückwärtszähler schicken dem zugehörigen Prozess ein Signal, wenn der Wert null ist.

Intern ähnlich wie der Itimer implementiert, bieten die Posix-Timer dem Programmierer mit Funktionen wie »timer_create()« und »clock_settime()« ein flexibles Interface. Die Benachrichtigung, dass ein Zähler abgelaufen ist, erfolgt wie bei den Itimern per Signal.

Listing 4: »cpunotifier.c«

```
01 #include <linux/fs.h>
02 #include <linux/notifier.h>
03 #include <linux/cpufreq.h>
04
05 static int my_cpufreq_notifier( struct
   notifier_block *nb, unsigned long val, void
   *data )
06 {
07    struct cpufreq_freqs *freq = data;
08
09    switch( val ) {
10    case CPUFREQ_PRECHANGE:
11       printk("CPUFREQ_PRECHANGE ");
12       break;
13    case CPUFREQ_POSTCHANGE:
14       printk("CPUFREQ_POSTCHANGE ");
15       break;
16    case CPUFREQ_RESUMECHANGE:
17       printk("CPUFREQ_RESUMECHANGE ");
18       break;
19    case CPUFREQ_SUSPENDCHANGE:
20       printk("CPUFREQ_SUSPENDCHANGE ");
21       break;
22    default:
23       printk("unknown val\n");
24       break;
25    }
26    printk("cpu: %d, old: %d, new: %d\n",
   freq->cpu, freq->old, freq->new );
27    return 0;
28 }
29
30 static struct notifier_block nb={
31    .notifier_call = my_cpufreq_notifier,
32 };
33
34 static int __init mod_init(void)
35 {
36    if( cpufreq_register_notifier(&nb,CPUFREQ_
   TRANSITION_NOTIFIER) ) {
37       printk("cpufreq_register_notifer
   failed\n");
38       return -EIO;
39    }
40    printk("mod_init: cpufreq_get: %d\n",
   cpufreq_get(0));// CPU 0
41    return 0;
42 }
43
44 static void __exit mod_exit(void)
45 {
46    cpufreq_unregister_notifier( &nb, CPUFREQ_
   TRANSITION_NOTIFIER );
47 }
48
49 module_init( mod_init );
50 module_exit( mod_exit );
51 MODULE_LICENSE("GPL");
```

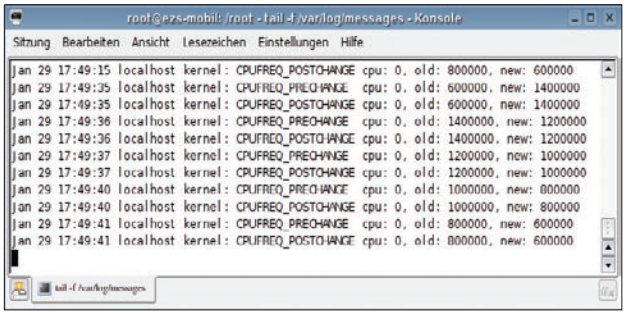


Abbildung 2: Im Syslog zu sehen: Bei jeder Änderung der Prozessortaktfrequenz wird die Notifier-Funktion vom Kernel zweimal aufgerufen, vorher und nachher.

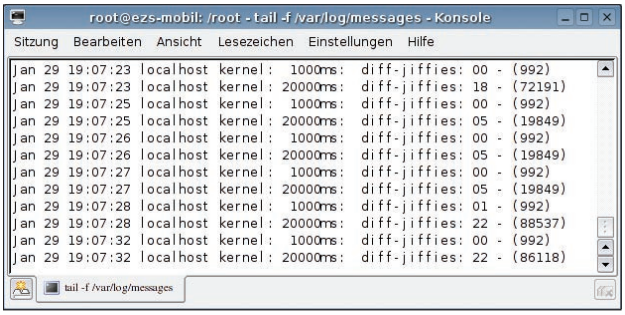


Abbildung 3: Nur bei kurzen Wartezeiten (1000 ms) erreicht »udelay()« eine hohe Genauigkeit. Bei hohen (20 000 ms) wird die Abweichung bereits recht groß.

Hochgenau lässt sich damit also nur in einer Interrupt Service Routine (ISR) warten, wenn andere Aktivitäten unterbunden sind.

Listing 6 verschafft ein Gefühl für die Tücken beim Verwenden von »udelay()«. Beim Laden des Moduls »udelaytest.ko« ruft das Beispiel »udelay()« einmal mit »1000« und einmal mit »20000« auf. Die real vergehende Zeit misst das Programm mit Hilfe des Time Stamp Counter. Das Ergebnis der Messung taucht wieder im Syslog auf (Abbildung 3).

Da die Wahrscheinlichkeit gering ist, bei einer kurzen Wartezeit unterbrochen zu werden, ist die tatsächliche Wartezeit bei kleinen Zeiten (1000 Mikrosekunden) sehr genau. Bei Wartezeiten von mehr als 4 Millisekunden kommt es aber durch den erwähnten Timertick-Interrupt zu ersten Verzögerungen, die sich insgesamt summieren. Im Beispiel wartet die Instanz statt der veranschlagten 20 Milliskunden bis zu 88 Millisekunden (22 Jiffies, Abbildung 3).

Das ohnehin empfehlenswertere passive Warten eliminiert diese zusätzlichen Verzögerungen. Dazu dienen die Funktionen »schedule_timeout_interruptible()« und »schedule_timeout_uninterruptible()«, denen der Programmierer die Wartezeit in Jiffies übergibt. Alternativ rechnen »msecs_to_jiffies()« und »usecs_to_jiffies()« Milli- oder Mikrosekunden-Angaben in Jiffies um. Da die zugehörige Instanz schlafen soll, stehen diese Funktionen nicht im Interrupt-Kontext zur Verfügung. Man kann sie also nicht in einer ISR oder einem Soft-IRQ (Tasklet, Timer) verwenden. Der Aufruf einer der beiden Funktionen garantiert, dass mindestens die angegebene Zeit geschlafen wird.

Timer oder Workqueue

Um nur eine Zeit abzuwarten und dann eine in sich geschlossene Aufgabe zu bearbeiten, ist es besser, mit »add_timer()« direkt auf Timer oder auf die bestehende Event-Workqueue (»queue_delayed_work()«, »schedule_delayed_work()«) zurückzugreifen. Timer-Funktionen laufen im Interrupt-, Workqueues im Prozess-Kontext ab. Timer starten relativ genau zum angegebenen Zeitpunkt; da sich hinter den Workqueues Kernelthreads verstecken, bestimmt der Scheduler deren genauen Abarbeitungszeitpunkt. Zum Aufsetzen eines Timer und die Übergabe einer Funktion an eine Workqueue siehe [1] oder [2].

Noch ein Hinweis: Die Kernelentwickler Thomas Gleixner und Ingo Molnar haben Linus Torvalds davon überzeugt, eine überarbeitete Version der Zeitverwaltung in Kernel 2.6.16 zu übernehmen. Damit halten auch neue High-Resolution Timer in den Kernel Einzug. Alles Wissenswerte dazu verrät die nächste Folge der Kern-Technik. *(ofr)* ■

Infos:

[1] Kunst, Quade, „Kern-Technik“, Folge 4: Linux-Magazin 11/03, S. 96
[2] Quade, Kunst, „Linux-Treiber entwickeln“: Dpunkt-Verlag, 1. Auflage, Juni 2004

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 5: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m       := cpunotifier.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09         $(MAKE) -C $(KDIR)  M=$(PWD) modules
10 endif
```

Listing 6: »udelaytest.c«

```
01 #include <linux/fs.h>
02 #include <linux/delay.h>
03 #include <linux/cpufreq.h>
04
05 static int __init delaytest_init(void)
06 {
07    long old_jiffies;
08    cycles_t t1, t2;
09    int freq=cpufreq_get(0)/1000; // in msec
10
11    t1=get_cycles();
12    old_jiffies = jiffies;
13    udelay(1000); // Kurzes Warten
14    t2=get_cycles();
15    printk(" 1000ms:  diff-jiffies: %2.2ld -
   (%ld)\n",
16           jiffies-old_jiffies, (unsigned
   long)(t2-t1)/freq);
17    old_jiffies = jiffies;
18    t1 = t2;
19    udelay(20000); // Langes Warten
20    t2=get_cycles();
21    printk("20000ms:  diff-jiffies: %2.2ld -
   (%ld)\n",
22           jiffies-old_jiffies, (unsigned
   long)(t2-t1)/freq);
23    return -EIO; // Nur ein Test: Das erspart
   uns das Entladen.
24 }
25
26 module_init( delaytest_init );
27 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 28

Kern-Technik

Mit jeder 64-Bit-Maschine, die über die Ladentheke geht, stehen die Kernelprogrammierer auf dem Prüfstand: Ist ihr eigener Code 64-Bit-tauglich oder nicht? Auch bei der Portierung auf exotische Prozessoren gibt es einiges zu beachten. Eva-Katharina Kunst, Jürgen Quade



Bis vor kurzem war die Welt noch einfach: Alle gängigen PCs waren mit einem Prozessor der Pentium-Klasse bestückt, deren Registerbreite stets 32 Bit beträgt. Zwar unterstützt der Linux-Kernel schon seit Jahren auch andere Architekturen. Aber Hand aufs Herz: Auf der Mehrheit der Maschinen prangt ein Intel- oder ein AMD-Logo.

64 statt 32 Bit

Die Sache mit dem Logo hat sich seitdem nicht grundlegend geändert, doch bei der Registerbreite hat AMD mit der ursprünglich AMD64 genannten Technik einen Sprung getan. Intel folgte mit der kompatiblen Technik EMT64 nach, sodass neue Rechner heute problemlos mit 64 Bit rechnen.

Beim Datentyp »long« kommt der wesentliche Unterschied zwischen 32- und 64-Bit-Technik zum Tragen: Er umfasst auf einem 32-Bit-System tatsächlich 32 Bit, auf dem 64-Bit-System 64. Anders beim Datentyp »int«: Dass er auf einem 64-Bit-Rechner ebenfalls 64 Bit umfasst, ist ein weit verbreiteter Irrglaube. Auf beinahe allen Plattformen ist »int« 32 Bit breit.

Spürbar wird dieser Unterschied dann, wenn ein Programmierer sich unbedacht »unsigned int« als Datentyp für einen so genannte neutralen Parameter wählt, dessen wirklicher Typ sich erst beim Aufruf entscheidet. Die meisten Kernelprogrammierer setzen dafür entweder den Typ »unsigned long« oder »void *« ein. Der entscheidende Punkt ist, dass der Datentyp »unsigned int« mit seinen 32 Bit auf einem 64-Bit-System nicht der Breite des Adressbusses entspricht. Während auf einer 32-Bit-Maschine keine Probleme auftreten (siehe Abbildung 1), ist es fatal, unter einem 64-Bit-Linux die Warnung des Compilers zu übersehen (Abbildung 2).

Listing 1 zeigt beispielhaft, wie ein derartiger, nicht portierbarer Code aussehen kann. Auf einem 64-Bit-System gehen beim Casting einer 8 Byte belegenden Adresse (»&var«) auf einen »int« (Zeile 14) die oberen 4 Bytes verloren. Die Abbildungen 1 und 2 zeigen die zugehörigen Ausgaben auf einer 32- und auf einer 64-Bit-Maschine. Während bei einer Portierung von 32 Bit auf 64 Bit das geschilderte Problem virulent ist, kann es im umgekehrten Fall zu Bereichsüberschreitungen kommen. Ein Beispiel ist der Jiffies-Zähler vom Typ »unsigned long«: Während die knapp 200 Tage bis zum Wrap-Around auf dem 32-Bit-System innerhalb der gesetzlichen Gewährleistungspflicht liegen, brauchen Besitzer von 64-Bit-Maschinen sich erst nach gut 2000 Milliarden Jahren Sorgen zu machen.

Ein anderer Problemfall sind Signale: Auf der 32-Bit-Maschine braucht man zur Verarbeitung von 64 Signalen zwei Wörter, auf einem 64-Bit-System dagegen nur eines. Man sollte sich als Programmierer also lieber zweimal überlegen, ob der Wertebereich der definierten Variablen auch auf Maschinen unterschiedlicher Wortbreite ausreicht. Der geschickte Einsatz des Makros »BITS_PER_LONG« hilft im Übrigen – wie bei den Signalen – dabei, in der Länge begrenzte Datenfelder portabel zu beherrschen.

Portable Datentypen

Wer eine Speicherzelle mit definierter Wortbreite braucht, kann auf entsprechende Typedefs zurückgreifen. Die in der Headerdatei »asm/types.h« definierten Datentypen »u8«, »u16«, »u32«, »u64« sowie »s8«, »s16«, »s32« und »s64« stellen entsprechende vorzeichenfreie und vorzeichenbehaftete Speicherzellen zur Verfügung. Tauscht der Kernel solche Daten mit einer Applikation aus, zum Beispiel im Rahmen eines IO-Con-

Tabelle 1: Datentypen für bitweise definierte Wörter

Format	Datentypnamen
Host-Format	__u8, __u16, __u32, __u64
Little-Endian-Format	__le16, __le32, __le64
Big-Endian-Format (Netzformat)	__be16, __be32, __be64

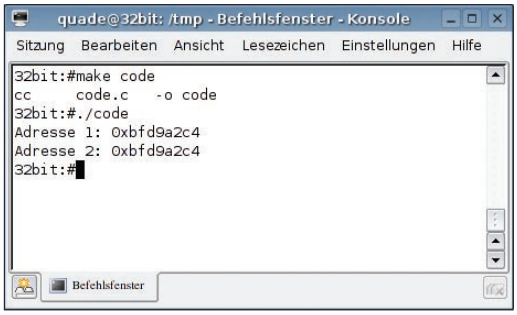


Abbildung 1: Ausgabe des Programms von Listing 1 auf einer 32-Bit-Maschine. Beide Adressen sind wie erwartet identisch.

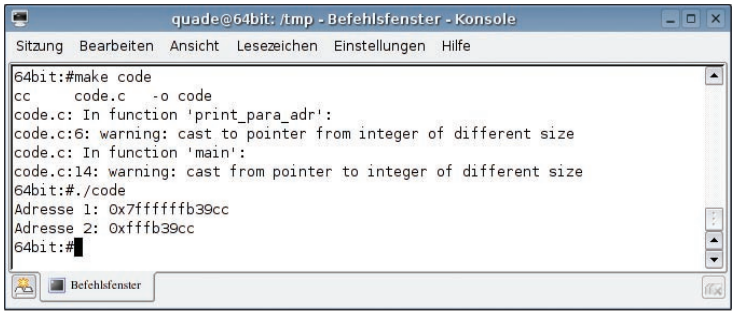


Abbildung 2: Ausgabe des Programms von Listing 1 auf einer 64-Bit-Maschine. Der Compiler warnt bereits bei der Übersetzung vor Problemen, die beim Ablauf auch prompt auftreten.

trol, sollte man auch die im Userspace bekannten Versionen mit den zwei Unterstrichen verwenden: »__u8«, »__u16«, »__u32«, »__u64« und »__s8«, »__s16«, »__s32« und »__s64« (siehe Tabelle 1).

Endianness

Doch selbst Prozessoren mit gleicher Wortbreite unterscheiden sich noch so grundlegend, dass es zu Portierungsproblemen kommt. Unterschiedliche Prozessoren legen Daten nämlich auch anders im Speicher ab. Denn unglücklicherweise gibt es gleich zwei Formate, um 16-, 32- oder 64-Bit-Wörter im Speicher zu repräsentieren.

Abbildung 3 zeigt, dass das Little-Endian-Format das niederwertige Byte eines Wortes auf die niederwertige Adresse legt. So macht es etwa der klassische PC mit x86-Prozessoren. Eine Sun-Workstation mit Sparc-Prozessor legt dagegen das niederwertige Byte an die höherwertige Adresse. Es gibt sogar Prozessoren, die beides können.

Der Endian-Unterschied wird dann zum Problem, wenn zwei oder mehr Prozessoren, die unterschiedliche Datenablageformate verwenden, auf denselben Datensatz zugreifen. Bei vernetzten Rechnersystemen ist das ohnehin oft der Fall. Das Problem kann aber auch beim Zugriff auf intelligente Hardware auftreten, deren Anbindung zumeist über ein Speicherinterface (Dual-Port-RAM, Shared Memory) realisiert ist. Gleiches gilt beim Speichern von Daten auf einer Festplatte – spätestens seit USB 2.0 und Firewire sind diese portabel und leicht von einem zum anderen Rechner zu transportieren.

Byte-Swapping sparen

Zur Lösung des Problems bietet Linux vom Ablageformat des Prozessors unabhängige Datentypen an: »__le16«,» __le32«, »__le64« und »__be16«, »__be32«, »__be64« (siehe Tabelle 1). Außerdem kann der Programmierer auf Makros zurückgreifen, die zwischen den Formaten konvertieren (siehe Tabelle 2).

Damit bei dieser Umwandlung nicht bereits vorher feststehen muss, ob der verwendete Prozessor ein Little- oder ein Big-Endian-Format unterstützt, wandeln sie immer in das CPU- oder Host-Format respektive aus dem CPU- oder dem Host-Format. Dabei ist natürlich berücksichtigt, ob die CPU selbst das Zielformat unterstützt. Dann wird beim Kompilieren der Wert ohne Byte-Swapping übernommen (siehe »__cpu_to_le32(x)« in Listing 2 , Zeile 2).

Die Makros, die im Netzwerkbereich zum Einsatz kommen, heißen: »htons()« (host to network short), »htonl()« (host to network long), »ntohs()« (network to host short) und »ntohl()« (network to host long). Ein Makro »htonc()« (host to char) gibt es nicht, denn das Problem tritt nur bei Variablen auf, die 16, 32 oder 64 Bit umfassen. Der Name Htonl ist üb-

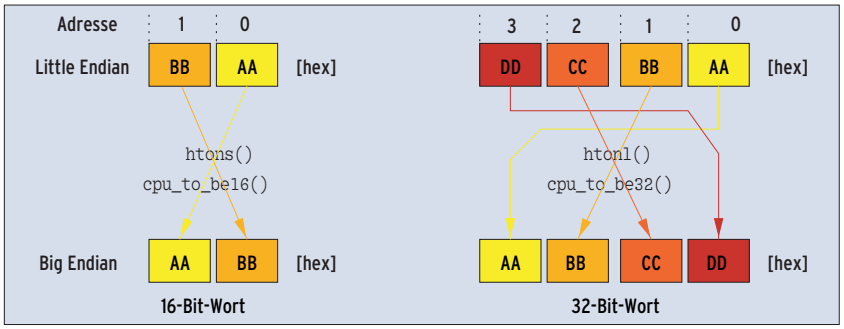


Abbildung 3: Prozessoren legen die einzelnen Bytes eines Datenwortes unterschiedlich im Speicher ab. Beim Little-Endian-Format steht das niederwertige Byte an der niederwertigen (»0«), bei Big Endian dagegen an der höherwertigen Adresse.

Listing 1: Typecast mit Datenverlust

```
01 #include <stdio.h>
02
03 void print_para_adr( unsigned int neutraler_parameter )
04 {
05    printf("Adresse 2: %p\n", (char *)neutraler_
   parameter);
06 }
07
08 int main( int argc, char **argv )
09 {
10    int var;
11
12    printf("Adresse 1: %p\n", &var );
13    print_para_adr( (int)&var );
14    return 0;
15 }
```

Listing 2: »linux/byteorder/little_endian.h«

```
01 ...
02 #define __cpu_to_le32(x) ((__u32)(x))
03 #define __le32_to_cpu(x) ((__u32)(x))
04 #define __cpu_to_le16(x) ((__u16)(x))
05 #define __le16_to_cpu(x) ((__u16)(x))
06 #define __cpu_to_be64(x) __swab64((x))
07 #define __be64_to_cpu(x) __swab64((x))
08 #define __cpu_to_be32(x) __swab32((x))
09 #define __be32_to_cpu(x) __swab32((x))
10 ...
```

Tabelle 2: Funktionen zur Konvertierung zwischen Wort-Ablageformaten

Erklärung	Funktionsnamen
Umwandlung aus dem Host- ins Netzwerk-Format (Big Endian)	__be16 htons(__u16 value), __be32 htonl(__u32 value)
Umwandlung aus dem Netzwerk- (Big Endian) ins Host-Format	__u16 ntohs(__be16 value), __132 ntohl(__be32 value);
Umwandlung aus dem Host- ins Little-Endian-Format	__le16 cpu_to_le16(__u16 value), __le32 cpu_to_le32(__u32 value), __le64 cpu_to_le64(__u64 value)
Umwandlung aus dem Host- ins Little-Endian-Format, das Ergebnis der Wandlung wird in der ursprünglichen Variablen abgelegt (in situ)	void cpu_to_le16s(__u16 *value), void cpu_to_le32s(__u32 *value), void cpu_to_le64s(__u64 *value)
Umwandlung aus dem Host- ins Big-Endian-Format	__be16 cpu_to_be16(__u16 value), __be32 cpu_to_be32(__u32 value), __be64 cpu_to_be64(__u64 value)
Umwandlung aus dem Host- ins Big-Endian-Format, das Ergebnis der Wandlung wird in der ursprünglichen Variablen abgelegt (in situ)	void cpu_to_be16s(__u16 *value), void cpu_to_be32s(__u32 *value), void cpu_to_be64s(__u64 *value)
Umwandlung einer Little-Endian-Variablen ins Host-Format	__u16 le_to_cpu16(__le16 value), __u32 le_to_cpu32(__le32 value), __u64 le_to_cpu64(__le64 value)
Umwandlung einer Little-Endian-Variablen ins Host-Format, das Ergebnis wird in der Variablen selbst abgelegt (in situ)	void le_to_cpu16s(__le16 *value), void le_to_cpu32s(__le32 *value), void le_to_cpu64s(__le64 *value)
Umwandlung einer Big-Endian-Variablen ins Host-Format	__u16 be_to_cpu16(__be16 value), __u32 be_to_cpu32(__be32 value), __u64 be_to_cpu64(__be64 value)
Umwandlung einer Big-Endian-Variablen ins Host-Format, das Ergebnis wird in der Variablen selbst abgelegt (in situ)	void be_to_cpu16s(__be16 *value), void be_to_cpu32s(__be32 *value), void be_to_cpu64s(__be64 *value)

rigens irreführend: Diese Funktion wandelt das Host-Format in ein 32-Bit-Wort (Integer). Wäre es wirklich ein »long«, käme es zu Problemen mit allen 64-Bit-Maschinen. Das Netzwerkformat ist traditionell vom Typ Big Endian. Auf dem x86-PC müssen daher alle Wörter in den Protokollheadern gedreht werden. Eine Sun-Workstation mit Sparc-Prozessor kann darauf verzichten.

Außerhalb der Netzwerkprogrammierung benutzt der Linux-Kernel die Makros »cpu_to_leX()«, »cpu_to_leXs()«, »cpu_to_beX()« und »cpu_to_beXs()« (siehe Tabelle 2), die auch 64 Bit breite Variablen unterstützen. Das angehängte »s« bei einigen der Makros steht für „in situ“. Diese Makros wandeln direkt die Variable selbst. Deshalb besitzen sie keinen Rückgabewert und bekommen nicht eine Kopie, sondern die Adresse der Variablen übergeben.

Das Makro »htons()« entspricht damit der Funktion »cpu_to_be16()«, »htonl()« entspricht »cpu_to_be32()«. Wie man mit Hilfe der Datentypen und Makros programmiert, demonstriert der Kernelcode, der das Ext-2-Filesystem realisiert. Einen Ausschnitt daraus zeigt Listing 3.

Portable Hardware-Kommunikation

Gerade beim Zugriff auf Hardware muss der Programmierer die zwischen dem Hauptprozessor und der Peripherie ausgetauschten Daten mit Hilfe der vorgestellten Datentypen definieren und über die Makros umwandeln. Außerdem darf er die Zugriffe selbst nur über die bereitgestellten Funktionen »read[bwl]()« (read byte, word oder long) respektive »write[bwl]()« (write byte, word oder long) programmieren.

Unterstützt der Prozessor einen eigenen IO-Bereich (Portzugriff), kommen zusätzlich die Funktionen »in[bwl]()« respektive »out[bwl]()« in Frage. Die genaue Verwendung der Funktionen ist in einer früheren Kern-Technik-Folge [1] beschrieben. Dass die Breite des Port-Bereichs auf den Plattformen unterschiedlich ist (x86: 16 Bit, ARM: 32 Bit), bleibt für Programmierer transparent.

Allerdings ist darauf zu achten, ob die Hardware die über einen gemeinsamen Speicher ausgetauschten und in einer Datenstruktur zusammengefassten Variablen an spezifischen Adressen erwartet. Der Compiler legt nämlich einzelne Variablen einer Datenstruktur aus Gründen der Performance an so genannten ausgerichteten (aligned) Adressen ab.

Da ein 32-Bit-Prozessor ein anderes Alignment als ein 16-Bit-Prozessor aufweist, muss der Programmierer sicherstellen, dass die Variablen eine eindeutige Adressenlage besitzen. Das GCC-spezifische Schlüsselwort »__attribute__((packed))« hilft ihm dabei. Abbildung 4 stellt das Problem und die Verwendung des Schlüsselworts beispielhaft dar.

Listing 3: Ausschnitt aus »linux/ext2_fs.h«

```
01 ...
02 struct ext2_super_block {
03         __le32  s_inodes_count;         /* Inodes count */
04         __le32  s_blocks_count;         /* Blocks count */
05         __le32  s_r_blocks_count;       /* Reserved blocks count */
06         __le32  s_free_blocks_count;    /* Free blocks count */
07         __le32  s_free_inodes_count;    /* Free inodes count */
08         __le32  s_first_data_block;     /* First Data Block */
09         __le32  s_log_block_size;       /* Block size */
10         __le32  s_log_frag_size;        /* Fragment size */
11         __le32  s_blocks_per_group;     /* # Blocks per group */
12         __le32  s_frags_per_group;      /* # Fragments per group */
13         __le32  s_inodes_per_group;     /* # Inodes per group */
14         __le32  s_mtime;                /* Mount time */
15         __le32  s_wtime;                /* Write time */
16         __le16  s_mnt_count;            /* Mount count */
17         __le16  s_state;                /* File system state */
18 ...
19 #define EXT2_HAS_COMPAT_FEATURE(sb,mask)                        \
20         ( EXT2_SB(sb)->s_es->s_feature_compat & cpu_to_le32(mask) )
21 #define EXT2_HAS_RO_COMPAT_FEATURE(sb,mask)                     \
22         ( EXT2_SB(sb)->s_es->s_feature_ro_compat & cpu_to_le32(mask) )
23 ...
```

Tabelle 3: Portierungsmerkmale ausgewählter Linux-Plattformen

Name	Bezeichnung	Wortbreite	Größe (»long«)	Seitengröße
x86	i386	32 Bit	4	4 KByte
AMD64/EMT64	x86-64	64 Bit	8	4 KByte
Alpha	alpha	64 Bit	8	8 KByte
Arm/Strong ARM	arm26	32 Bit	4	32, 16 KByte
HP PA-Risc	parisc	32 oder 64 Bit	4/8	4 KByte
Etrax 100LX	cris	32 Bit	4	8 KByte
IA-64	ia64	64 Bit	8	4, 8, 16, 64 KByte
Mips	mips	32 oder 64 Bit	4/8	4, 8, 16, 64 KByte
Motorola 68xxx	m68k	32 Bit	4	4, 8 KByte
PowerPC	ppc	32 Bit	4	4 KByte
PowerPC64	ppc64	64 Bit	8	4 KByte
S390	s390	32 oder 64 Bit	4/8	4 KByte
Sun Sparc	sparc	32 Bit	4	4, 8 KByte
Ultra Sparc	sparc64	64 Bit	8	8, 64, 512, 4096 KByte
User Mode Linux	um	32 oder 64 Bit	4/8	4 KByte

Natürlich gibt es noch weitere Fallen. Doch der Linux-Kernel bemüht sich darum, dem Programmierer Portierungsfehler so schwer wie möglich zu machen. So sind im Kernel einige Datentypen über Typedefs abstrakt gehalten (»ssize_t«, »size_t«, »pid_t«, …). Der Typ »ssize_t« beispielsweise ist, abhängig von der Plattform, entweder auf den Datentyp »int« oder aber auf »long« abgebildet. Deshalb sollte man im eigenen Code grundsätzlich die vordefinierten Datentypen verwenden.

Seitengröße

Auch die Seitengröße (Pagesize) führt schließlich noch zu Portabilitätsproblemen. Tabelle 3 zeigt, dass sie zwischen den verschiedenen Plattformen stark variiert. Einige Kernelfunktionen – zum Beispiel Funktionen des Gerätemodells (Sys-Filesystem) – übergeben jeweils eine Speicherseite. Das Einzige, worauf sich der Programmierer hier verlassen kann, ist, dass sie eine Minimalgröße von 4 KByte aufweist.

Zum Abschluss die drei wichtigsten Regeln für plattformunabhängigen Code zusammengefasst: Auf die Datentypen achten, an der richtigen Stelle konvertieren und auf Hardware über Kernelfunktionen zugreifen. *(ofr)*

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 3: Linux-Magazin 10/03, S. 81

[2] Greg Kroah-Hartman: Writing Portable Device Drivers, [http://www.linuxdevices.com/articles/AT5340618290.html]

[3] Robert Love, „ Linux Kernel Development", S. 321: Novell Press, 2. Auflage, 2005

[4] Corbet et.al., „Linux Device Drivers", S. 288: O'Reilly, 3. Auflage, 2005

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

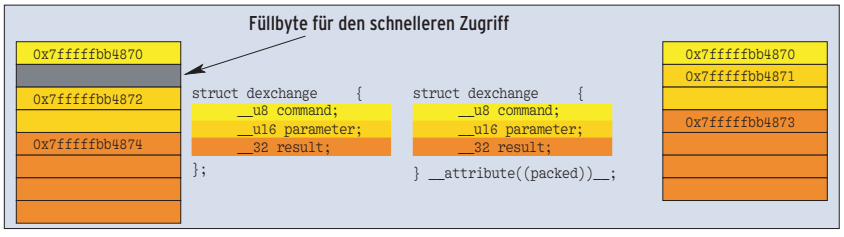


Abbildung 4: Mit Hilfe des Schlüsselworts »packed« kann der Programmierer Einfluss auf die Anordnung von Strukturelementen im Speicher nehmen.

Workshop: Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 29

Kern-Technik

Das Relay-Subsystem ermöglicht den Datenaustausch großer Datenmengen zwischen Kernel- und Userspace in Hochgeschwindigkeit. Diese Kern-Technik-Folge erklärt dies Schritt für Schritt. Eva-Katharina Kunst, Jürgen Quade



Debug- oder Log-Informationen werden innerhalb des Kernels klassischerweise mit Hilfe von »printk()« in den Userspace übertragen und dort vom Syslog weiterverarbeitet. Neben dieser einfachen Variante hat sich in der Vergangenheit das Proc- und seit neuestem das Debug-Filesystem etabliert. Ob per »printk()« oder mit der Ausgabe über ein virtuelles Dateisystem: Der Transfer großer Datenmengen zwischen Kernel und Applikationen ist aufwändig und zudem programmiertechnisch kompliziert. Daten werden nämlich zuerst in Puffern (die überlaufen können) zwischengespeichert, bevor sie im Speicherbereich einer Applikation landen.

Um diese Probleme zu lösen, entwickelten einige Kernelhacker mit dem Relay-FS ein neues, virtuelles Dateisystem, das einerseits einfach zu handhaben ist und andererseits dank »mmap()« einer Anwendung den direkten Zugriff auf die Daten ohne weiteren Kopiervorgang ermöglicht. Mit dem Kernel 2.6.14 nahm Linus Torvalds dieses Dateisystem in den Kernel auf.

Suchen Sie allerdings in einem aktuellen Kernel nach dem Relay-Filesystem, wird die Suche vergeblich sein: Getreu dem Motto „Keep it simple“ haben die Entwickler den Code kräftig abgespeckt, einige Betriebsmodi wie Lockless und den Part des virtuellen Dateisystems komplett entfernt und nur das Kernfeature übrig gelassen: schnelle Transfers großer Datenmengen zwischen Kernel- und Userspace. Statt eines speziellen Filesystems sollen die Nutzer jetzt die vorhandenen virtuellen Dateisysteme nutzen, insbesondere das Debug-Filesystem. An einer einfachen Anbindung des Sys-FS arbeiten die Entwickler noch.

Debug-Filesystem hilft

Wollen Sie also von einem Treiber oder einem anderen Kernelmodul aus einer Anwendung Daten zur Verfügung stellen, bietet sich eine Relay-Datei im Debug-Filesystem an. Das Verfahren ist denkbar einfach (siehe Abbildung 1): Als Erstes legen Sie mit »debugfs_create_dir()« im Debug-Filesystem ein Verzeichnis an und erzeugen dann einen so genannten Relay-Kanal (»rchan«, »relay_open()«, siehe Tabelle 1).

Dafür ist es nötig, zwei kurze Callback-Funktionen zu implementieren (siehe Abbildung 2). Den ersten Callback ruft der Kernel auf, wenn das Relay-Subsystem die Relay-Datei anlegen möchte. Erzeugen Sie dazu die Datei mit Hilfe des Debug-Filesystems (siehe Kasten „Debug-Filesystem“). Der zweite Callback

Tabelle 1: Interface des Relay-Subsystems

Funktionsname	Beschreibung
struct rchan *relay_open(const char *base_filename, struct dentry *parent, size_t subbuf_size, size_t n_subbufs, struct rchan_callbacks *cb)	Öffnet einen Relay-Kanal, dessen zugehörige Dateien den Basisnamen »base_filename« erhalten. Die Relay-Dateien werden unterhalb des durch »parent« angegebenen Verzeichnisses erstellt. Der erzeugte Ringpuffer besteht aus »n_subbufs«-Teilpuffern, die jeweils eine Größe von »subbuf_size« haben. Die Adressen der Callback-Funktionen finden sich in »cb«.
void relay_write(struct rchan *chan, const void *data, size_t length)	Schreibt »length« Bytes der an der Adresse »data« stehenden Daten in den durch »chan« angegebenen Relay-Kanal.
void __relay_write(struct rchan *chan, const void *data, size_t length)	Schreibt »length« Bytes der an der Adresse »data« stehenden Daten in den durch »chan« angegebenen Relay-Kanal. Die Funktion darf nicht im Interrupt-Kontext eingesetzt werden.
void relay_close(struct rchan *chan)	Schließt den Relay-Kanal »chan«.
void relay_subbufs_consumed(struct rchan *chan, unsigned int cpu, size_t consumed)	Die Funktion signalisiert dem Relay-Subsystem, dass die Anwendung eine Anzahl »consumed« an Puffern des Relay-Kanals »chan« auf der CPU »cpu« gelesen hat.

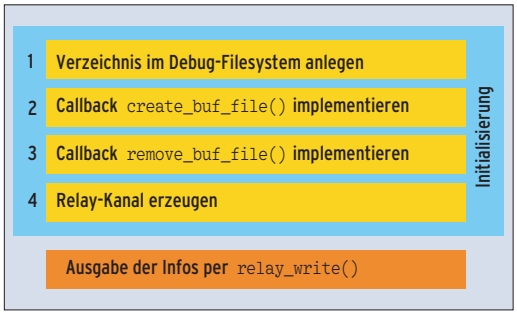


Abbildung 1: In vier Schritten zum eigenen Relay-Kanal. Dann ist er mit der Funktion »relay_write()« verwendbar.

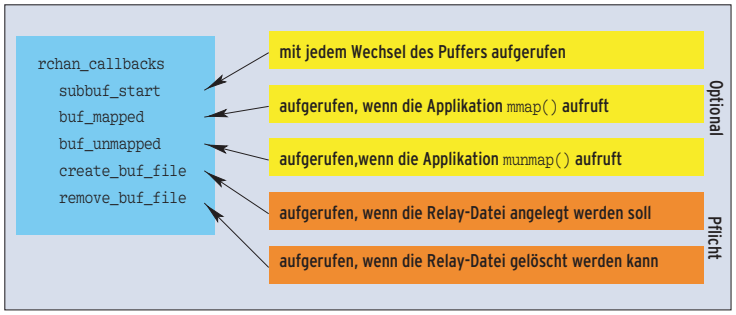


Abbildung 2: Bei den zu implementierenden Callbacks für eine Relay-Datei sind optionale und obligatorische zu unterscheiden.

wird aufgerufen, um die Relay-Datei wieder zu entfernen. Damit löschen Sie die zuvor erzeugte Datei im Debug-Filesystem wieder.

Mehrere CPUs

Die über den so angelegten Relay-Kanal publizierten Informationen bildet der Userspace auf einen Dateizugriff ab, wobei für jede CPU eine eigene Ausgabedatei existiert. Auf einer Einprozessor-Maschine gibt es also eine, auf einer Zweiprozessor-Maschine zwei Relay-Dateien. Hintergrund dieser Maßnahme: Jede CPU schreibt die Daten in eigene Per-CPU-Variablen (Ringpuffer), Locking ist nicht notwendig (siehe [1]). Der für jede CPU einmal vorhandene Ringpuffer unterteilt sich wiederum in Teilpuffer (siehe Abbildung 3).

Größe und Anzahl dieser Puffer werden beim Erzeugen des Kanals festgelegt. Die Gesamtgröße des verwendeten Speichers ergibt sich über die Formel: Gesamtgröße = (Anzahl_Puffer * Puffergröße) * Anzahl_Prozessoren.

Mehrere Kriterien bestimmen die Größe der einzelnen Puffer. Zum einen sollte sie ein Vielfaches von 2 sein und am besten in Beziehung zu einer Page (4096 Bytes) stehen. Das Relay-Subsystem reserviert den notwendigen Speicher nämlich auf Basis von Speicherseiten. Zum anderen sollte die längste, durch einen Funktionsaufruf zu schreibende Nachricht in den Puffer passen. Denn das Relay-Subsystem weigert sich aus Performance-Gründen, Log-Messages auf mehrere Puffer aufzusplitten. Wählen Sie beispielsweise eine Puffergröße von 128 Bytes, können Sie nicht mit einem einzelnen Aufruf 129 Bytes schreiben, die Daten würden verworfen.

Sind die zu schreibenden Häppchen klein genug, werden sie hintereinander in den aktiven Puffer geschrieben, solange noch ausreichend Platz ist, um die Nachricht komplett abzulegen. Sollte der Puffer bereits so weit gefüllt sein, dass die Daten des aktuellen Auftrags nicht mehr hineinpassen, gilt der Puffer als gefüllt, auch wenn einige Bytes ungenutzt bleiben, die man Padding-Bytes nennt. Die Daten des aktuellen Auftrags fließen dann in den nächsten Puffer.

Sobald der letzte Puffer gefüllt ist, legt das Relay-Subsystem die Daten wieder im ersten Puffer ab. Ob es tatsächlich dazu kommt, hängt allerdings davon ab, ob eine Applikation die Daten in der Zwischenzeit gelesen hat. Ist dies nicht der Fall, werden die neuen Daten verworfen. Zur Realisierung zählt das Relay-Subsystem beim Aufruf von »read()« mit, wie viele Daten bereits in den Userspace transferiert wurden.

Zwei Modi zur Auswahl

Einfacher ist es, die Relay-Datei nicht im Non-Override-Modus, sondern im Override-Modus zu betreiben. Diese Betriebsart gestattet es dem Relay-Subsystem, einen Puffer zu überschreiben und damit unter Umständen den Verlust von (alten) Daten in Kauf zu nehmen. Um den Überschreibmodus (Override) zu aktivieren, implementieren Sie einen eigenen Callback »subbuf_start«. Diese Callback-Funktion wird jedes Mal aufgerufen, wenn das Relay-Subsystem einen Puffer neu beschreiben möchte (also beim Wechsel der Puffer). Die Callback-Funktion hat drei Aufgaben:

- Den nächsten Puffer zu initialisieren, falls dies nötig ist.
- Den soeben beschriebenen Puffer zu finalisieren.
- Über den Rückgabewert dem Relay-Subsystem mitzuteilen, ob es den nächsten Puffer beschreiben darf oder nicht. ▶

Listing 1: »debugfs.c«

```
01 ...
02 static struct dentry *vardir, *varfile;
03 static u32 index=99;
04
05 static int __init mod_init(void)
06 {
07     vardir = debugfs_create_dir("vardir", NULL);
08     if( vardir==NULL ) {
09         printk("Kann debugfs-dir nicht
   anlegen\n" );
10         return -EIO;
11     }
12     varfile = debugfs_create_u32("index",
   S_IRUGO|S_IWUGO, vardir, &index);
13     if( varfile==NULL ) {
14         printk("Kann debugfs-file nicht
   erzeugen\n");
15         debugfs_remove( vardir );
16         return -EIO;
17     }
18     printk("Laden: index hat den Wert: %d\n",
   index );
19     return 0;
20 }
21
22 static void __exit mod_exit(void)
22 {
23     printk("Entladen: index hat den Wert: %d\
   n", index );
24     debugfs_remove( varfile );
25     debugfs_remove( vardir );
26 }
27 ...
```

Debug-Filesystem

Mit dem Debug-Filesystem haben die Kernelentwickler eine elegante Möglichkeit geschaffen, einzelne Variablen und ganze Speicherbereiche (Blobs) des Kernels einer Applikation zum Lesen und auch zum Schreiben zur Verfügung zu stellen. Die Variablen selbst sind aus Sicht der Anwendung auf Pseudodateien abgebildet, die der Anwendungsentwickler auch über die Systemcalls »read()« und »write()« lesen und schreiben kann. Beim Lesen wird der Wert der Variablen in eine Ascii-Repräsentation umgewandelt. Entsprechend übergibt die Anwendung den Wert ebenfalls in Ascii.
Die Programmierschnittstelle zu dem Debug-Filesystem ist einfach, Tabelle 2 listet die Prototypen der Funktionen auf. Angenehm ist: Jeweils nur ein Funktionsaufruf legt sowohl ein Verzeichnis im Debug-Filesystem als auch eine Variablendatei an. Der Funktion zum Erstellen der Variablendatei übergeben Sie unter anderem die Adresse der Variablen, die beim Zugriff auf die Datei ausgelesen respektive geschrieben werden soll. Als Datentypen stehen neben den Standardformaten (»u8«: Unsigned Char, »u16«: Unsigned Short, »u32« Unsigned Int) auch »boolean« und »blob« zur Verfügung. Bei Blobs handelt es sich um normale Speicherbereiche zum Austausch binärer Daten.

Pseudodateien selbst gemacht

Zusätzlich erlaubt das Debug-FS das Anlegen eigener Pseudodateien. Dabei implementieren Sie selbst die Lese- und die Schreibfunktion »read()« und »write()«. Diesen Mechanismus nutzt auch das Relay-Subsystem. Das Aufräumen übernimmt »debugfs_remove()«.

Listing 1 zeigt ausschnittweise den Einsatz des Debug-Filesystems an einem einfachen Beispiel, der vollständige Code ist auf der Website des Linux-Magazins zu finden [3]. Zeile 7 erzeugt zunächst ein Verzeichnis, Zeile 12 legt innerhalb des neuen Verzeichnisses die Debug-FS-Datei »index« an. Eine Applikation kann über diese Datei die Variable »index« lesen und schreiben. Der Datenaustausch zwischen User- und Kernelspace findet ebenfalls im Ascii-Format statt.
Und noch zwei Hinweise. Erstens: Bei den über das Debug-Filesystem exportierten Variablen muss es sich um globale Variablen handeln. Zweitens: Auf Blobs lässt sich nur lesend zugreifen. Überhaupt ist der Zugriff auf die Variablen natürlich nur dann möglich, wenn das Debug-Filesystem gemountet ist.

Gibt die Callback-Funktion 0 zurück, bedeutet dies, dass die Message nicht in den nächsten Puffer wandert. Wahrscheinlich hat bisher keine Applikation die Daten in diesem Puffer abgeholt. Der Rückgabewert 1 zeigt an, dass der Puffer beschreibbar ist. Befinden sich dort nicht abgeholte Daten, werden sie überschrieben. Die Callback-Funktion »subbuf_cb()« gibt für diesen Override-Fall den Wert 1 zurück:

```
static int subbuf_cb (struct rchan_buf *buf,
void *subbuf, void *prev_subbuf,size_t prev_
padding) { return 1; }
```

Der Parameter »subbuf« spezifiziert die Speicheradresse des Puffers, in den die nächsten Daten wandern sollen. »prev_subbuf« enthält die Speicheradresse des soeben beschriebenen Puffers, »prev_padding« enthält die Anzahl der im letzten Puffer ungenutzten Bytes.
Zusätzlich gibt es die Callbacks »buf_mapped« und »buf_unmapped«. Der Kernel ruft sie immer dann auf, wenn eine Applikation ein »mmap()« respektive ein »munmap()« durchführt (siehe Abbildung 2).

Daten ausgeben

Die eigentliche Ausgabe der Daten übernimmt innerhalb des Moduls die Funktion »relay_write()«. Sie ähnelt in ihren Parametern dem aus dem Userspace bekannten Systemaufruf »write()«: die Channel-ID als Referenz auf den Ausgabekanal, die Adresse der zu schreibenden Daten und die Anzahl der zu schreibenden Bytes. Die Funktion darf im Kernel sowohl im Interrupt- als auch im Kernelrespektive im Prozess-Kontext ablaufen. Sind Sie sicher, niemals Daten im Interrupt-Kontext zu protokollieren, können Sie auch die Variante »__relay_write()« verwenden. So ersparen Sie dem Kernel eine Lock-Operation.
Listing 2 zeigt den Kernelcode, der eine Relay-Datei anlegt und darüber Log-Informationen ausgibt. Um den Code zu testen, müssen Sie allerdings einen

Listing 2: »log_debugfs.c«

```
01 ...
02 static char *TEXT="hello world\n";
03 static struct rchan *channel_id;
04 static struct dentry *logdir;
05
06 static int lremove_buf_file(struct dentry
   *dentry)
07 {
08     debugfs_remove(dentry);
09     return 0;
10 }
11
12 static struct dentry *lcreate_buf_file(const
   char *filename,
13     struct dentry *parent, int mode, struct
   rchan_buf *buf, int *is_global)
14 {
15     return debugfs_create_file(filename,
   mode, parent, buf,
16         &relay_file_operations);
17 }
18
19 static int lsubbuf_start(struct rchan_buf
   *buf, void *subbuf, void *prev_subbuf,
20     unsigned int prev_padding)
21 {
22     return 1; // bufferchange can occur
23 }
24
25 struct rchan_callbacks cb = {
26     subbuf_start:  lsubbuf_start,
27     create_buf_file: lcreate_buf_file,
28     remove_buf_file: lremove_buf_file,
29 };
30
31 static int __init mod_init(void)
32 {
33     int i;
34
35     logdir = debugfs_create_dir("relay", NULL);
36     if( logdir==NULL ) {
37         return -EIO;
38     }
39
40     channel_id = relay_open( "logfile",
   logdir, 32, 4, &cb );
41     for( i=0; i<20; i++ ) { // Probehalber
   die Puffer fuellen
42         relay_write( channel_id, TEXT,
   strlen(TEXT)+1 );
43     }
44     return 0;
45 }
46
47 static void __exit mod_exit(void)
48 {
49     relay_close( channel_id );
50     debugfs_remove( logdir );
51 }
52 ...
```

Tabelle 2: Kernel-Schnittstelle zum Debug-Filesystem

Prototyp	Beschreibung
struct dentry *debugfs_create_dir(const char *name, struct dentry *parent)	Legt ein Verzeichnis unterhalb des durch »parent« gegebenen Verzeichnisses an. Rückgabe: Referenz auf das neu angelegte Unterverzeichnis.
struct dentry *debugfs_create_u8(const char *name, mode_t mode, struct dentry *parent, u8 *value)	Erzeugt die Pseudodatei »name« unterhalb des durch »parent« angegebenen Verzeichnisses; »value« enthält die Adresse der zugehörigen globalen 8-Bit-Variablen.
struct dentry *debugfs_create_u16(const char *name, mode_t mode, struct dentry *parent, u16 *value)	Legt die Pseudodatei »name« unterhalb des durch »parent« angegebenen Verzeichnisses; nisses an. »value« enthält die Adresse der zugehörigen globalen 16-Bit-Variablen.
struct dentry *debugfs_create_u32(const char *name, mode_t mode, struct dentry *parent, u32 *value)	Erzeugt die Pseudodatei »name« unterhalb des durch »parent« angegebenen Verzeichnisses; »value« enthält die Adresse der zugehörigen globalen 32-Bit-Variablen.
struct dentry *debugfs_create_bool(const char *name, mode_t mode, struct dentry *parent, u32 *value)	Legt die Pseudodatei »name« unterhalb des durch »parent« angegebenen Verzeichnisses an. »value« enthält die Adresse der zugehörigen globalen 32-Bit-Variablen. Der Wert dieser Variablen wird als Boolean interpretiert, wobei sie die Werte »Y«, »y« oder »1« für True und »N«, »n« oder »0« für False annehmen kann.
struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob)	Legt die Pseudodatei »name« unterhalb des durch »parent« angegebenen Verzeichnisses an. »blob« enthält in einer Datenstruktur Adresse und Umfang des zugehörigen Speicherbereichs. Blobs können nur gelesen, nicht geschrieben werden.
struct dentry *debugfs_create_file(const char *name, mode_t mode, struct dentry *parent, void *data, const struct file_operations *fops)	Legt die Pseudodatei »name« unterhalb des durch »parent« gegebenen Verzeichnisses an. »fops« enthält die Adressen der Zugriffsfunktionen »read« und »write«.
void debugfs_remove(struct dentry *dentry)	Entfernt die durch »dentry« angegebene Pseudodatei oder das Pseudoverzeichnis.

Kernel mindestens in Version 2.6.17. verwenden. Er muss sowohl das Debug-Filesystem als auch das Relay-Subsystem aktiviert haben. Falls nicht, kommen Sie um Kernelkonfiguration und -generierung nicht herum. Hinweise hierzu finden Sie unter [2]. Den Quellcode des Moduls übersetzen Sie am einfachsten mit dem Makefile von [3].

Problemfall »mmap()«

Auf Anwendungsseite lesen Sie den Inhalt der Relay-Datei beispielsweise mit Hilfe von »cat«. Viele Daten schreibt das Testmodul jedoch nicht, einige Hello-World-Strings müssen zur Demonstration genügen.

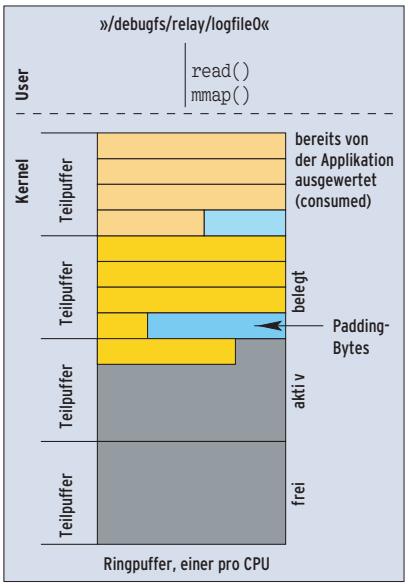


Abbildung 3: Kernel-intern ist die Relay-Datei als Ringpuffer realisiert, der seinerseits aus mehreren Teilpuffern besteht.

Der schnelle Zugriff auf die Daten per »mmap()« ist gar nicht so einfach umzusetzen. Da die Anwendung unbemerkt vom Kernel auf den Datenbereich zugreift, muss er sie auf irgendeinem Weg informieren, wann sie das darf. Umgekehrt muss sie den Kernel – im Non-Override-Modus – davon in Kenntnis setzen, welche Puffer sie bereits konsumiert hat. Beide Mechanismen lassen sich auf Basis des Debug-Filesystems realisieren.

Notwendige Variablen wie die Puffergröße und die Anzahl der Teilpuffer exportiert die Funktion »debugfs_create_uXX()« in den Userspace. Indem der Kernel mit der Funktion »read()« eine mit »debugfs_create_file()« erzeugte Datei liest, teilt er der Anwendung mit, dass sie einen neuen Teilpuffer lesen kann. Damit erfährt die Anwendung außerdem, wie viele gültige Bytes der Puffer enthält. Mit »write()« signalisiert sie dem Kernel, dass sie den Puffer konsumiert hat. Der Kernel ruft daraufhin die Funktion »subbufs_consumed()« aus dem Relay-Filesystem auf.

Kleiner und weniger Bugs

Als das Relay-Subsystem noch das Relay-Filesystem war, hatte es – wie der leider nicht mehr aktuellen Dokumentation [4] und [5] zu entnehmen ist – noch einige Features mehr zu bieten. Doch der Devise „Weniger ist mehr“ folgend reduzierten die Kernelentwickler den neuen Mechanismus auf seine Grundfunktionalität (siehe den Artikel [6]). Erfreulicherweise haben sie dabei gleich noch einige Fehler behoben. Nutzer des neuen Relay-Subsystems gibt es im Übrigen im Kernel bisher überraschend wenige – was sich sicherlich bald ändern wird. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 26: Linux-Magazin 02/06, S. 102
[2] Peter Kreußel, „Kernel übersetzen und installieren“: LinuxUser 6/06, S. 88
[3] Listings online: [http://www.linux-magazin.de/Service/Listings/2006/09/Kern-Technik]
[4] Zanussi et.al., „Relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space“: [http://www.research.ibm.com/K42/papers/ols03.ps]
[5] Relayfs-Dokumentation in den Kernelquellen: »/usr/src/linux-2.6.17/Documentation/filesystems/relayfs.txt«
[6] Statt Relay-Filesystem nur noch Relay-Subsystem: [http://kerneltrap.org/node/4593]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr Buch „Linux-Treiber entwickeln“ ist soeben in zweiter Auflage erschienen.

Workshop: Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 30

Kern-Technik

Hochwertige Zufallszahlen sind schwieriger zu erzeugen, als man denkt. Dieser Artikel verrät, wie Linux virtuell würfelt und damit eine hohe Qualität der generierten Zahlenfolgen sicherstellt. Einige Bereiche lassen aber noch Raum für Verbesserungen. Eva-Katharina Kunst, Jürgen Quade



Nicht nur Computerspiele, in denen Monster sich zufällig bewegen, sondern vor allem Programme mit kryptographischen Funktionen benötigen Zufallszahlen. Während sich Games auch mit Pseudo-Zufallszahlen zufrieden geben, die ein Mathematiker knacken könnte, benötigen Programme und Protokolle wie SSL/TLS, SSH, IPsec oder PGP hochwertige Zufallszahlensequenzen, die weder berechenbar noch leicht zu erraten sind. Der Betriebssystemkern hat es sich zur Aufgabe gemacht, solche echten Zufallszahlen zu generieren.

Zufall aus Rauschen

Ähnlich den Physikern, die ihre Zufallszahlen aus dem Rauschen einer Strahlungsquelle oder elektronischer Bauelemente erzeugen, nutzt Linux ebenfalls ein Rauschen, das so genannte Umgebungsrauschen. Genauer: Linux berücksichtigt den zeitlichen Abstand zwischen Anforderungen unterschiedlicher Peripheriegeräte.

Zum Beispiel misst der Kernel den zeitlichen Abstand zwischen zwei Tastenanschlägen oder zwischen zwei Interrupts, die von der Festplatte ausgelöst werden. Da sich kaum vorhersagen lässt, wann ein Benutzer die nächste Taste auf dem Keyboard anschlägt oder die Festplatte den nächsten Interrupt auslöst, handelt es sich um zufällige Zeitpunkte, die Linux in einen Zufallszahlenpool einspeist. Timer-Interrupts berücksichtigt es dagegen nicht, da sie im ungefähr gleichen zeitlichen Abstand auftreten und damit vorhersagbar sind.

Hilfe durch Anwendungen

Wer einen Gerätetreiber schreibt, kann beim Reservieren des Interrupts durch Setzen des Flag »SA_SAMPLE_RANDOM« dem Kernel anzeigen, dass die Geräte-Interrupts zufällig auftreten und er sie deshalb beim Erzeugen von Zufallszahlen berücksichtigen kann. Zufällig heißt in diesem Zusammenhang, dass der Wert weder vorhersagbar noch der Auftrittszeitpunkt beobachtbar ist. Je mehr gute, unkorrelierte Zufallsquellen zusammenkommen, desto besser. Applikationsprogrammierer können mit Schreibzugriffen auf die Gerätedatei »/dev/urandom« Daten zur Erzeugung guter Zufallszahlen beisteuern.

Der Kernel übernimmt dabei nicht jeden mutmaßlichen Zufallswert blind. Er merkt sich die Differenz zum letzten (»delta«) und vorletzten Wert (»delta2«) und überprüft, ob die Werte ausreichend unterschiedlich sind. Regelmäßige Tastenanschläge tragen kaum zur Generierung von Zufallszahlen bei.

Zwar mischt der Kernel jeden noch so schlechten Wert in seinen Input Pool, stellt aber mit der Entropie eine Maßzahl bereit, die die Zufallsqualität und damit den Zustand des Speichers ausdrückt. Ein 32-Bit-Wert kann maximal 12 Bit Entropie liefern, 1 Byte liefert also 3 Bit Entropie. Das Kommando »cat /proc/sys/kernel/random/entropy_avail« gibt die Entropie des Input Pool aus (siehe Tabelle 1). Der Superuser kann per IO-Control diesen Entropiewert sogar manipulieren (siehe Tabelle 2).

Besser mischen

Da einzelne Zeitdifferenzen zum Teil nicht so zufällig sind wie erwünscht, legt der Kernel sie nicht einfach nacheinander im Speicher ab. Vielmehr verrechnet er jeden neuen Zufallswert über die so genannte Mixing Function mit einigen im Input Pool bereits vorhandenen Daten (siehe auch [1]). Im ungünstigen Fall (der neue Wert ist nicht wirklich zufällig) werden damit bereits vorhandene Zufallswerte nicht noch zufälliger.

Zur Verrechnung dienen Bit-Schiebereien und XOR-Verknüpfungen, die einen kryptologischen Angriff verhindern sollen. Der Input Pool ist übrigens 128 Wörter à 4 Bytes (also 512 Bytes beziehungsweise 4096 Bits) groß. Den schematischen Aufbau des Zufallszahlengenerators zeigt Abbildung 1.

Für den Zugriff auf die Zufallszahlen stellt der Kernel zwei Gerätedateien zur

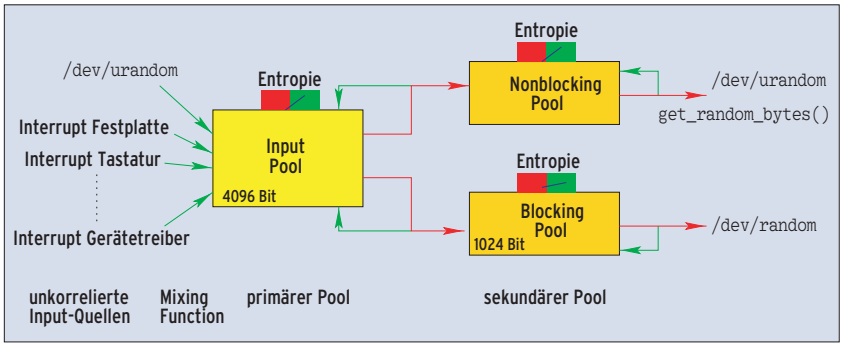


Abbildung 1: Möglichst viele unkorrelierte Input-Quellen liefern Zufallswerte, die eine Mixing Function in den so genannten Input Pool mischt. Liest eine Applikation über »/dev/random« oder »/dev/urandom« eine Zufallszahl, vermischt der Kernel gehashte Daten aus dem Input Pool mit den Daten im sekundären Pool und liefert sie mit einem Hash zurück. Der sekundäre Pool teilt sich wiederum in Nonblocking und Blocking Pool.

Verfügung. Über »/dev/random« und »/dev/urandom« lesen Applikationen Zufallszahlen aus. Damit die Applikationen aus den gelesenen Daten keine Rückschlüsse auf den Inhalt des Input Pool ziehen können, ergreift der Kernel eine Reihe von Maßnahmen:

- Er bildet über jeweils 512 Bit des Input Pool einen SHA-1-Hash (siehe Kasten „SHA-1“) mit 5 Wörtern à 4 Byte (=160 Bit). Mit jeweils einem dieser Wörter modifiziert er den Input Pool. Nach einer weiteren Änderung des Input Pool kopiert er 16 Wörter (512 Bit) an Daten in einen Zwischenspeicher. Der Zugriff auf den Input Pool führt also zu einer Veränderung der Daten im Input Pool. Von den Daten im Zwischenspeicher bildet der Zufallszahlengenerator den SHA-1-Hash, den er an einen so genannten Sekundärspeicher weiterreicht.
- Den SHA-1-Hash vermischt der Kernel mit den Daten des Sekundärspeichers, also des Blocking und des Nonblocking Pool.
- Über jeweils 512 Bits des Secondary Pool bildet der Kernel einen SHA-1-Hash. Mit jeweils einem der fünf Wörter des Hash modifiziert er den Secondary Pool (Abbildung 2, a und b). Nach einer weiteren Modifikation des Pool kopiert er 512 Bit an Daten in einen Zwischenspeicher (Abbildung 2, c). Somit wird mit jedem Lesezugriff auch der Secondary Pool für einen Angreifer nicht nachvollziehbar modifiziert. Von den Daten im Zwischenspeicher errechnet der Kernel wieder einen SHA-1-Hash (Abbildung 2, d). Dessen Bytes verknüpft der Zufallszahlengenerator dann noch Byteweise über XOR mit sich selbst.
- Den Hash kopiert er als Zufallswert in den Speicherbereich der aufrufenden Instanz.
- Variablen, die Zwischenwerte aufgenommen haben, setzt der Kernel explizit auf 0.

Für jedes Byte, das der Kernel vom primären in den sekundären Pool überführt, erniedrigt sich die Entropie des Input Pool um 8 (1 Byte = 8 Bits). Ebenso sinkt die Entropie des sekundären Pools um die Anzahl der Bits, die dort tatsäch-

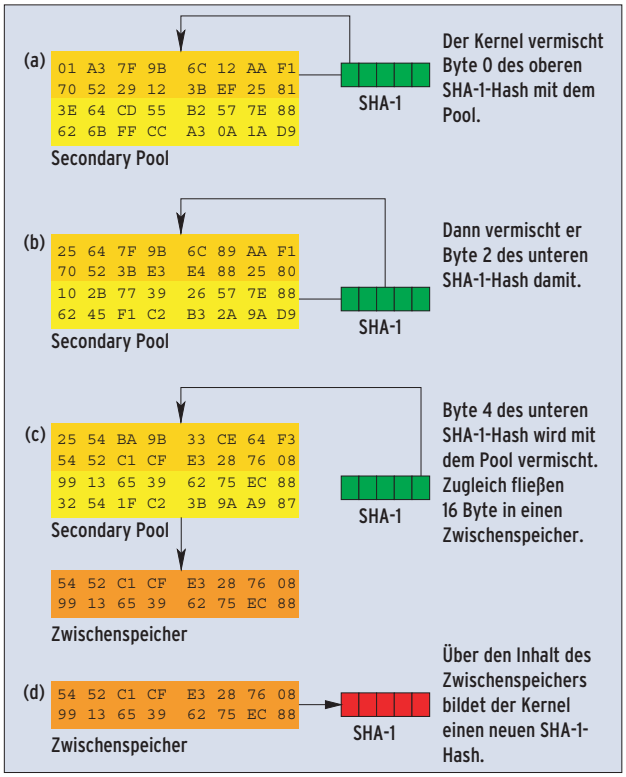


Abbildung 2: Auch der Secondary Pool wird mehrfach durchmischt, wenn eine Applikation oder der Kernel Zufallszahlen auslesen. Kritiker finden, dass diese Methode die Qualität der Zufallszahlen nicht verbessert.

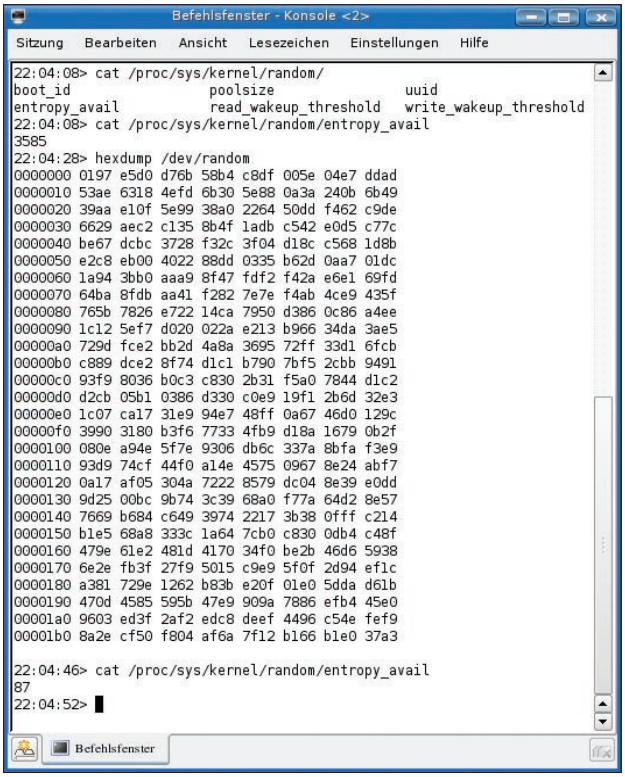


Abbildung 3: Ansatzpunkt für eine Denial-of-Service-Attacke: Bei ungenügender Entropie im Zufallszahlenspeicher blockieren jene Applikationen, die von »/dev/random« lesen. Die Vergabe geeigneter Zugriffsrechte minimiert diese Gefahr.

SHA-1

Der SHA-1-Algorithmus dient zur Berechnung eines eindeutigen Prüfwerts beliebiger elektronischer Daten (siehe [2]). Ein Block von 512 Bits führt zu einem Prüfwert mit einer Länge von 160 Bit. Der Algorithmus weist im Wesentlichen zwei Eigenschaften auf: Erstens ist er kollisionsfrei: Zwei unterschiedliche Datensätze besitzen auch einen unterschiedlichen Hashwert. Zweitens galt er lange als irreversibel: Auf Basis des Hashwerts lassen sich daher keine Rückschlüsse auf den zugrunde liegenden Datensatz ziehen.
Inzwischen konnte theoretisch nachgewiesen werden, dass SHA-1 nicht wirklich kollisionsfrei ist. Ein praktischer Beweis hierzu steht aber noch aus. Für den Einsatz im Kernel ist das nicht zentral. Erst wenn mit vertretbarem Aufwand aus einem Hash auf den zugrunde liegenden Datensatz geschlossen werden kann, wenn der Algorithmus also nicht irreversibel ist, muss ein anderer Algorithmus implementiert werden.

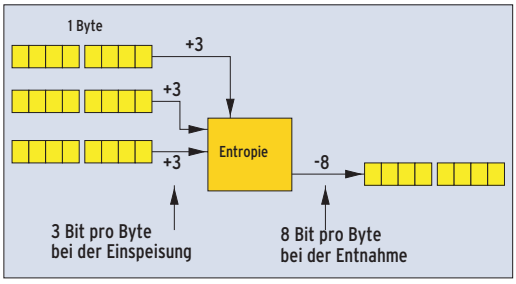


Abbildung 4: Die Entropiebilanz zeigt, dass Linux auf Nummer Sicher geht: Jeder echte Zufallswert mit 8 Bit liefert maximal 3 Bit Entropie. Jedes entnommene Zufallsbyte schlägt jedoch mit 8 Bit zu Buche.

lich entnommen werden. Zur Sicherheit speist Linux für jedes entnommene Zufallsbyte (bei der Entnahme gehen 8 Bit Entropie verloren) 3 echte Zufallsbytes (mit jeweils 3 Bit Entropie) in seinen Input Pool (siehe Abbildung 4).

Mehr Entropie

Abbildung 1 zeigt, dass Linux zwei sekundäre Pools implementiert, den Blocking Pool und den Nonblocking Pool. Der Blocking Pool ist der Gerätedatei »/dev/random« zugeordnet. Wie sein Name bereits vermuten lässt, können diesem Pool nicht mehr Zufallsbits entnommen werden, als Entropie im Speicher vorhanden ist.
Ist die Entropie aufgebraucht, schläft eine zugreifende Applikation, bis wieder genug Lese-Entropie (Proc-Datei »/proc/sys/kernel/random/read_wakeup_threshold«) vorhanden ist. Der Screenshot in Abbildung 3 zeigt dieses Verhalten: Vor dem Aufruf des Kommandos »hexdump /dev/random« ist mit 3585 Bit genügend Entropie im Input Pool vorhanden. Gut 400 Zufallswerte gibt das Kommando direkt nach dem Aufruf aus, danach blockiert sie. Unterbricht der Programmierer nach 18 Sekunden das Hexdump-Kommando mit [Strg] + [C] und liest die Entropie erneut aus, zeigt sich: Sie ist auf 87 gesunken. Zwischen unterschiedlichen Systemen variieren diese Werte natürlich, abhängig davon, in welchem Ausmaß Zufallswerte erzeugt und verbraucht werden.
Der Nonblocking Pool ist der Gerätedatei »/dev/urandom« zugeordnet. Über diesen Pool erzeugt der Kernel per SHA-1-Hash beliebig viele Zufallszahlen, selbst dann, wenn die gesamte Entropie bereits aufgebraucht ist. Auch wenn Kryptographen die auf diese Weise generierten Zufallszahlen nicht unbedingt als stark bezeichnen, reichen sie für die meisten Applikationen aus. Ein Angreifer müsste nämlich aus dem Hash den initialen Zustand des Pools herleiten können, was nach dem heutigem Stand der Forschung nicht möglich ist.
Über den Nonblocking Pool greifen typischerweise auch In-Kernel-User zu. Für diese exportiert der Zufallszahlengenerator die Funktion »get_random_bytes(void *buf, int nbytes)«. Die Funktion extrahiert »nbytes« Zufallzahlen und legt sie an der Adresse »buf« im Speicher ab. TCP nutzt beispielsweise diese Funktion für die initialen Sequenznummern.

Schwelle fürs Schlafen

Mit einem Schreibzugriff auf die Gerätedatei »/dev/urandom« fügen Applikationsprogrammierer Daten dem Input Pool hinzu. Allerdings wird die Entropie dabei nicht aktualisiert. Die Gerätedateien »/dev/random« und »/dev/urandom« unterstützen auch die Funktion »select()«. Eine Applikation kann sich also so lange schlafen legen, bis der Entropiewert des Input Pool unterhalb von »write_wakeup_threshold« (128 Bit) fällt. Dann wird die Applikation aufgeweckt, um den Pool mit ausreichend neuen Zufallsdaten zu füllen.

Tabelle 1: Proc-Dateien

Dateiname	Funktion
boot_id	Einmalig generierte, 128 Bit große Zufallszahl.
entropy_avail	Aktuell im Input Pool vorrätige Entropie.
poolsize	Größe des Input Pool in Bits.
read_wakeup_threshold	Schwellenwert für die Entropie des Input Pool, bei dessen Überschreiten Applikationen aufgeweckt werden, die beim Zugriff auf »/dev/random« schlafen. Diese Datei ist auch beschreibbar.
uuid	Das Lesen dieser Datei erzeugt eine 128 Bit große UUID als Ascii-Datei.
write_wakeup_threshold	Wenn die Entropie im Input Pool unterhalb dieses Grenzwerts fällt, werden Applikationen aufgeweckt, die per »poll« oder »select« darauf warten, Daten in den Pool schreiben zu können. Diese Datei ist auch beschreibbar.

Tabelle 2: IO-Controls

IO-Control	Bedeutung
RNDGETENTCNT	Liest die Entropie des Input Pool aus.
RNDADDTOENTCNT	Modifiziert den Entropiewert des Input Pool. Der Aufruf ist nur dem Superuser gestattet.
RNDADDENTROPY RNDZAPENTCNT	Der IO-Control hat drei Parameter: Der Parameter »ent_count« spezifiziert den Entropiewert, der zur Entropie des Input Pool hinzuaddiert wird. Der Parameter »size« gibt die Anzahl der Bytes an, die dem Input Pool hinzugefügt werden sollen. Der Parameter »adr« ist die Speicheradresse im Userspace, ab der sich die dem Input Pool hinzuzufügenden Werte befinden. Der Aufruf beider IO-Controls ist nur dem Superuser gestattet.
RNDCLEARPOOL	Der IO-Control initialisiert den Input Pool, den Blocking Pool und den Nonblocking Pool. Der Aufruf ist nur dem Superuser gestattet.

Ein Schreibzugriff auf die Gerätedatei »/dev/urandom« initialisiert den Zufallszahlengenerator. Da typischerweise gerade beim Hochfahren des Rechners die Abläufe vorhersagbar sind, könnte ein Angreifer mit etwas Glück den Zustand des Zufallszahlengenerators direkt nach dem Hochfahren prädizieren. Wenn aber der Zahlenspeicher noch vor dem ersten Zugriff mit echten Zufallswerten gefüllt wird, hat ein Angreifer, der diese Werte nicht kennt, keine Chance. Daher speichern moderne Linux-Systeme beim Herunterfahren einige Zufallswerte in einer Datei und kopieren sie beim Hochfahren zurück in den Input Pool.

Direkt nach dem Kopieren überschreibt das System die in der Datei gespeicherten Zufallszahlen mit neuen Werten. Wird der Rechner nämlich nicht ordentlich heruntergefahren, sondern einfach ausgeschaltet, verhindert diese Maßnahme, dass Linux den Input Pool mit immer den gleichen Werten initialisiert. Das zugehörige Skript heißt »/etc/init.d/urandom«.

Keine Panik

Die Kernelentwickler haben eine ganze Reihe von Maßnahmen ergriffen, um echte Zufallszahlen zu generieren. Dennoch ist ihre Implementierung nicht frei von Kritik. Im Mai dieses Jahres veröffentlichten drei Autoren aus Israel eine Analyse der rund 2500 Codezeilen und kritisierten darin vor allem die Qualität des Code: Er sei unübersichtlich und zudem weitgehend undokumentiert, so lautet ihr Fazit.

Doch ihre Kritik zielt auch auf die Implementation selbst. Die wiederholte Anwendung von SHA-1 erhöhe nur die Komplexität, nicht aber die Sicherheit. Der blockierende Zugriff über »/dev/random« erlaube zudem eine Denial-of-Service-Attacke. Denn sobald ein Angreifer dem System ausreichend Entropie (durch eigene Zugriffe auf die Gerätedatei) entzieht, blockiert eine zweite Applikation unter Umständen lebenslang.

Dass die (Lese-)Modifikationen am Input, Blocking und Nonblocking Pool durch Rückführung einzelner SHA-1-Wörter geschehen, bevor die Zufallszahl generiert wird, ermöglicht einen Vorwärtsangriff. Ist nämlich der Zustand des Pools bekannt, lässt sich damit die daraus generierte Zufallszahl ableiten. Würden erst die Daten dem Pool entnommen und danach die (Lese-)Modifikation stattfinden, wäre es nicht möglich, aus dem Poolzustand die letzte Zufallszahl abzuleiten.

Die Angriffsgefahr ist zwar nachvollziehbar, für die Kernelhacker selbst aber nur theoretisch. Der nicht blockierende Zugriff über »/dev/urandom« ist vor einem Denial-of-Service-Angriff gefeit und liefert ausreichend echte Zufallszahlen. Und wenn der Zustand des Input Pool bekannt ist, ist es ohnehin zu spät. Der Hacker ist schon im System.

Mehr Zufall durch Hardware

Übrigens: Wem die Qualität des Linux-Zufallszahlengenerators wider Erwarten nicht ausreicht, dem sei die Bastelanleitung aus **[4]** empfohlen. Alles was er dazu braucht, sind eine Webcam, ein Rauchmelder und ein wenig Geschick im Umgang mit Strahlungsquellen. Viel Spaß beim Bauen. *(ofr)*

Infos

[1] Eastlake, Crocker, Schiller, „Randomness Recommendations for Security", RFC 1750

[2] Wikipedia, „Sicherer Hash-Algorithmus": [http://de.wikipedia.org/wiki/SHA-1]

[3] Gutterman, Pinkas, Reinman, „Analysis of the Linux Random Generator": [http://www.gutterman.net/publications/GuttermanPinkasReinman2006.pdf]

[4] Jared Bouck, „Alpha Radiation Visualizer": [http://inventgeek.com/Projects/alpharad/overview.aspx]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr Buch „Linux Treiber entwickeln" ist seit Juni in zweiter Auflage im Handel.

Anzeige

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 31

Kern-Technik

Mit der Integration der hochauflösenden High-Resolution-Timer lässt Linus Torvalds die schleichende Unterwanderung der traditionellen, auf Jiffies basierenden Zeitverwaltung zu. Die Auswirkungen auf den Linux-Kernel sind gewaltig. Eva-Katharina Kunst, Jürgen Quade



Im Jahr 1905 stellte Albert Einstein seine spezielle Relativitätstheorie in den „Annalen der Physik“ der Öffentlichkeit vor und beendete damit die Jahrtausende alte Vorstellung, die Zeit verginge für alle gleich. Ein wundersames Phänomen: Zeit ist nicht absolut, sondern relativ und hängt vom Betrachter ab, und erstaunlicherweise stehen Zeit und Raum auch noch in direkter Beziehung zueinander.

Auch der Linux-Kernel baute jahrelang auf eine interne, konstant laufende Zeitbasis. Doch mit der Übernahme des High Resolution Timer Code von Thomas Gleixner und Ingo Molnar – kurz Hrtimer genannt – hat die Relativitätstheorie (wenn auch nicht die von Albert Einstein) auch in den Linux-Kernel Einzug gehalten. Der neue Code bildet nicht nur die Basis für wirklich hoch zeitauflösende Timer, sondern auch für ein beeindruckend hochgenaues Zeitverhalten (High Precision Timer).

Böser Admin

Ein Problem bei der Zeitzählung im Linux-System ist der ungeduldige Admin, der in Unkenntnis von »ntp« und »adjtimex« mit der Funktion »settimeofday()« über »date« oder »rdate« kleinere oder sogar größere Korrekturen an der aktuellen Systemzeit vornimmt. Dass die Systemuhr bei diesem Vorgehen einen Hüpfer macht, vertragen nicht alle Systemkomponenten gut: Bei einer abrupten Sommerzeitumstellung wird beispielsweise aus einer Sekunde Wartezeit plötzlich mehr als ein Stündchen.

Mit dem neuen Timercode – seit Kernel 2.6.16 im Standardkernel – geht es besser (zur alten Lösung siehe **[1]**). Jetzt kann der Programmierer angeben, ob eine Funktion nach einer angegebenen Zeitdauer (also relativ) aufgerufen werden soll oder zu einem bestimmten Zeitpunkt (absolut). Dazu muss er lediglich ein Hrtimer-Objekt reservieren und bei der Initialisierung dieses Objekt einer von zwei Zeitquellen zuweisen.

Tick für Tick

Die Zeitquelle »CLOCK_MONOTONIC« zählt unbeirrt Timer-Ticks im System. Auf einem Linux-PC mit neuerem Kernel

Listing 1: »hrtimer.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/ktime.h>
04
05 struct hrtimer hrt_monotonic, hrt_realtime;
06
07 static int print_jiffies( struct hrtimer *hrt )
08 {
09    printk("%ld jiffies ( %p )\n", jiffies,
   hrt );
10    return HRTIMER_NORESTART;
11 }
12
13 static int __init mod_init(void)
14 {
15    struct timespec tp;
16    ktime_t tim;
17
18    hrtimer_init( &hrt_monotonic, CLOCK_
   MONOTONIC, HRTIMER_REL );
19    hrtimer_init( &hrt_realtime, CLOCK_
   REALTIME, HRTIMER_ABS );
20    hrt_monotonic.function = print_jiffies;
21    hrt_realtime.function = print_jiffies;
22    tim = ktime_set( 20, 0 );
23    hrtimer_start( &hrt_monotonic, tim,
   HRTIMER_REL );
24    tim = ktime_add( ktime_get_real(), tim );
25    hrtimer_start( &hrt_realtime, tim,
   HRTIMER_ABS );
26    printk("Timer activated at %ld jiffies\n",
   jiffies );
27    return 0;
28 }
29
30 static void __exit mod_exit(void)
31 {
32    hrtimer_cancel( &hrt_monotonic );
33    hrtimer_cancel( &hrt_realtime );
34 }
35
36 module_init( mod_init );
37 module_exit( mod_exit );
38
39 MODULE_LICENSE("GPL");
```

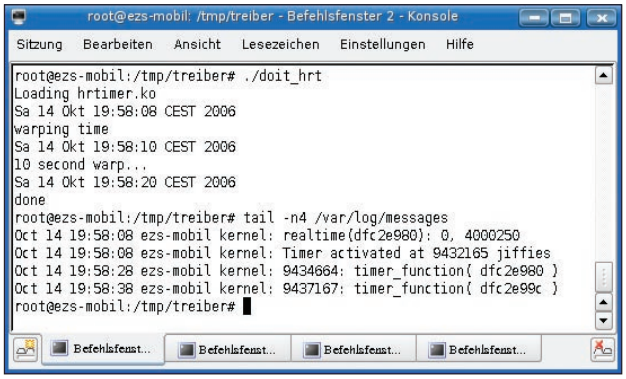


Abbildung 1: Die Zeitquelle Monotonic berücksichtigt eventuelle Zeitsprünge nicht, die Zeitquelle Realtime hingegen schon.

kommen pro Minute 250 Zählerpunkte zusammen. Tick für Tick wächst der Zählerstand, selbst wenn der Admin die Uhr kaltblütig zurückdreht. Auf solche Missetaten reagiert die Zeitquelle »CLOCK_REALTIME« aber empfindlich. Beim Vorstellen der Systemuhr warten zugeordnete Timerfunktionen länger. Wird die Uhr zurückgedreht, verkürzt sich die Wartezeit entsprechend.

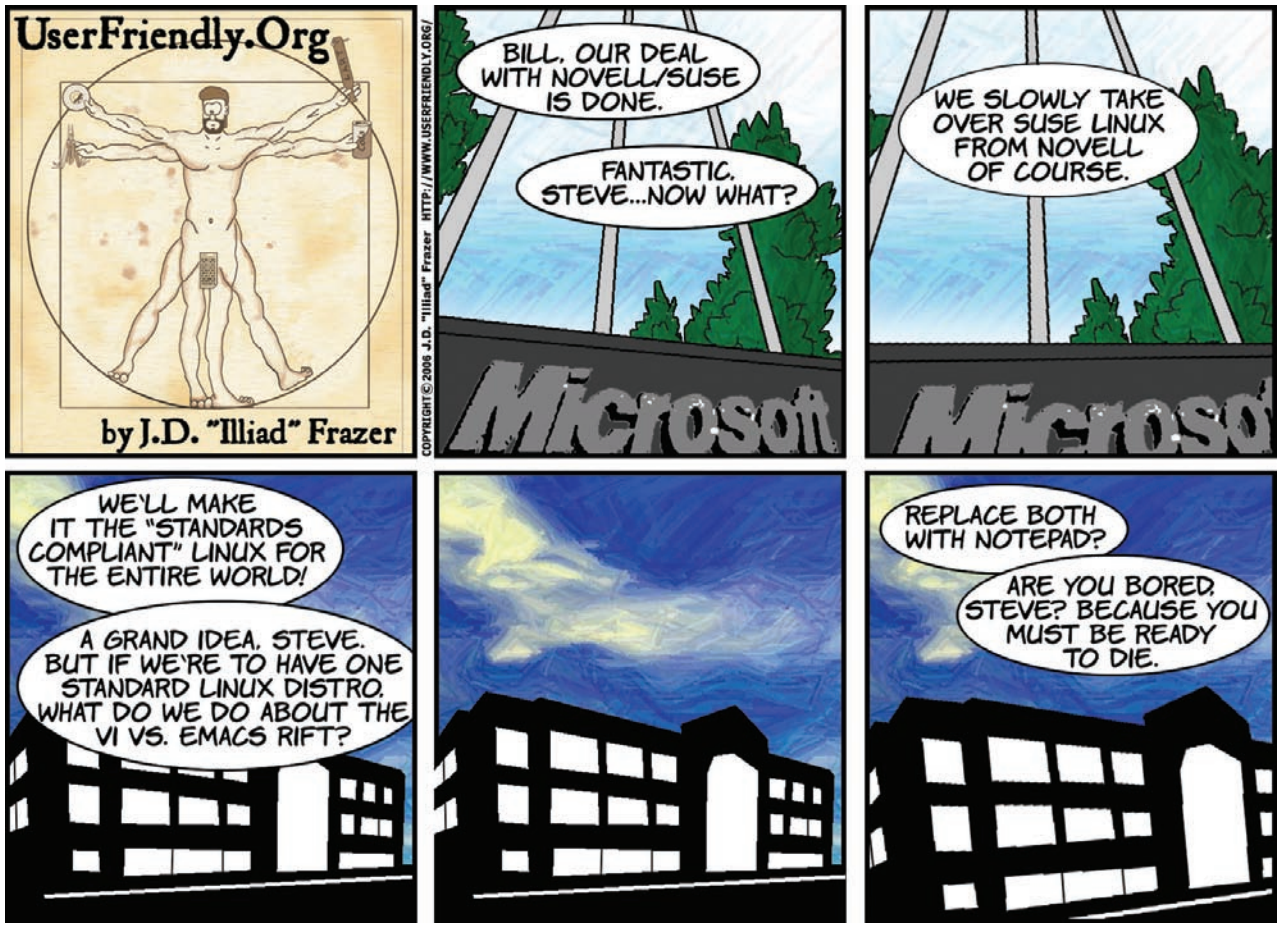
Der Code in Listing 1 demonstriert dieses unterschiedliche Verhalten und zeigt damit zugleich die Verwendung der Hrtimer. Wie bereits die Namen der Variablen»hrt_monotonic« und »hrt_realtime« vermuten lassen, werden diese beiden Hrtimer-Objekte bei der Initialisierung mit Hilfe der Funktion »hrtimer_init()« den korrespondierenden Zeitquellen »CLOCK_MONOTONIC« und »CLOCK_REALTIME« zugeordnet (Zeilen 18 und 19).

Die Funktion »hrtimer_start()« aktiviert in den Zeilen 23 und 25 beide Timer so, dass sie nach 20 Sekunden die Timerfunktion »print_jiffies()« aufrufen. Der Code lässt sich mit dem Makefile aus Listing 2 übersetzen und mit »insmod hrtimer.ko« in den Kernel laden. Im Syslog ist dann zu sehen, dass die Funktion »print_jiffies()« wie erwartet nach exakt 20 Sekunden zweimal abläuft.

Manipuliert man jedoch während dieser 20 Sekunden die Systemzeit, sieht das Ergebnis anders aus. Abbildung 1 zeigt den Fall, dass der Admin die Systemzeit während des Wartens um 10 Sekunden vorstellt. Der Timer »hrt_realtime« bekommt diese Modifikation mit und verkürzt die Wartezeit um 10 Sekunden. Danach läuft die Funktion »print_jiffies()« ab, die ihre Ausgabe gemäß der im Logfile protokollierten Zeit (»19:58:28«) zum richtigen Zeitpunkt macht.

Der Timer »hrt_monotonic« dagegen ignoriert den Zeithüpfer, wartet genau 20 Sekunden und ruft dann die Funktion auf. Absolut betrachtet kommt dieser Aufruf, wie im Logfile zu sehen, 10 Sekunden zu spät (»19:58:38«). Die exakte Wartezeit ist auch aus dem ebenfalls mit ausgegebenen Jiffie-Wert abzulesen. Während der Monotonic-Timer die erwarteten 5000 Jiffies (»9432165 + 5000«)

User Friendly – der monatliche Comic im Linux-Magazin



mehr anzeigt, hat der Realtime-Timer nur 2500 Jiffies lang gewartet.

Die vom Hrtimer aufgerufene Funktion wird im Kontext des Timer-Softinterrupt abgearbeitet, also im Interrupt-Kontext. Damit steht nur ein eingeschränkter Funktionsumfang bereit. Bei umfangreicheren Aufgaben würde die Timerfunktion also typischerweise eine Workqueue oder einen Kernelthread aufwecken, um die Arbeit zu verrichten.

Ein weiterer neuralgischer Punkt findet sich in der Initialisierung des Timers »hrt_realtime« (Listing 1, Zeile 19): Ein Timer, der der Zeitquelle »CLOCK_REALTIME« zugeordnet werden soll, muss den Startzeitpunkt der Timerfunktion als Absolut-Wert erhalten. Darauf ist bereits bei der Initialisierung über die Funktion »hrtimer_init()« zu achten. Wenn hier nicht das Flag »HRTIMER_ABS« gesetzt ist, ordnet der Kernel den Timer stillschweigend der Zeitquelle »CLOCK_MONOTONIC« zu.

Wer das kleine Experiment selbst nachvollziehen möchte, findet in Listing 3 eine kurze Applikation, die mit Superuser-Rechten ausgestattet die Systemzeit vorstellt. Der Aufruf von »make warp_time« übersetzt das Programm. Es versteht sich von selbst, dass nicht nur die Zeit hüpft, wenn man diese Software auf Produktivsystemen ausprobiert.

Das kurze Skript aus Listing 4, in jenem Verzeichnis ausgeführt, in dem sich sowohl das Kernelmodul »hrtimer.ko« als auch das Zeitsetzungsprogramm »warp_time« befinden, demonstriert das Verhalten, das in Abbildung 1 zu sehen ist. Wichtig ist es, nach Abschluss des Experiments die Uhrzeit wieder zurückzusetzen, entweder mit »warp_time« und einem negativen Parameter oder beispielsweise »ntpdate de.pool.ntp.org«.

Moderne Technik

Die Entwickler des Hrtimer-Code kommen aus dem Umfeld des Realtime-Computing. Kein Wunder also, dass einfaches Locking, optimierte Datenstrukturen und übersichtliche Abläufe zum Einsatz kommen. Neben einem Red-Black-Tree (siehe [2]), der aktivierte Timer effizient einsortiert, wird der neue Zeitdatentyp »ktime_t« eingeführt. Er basiert nicht mehr auf Timerticks, den bekannten Jiffies, sondern auf Nanosekunden.

Bitparade

64-Bit-Systeme handhaben den neuen Zeittyp anders als 32-Bit-Systeme, aber für den Programmierer transparent. Auf 64-Bit-Systemen, für die die Variable optimiert ist, kommt eine skalare Version zum Einsatz: Zeitwerte werden als 64-Bit-Nanosekunden abgelegt. Auf 32-Bit-Maschinen ist der 64-Bit-Wert dagegen in zwei 32-Bit-Wörter eingeteilt (Union-Variante). Das eine Wort speichert die Sekunden, das andere die Nanosekunden ab – übrigens in normalisierter Form: Der Nanosekundenanteil darf niemals größer als 1 Sekunde werden.

Die beiden 32-Bit-Wörter sind zudem auf einer Little-Endian- anders im Speicher abgelegt als auf einer Big-Endian-

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m := hrtimer.o
03
04 else
05 KDIR := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09     $(MAKE) -C $(KDIR) M=$(PWD) modules
10 endif
```

Listing 3: Zeitsprung »warp_time.c«

```
01 #include <stdio.h>
02 #include <sys/time.h>
03
04 int main( int argc, char **argv )
05 {
06    struct timeval tv;
07    unsigned long ttw=10;
08
09    if( argc==2 )
10       ttw = strtoul( argv[1], NULL, 0 );
11    printf("%d second warp...\n", ttw);
12    gettimeofday( &tv, NULL );
13    tv.tv_sec += ttw;
14    if( settimeofday( &tv, NULL ) ) {
15       perror( "settimeofday" );
16    }
17    return 0;
18 }
```

Tabelle 1: Funktionsübersicht »ktime_t«

Funktionsname	Beschreibung
ktime_t ktime_set(const long secs, const unsigned long nsecs);	Setzt eine »ktime_t«-Variable auf Basis von »secs« und »nsecs«. Die Funktion gibt den »ktime_t«-Wert zurück.
ktime_t ktime_sub(const ktime_t lhs, const ktime_t rhs);	Subtrahiert »rhs« von »lhs« und gibt das Ergebnis zurück.
ktime_t timespec_to_ktime(struct timespec ts);	Konvertiert den im »struct timespec«-Format übergegebenen Wert »ts« und gibt diesen im »ktime_t«-Format zurück.
ktime_t timeval_to_ktime(struct timeval tv);	Konvertiert den im »struct timeval«-Format übergebenen Wert »tv« und gibt ihn im »ktime_t«-Format zurück.
ktime_t ktime_add(const ktime_t add1, const ktime_t add2);	Addiert »add1« und »add2« und gibt das Ergebnis im »ktime_t«-Format zurück.
ktime_t ktime_add_ns(const ktime_t kt, u64 nsec);	Addiert den »ktime_t« »kt« zu den im Skalarformat übergebenen Nanosekunden »nsec« und gibt das Ergebnis als »ktime_t« zurück.
ktime_t timespec_to_ktime(const struct timespec ts);	Konvertiert den als »struct timespec« übergebenen Wert »ts« und gibt ihn als »ktime_t« zurück.
ktime_t timeval_to_ktime(const struct timeval tv);	Konvertiert den als »struct timeval« übergebenen Wert »tv« und gibt diesen als »ktime_t« zurück.
static inline struct timespec ktime_to_timespec(const ktime_t kt);	Gibt »kt« als Wert vom Typ »struct timespec« zurück.
static inline struct timeval ktime_to_timeval(const ktime_t kt);	Gibt »kt« als Wert vom Typ »struct timeval« zurück.
u64 ktime_to_ns(const ktime_t kt)	Gibt »kt« als Wert in Nanosekunden (skalar) zurück.
void ktime_get_ts(struct timespec *ts);	Legt in »ts« die Zeit der Zeitquelle »CLOCK_MONOTONIC« als Wert vom Typ »struct timespec« ab.
ktime_t ktime_get_real_ts(struct timespec *ts);	Legt in »ts« die Zeit der Zeitquelle »CLOCK_REALTIME« als Wert vom Typ »struct timespec« ab.
ktime_t ktime_get_real(void);	Gibt die Zeit der Zeitquelle »CLOCK_REALTIIME« als Wert vom Typ »ktime_t« zurück. Für diese Funktion findet sich in der Headerdatei kein Prototyp.

Maschine. Durch diese spezielle Art der Speicherung lassen sich die 32-Bit-Wörter auch wie ein 64-Bit-Wert verarbeiten. Funktionen und Makros, die den Umgang mit dem neuen, von der Prozessorarchitektur abhängigen »ktime_t«-Objekten erleichtern, führt Tabelle 1 auf. Listing 1 zeigt neben dem Umgang mit den Hrtimer-Objekten auch den Umgang mit dem neuen Zeitdatentyp.

Hrtimer als solche stehen aber nicht nur den In-Kernel-Usern zur Verfügung, also beispielsweise Gerätetreibern. Da »nanosleep()«, »getitimer()«, »setitimer()« und Posix-Timer an den neuen Hrtimer-Code gekoppelt sind, machen Applikationen von Hrtimer hinter den Kulissen bereits ausgiebig Gebrauch.

Tabelle 2 listet weitere Funktionen, die im Umgang mit High-Resolution-Timern nützlich sind. Alle Funktionen sind nur in Kernelmodulen mit GPL-Lizenz nutzbar. Erwähnenswert ist die Funktion »hrtimer_get_res()«, die die Auflösung des Timers zurückgibt. Wer etwa die Funktion »mod_init()« um die Zeilen

```
struct timespec tp;

hrtimer_get_res( CLOCK_REALTIME, &tp );
printk("realtime(%p): %ld, %ld\n",
    &hrt_realtime, tp.tv_sec, tp.tv_nsec );
```

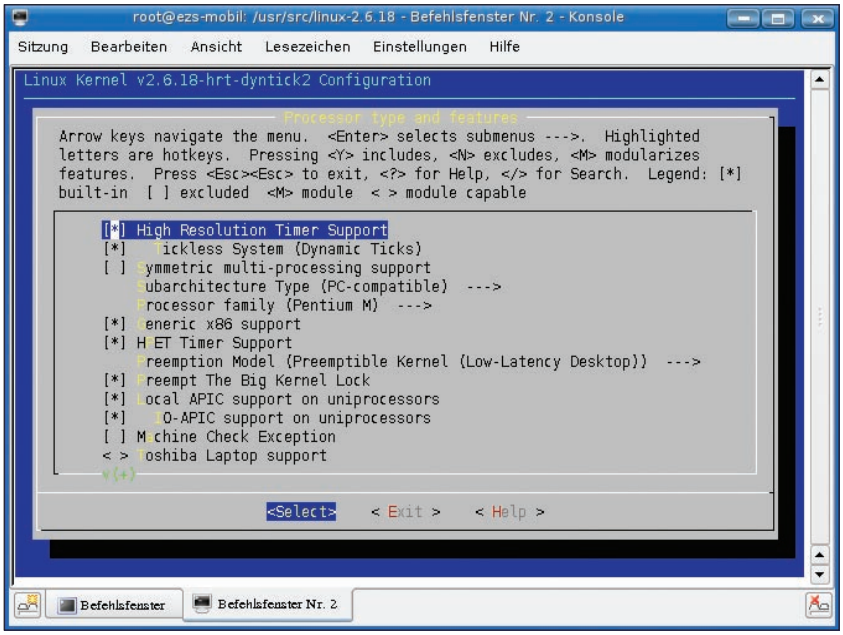


Abbildung 2: Nur wer nach dem Patchen des Kernels den High-Resolution-Timer und das Dyntick-Patch aktiviert, kommt in den Genuss erheblich verkürzter Latenzzeiten.

ergänzt, sieht im Syslog, dass auf einer PC-Plattform die Hrtimer mit einer Auflösung von nur 4 Millisekunden oder 250 Hz arbeiten. Dabei lässt der Name doch anderes vermuten. Mit dem Hrtimer-Code allein, wie er in den Kerneln 2.6.16 bis 2.6.19 implementiert ist, bietet der Kernel noch keine höhere Auflösung oder höhere Genauigkeit als die des Systemtakts. Wer seinen Kernel aber mit dem auf der Kernel-Mailingliste kursierenden Dyntick-Patch tunt (siehe [5]), darf sich auf ein Aha-Erlebnis einstellen: Statt mit Millisekunden hat er es jetzt mit Mikrosekunden zu tun (siehe Kasten „Dynamische Ticks“). ▶

Tabelle 2: Hrtimer-Funktionsübersicht

Funktionsname	Beschreibung
extern void hrtimer_init(struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode);	Initialisiert den Timer »timer« und ordnet ihn der Zeitquelle »CLOCK_REALTIME« oder »CLOCK_MONOTONIC« zu. Ein »mode« von »HRTIMER_ABS« bedeutet, dass die Zeit absolut angegeben wird (für »CLOCK_REALTIME« ist diese Wahl zwingend). Falls »mode« »HRTIMER_REL« ist, wird die Zeit relativ angegeben.
extern int hrtimer_start(struct hrtimer *timer, ktime_t tim, const enum hrtimer_mode mode);	Aktiviert den Timer »timer« auf der aktuellen CPU, sodass im Fall von »mode=HRTIMER_REL« nach der Zeitdauer »tim« oder im Fall von »mode=HRTIMER_ABS« zum Zeitpunkt »tim« die Timerfunktion »timer->function« (im Interrupt-Kontext) startet. Gibt im Erfolgsfall 0 zurück und eine 1, falls »timer« bereits aktiviert war.
extern int hrtimer_cancel(struct hrtimer *timer);	Die Funktion deaktiviert »timer«. Falls zum Zeitpunkt des Aufrufs die »timer->function« abgearbeitet wird, wartet die Funktion bis zum Ende der Abarbeitung. Die Funktion gibt 0 zurück, falls »timer« inaktiv war, sonst 1 (»timer« ist abgearbeitet).
extern int hrtimer_try_to_cancel(struct hrtimer *timer);	Diese Funktion versucht »timer« zu stoppen. Sie gibt 0 zurück, falls »timer« inaktiv war, und 1, falls »timer->function« ausgeführt worden ist, und -1, falls gerade »timer->function« ausgeführt wird und nicht gestoppt werden kann.
extern ktime_t hrtimer_get_remaining(const struct hrtimer *timer);	Gibt die verbliebene Wartezeit bis zum Start der Callback-Funktion »timer->function« zurück.
extern int hrtimer_get_res(const clockid_t which_clock, struct timespec *tp);	Die Funktion speichert die Auflösung der Zeitquelle »which_clock« in »tp« ab. »which_clock« kann entweder den Wert »CLOCK_REALTIME« oder »CLOCK_MONOTONIC« haben. Die Funktion gibt grundsätzlich 0 zurück.

Listing 4: Skript »doit_hrt«

```
01 #/bin/bash
02 echo "Loading hrtimer.ko"
03 insmod hrtimer.ko
04 sleep 2
05 echo "warping time"
06 ./warp_time 10
07 echo "done"
```

Listing 5: »linux/ktime.h«

```
01 ...
02 typedef union {
03         s64     tv64;
04 #if BITS_PER_LONG != 64 && !defined(CONFIG_KTIME_
   SCALAR)
05         struct {
06 # ifdef __BIG_ENDIAN
07         s32     sec, nsec;
08 # else
09         s32     nsec, sec;
10 # endif
11         } tv;
12 #endif
13 } ktime_t;
14 ...
```

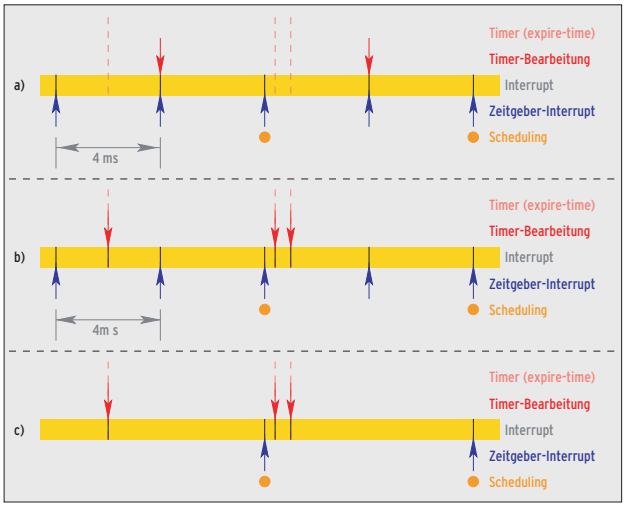


Abbildung 3: Die Zeitverwaltung im Linux-Kernel bis Version 2.6.19 basiert auf dem periodischen Zeitgeber-Interrupt ((a), blaue Pfeile). Kommende Versionen erzeugen zusätzlich dynamisch Interrupts, die die Timergenauigkeit erhöhen ((b), rote Pfeile). In einem nächsten Schritt sollen die dynamischen Interrupts den klassischen Zeitgeber-Interrupt ablösen (c).

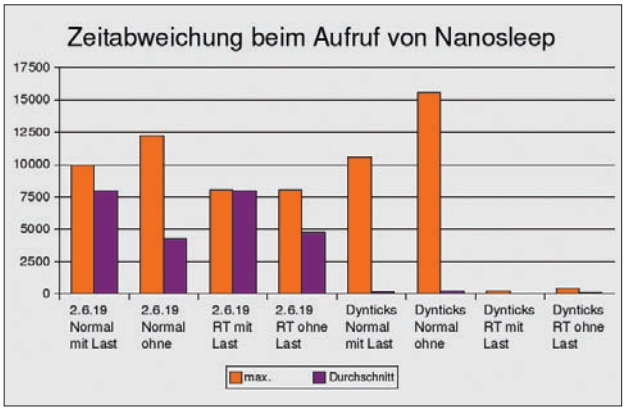


Abbildung 4: Ein mit den Dynticks gepatchter Hrtimer-Kernel zeigt Latenzzeiten, die im Durchschnitt deutlich unter 100 Mikrosekunden liegen (lila Balken). Die mit RT gekennzeichneten Echtzeitprozesse scheinen bei diesem Kernel grundsätzlich mit hoher Präzision zu reagieren.

Eigene Messungen bestätigen die in [3] veröffentlichten Zahlen. Abbildung 4 zeigt Abweichungen, die beim Aufruf der Funktion »nanosleep()« von der der Funktion übergebenen Soll-Schlafenszeit auftreten. Auf dem Testsystem – Pentium-M-Notebook mit 1,4 GHz – konnten die Autoren einen Fehler von durchschnittlich gut 5 Millisekunden (lila Balken) messen. Maximal waren es sogar 12,5 Millisekunden (orange).

Ein mit Hrtimer und Dynticks gepatchter Kernel spielt bei normalen Prozessen mit den Maximalzeiten in der gleichen Liga, die Durchschnittszeiten – relevant für weiche Echtzeitsysteme – sind mit unter 100 Mikrosekunden aber erheblich besser. Mehr noch: Versehen mit Realzeitpriorität ließ sich keine Zeitabweichung messen, die mehr als 500 Mikrosekunden betrug. Dabei war das Verhalten relativ unabhängig davon, ob das System unter Last stand oder nicht.

Einer zu viel

Mit den Hrtimern hält ein zweites Zeitverwaltungssystem Einzug in den Linux-Kernel. Bislang ergänzen sich Low-Resolution- und Low-Precision- (das bisherige Timekeeping) sowie High-Resolution- und High-Precision-Zeitverwaltung. Eine gelungene Symbiose mit optimierter Aufgabenverteilung, so die Werbung der Kernelprogrammierer. Doch die Hrtimer ziehen mehr und mehr Aufgaben an sich. Ist vielleicht doch ein Zeitverwaltungssystem zu viel im Kernel?

Thomas Gleixner und Ingo Molnar merken zu dieser Frage an [4]: „Die Implementierung lässt Raum für weitere Entwicklungen, wie die eines völlig ohne periodischen Takt betriebenen Systems (bei dem der Scheduler die Zeitscheibe kontrolliert), Profiling mit variabler Frequenz und dem kompletten Entfernen der Jiffies (Kernel-interne Zeiteinheit, die auf dem periodischen Tick basiert) in der Zukunft." *(ofr)* ■

Dynamische Ticks

Möglicherweise wird die Kernelversion 2.6.20 bereits die auf der Kernel-Mailingliste kursierenden Dyntick-Patches enthalten (siehe [5] und Abbildung 2), die zu spürbar kürzeren Latenzzeiten führen.

Technisch basiert dieses Wunder darauf, nicht mehr nur zu den periodisch kommenden Zeitgeber-Interrupts (Ticks) den Ablauf der Timer zu überprüfen, sondern im Vorhinein den nächsten Ablaufpunkt auszurechnen und einen Hardware-Timer dafür aufzuziehen. Abbildung 3 erläutert das Timekeeping der verschiedenen Linux-Kernel.

a) Die Zeitverwaltung bis Kernel 2.6.16 basiert auf einem periodischen Interrupt. Beim Auftreten dieses Zeitgeber-Interrupt ruft der Kernel zuvor abgelaufene Timerfunktionen auf und stößt – falls dies notwendig ist – das Scheduling an.

b) Das auf periodischen Timerticks basierende Zeitverwaltungssystem koexistiert mit den neuen Hrtimern, die mit Unterstützung der dynamische Ticks hochpräzise arbeiten. Die Dynticks sorgen dafür, dass immer dann Interrupts entstehen, wenn ein Timer abläuft. Sie sollen voraussichtlich ab Version 2.6.20 im Linux-Kernel integriert sein.

c) Für spätere Linux-Versionen ist es vorgesehen, den periodischen Zeitgeber-Interrupt ebenfalls durch die Dynticks zu ersetzen. Überflüssige Zeitgeber-Interrupts gehören dann der Vergangenheit an.

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 27: Linux-Magazin 04/06, S. 118

[2] Tim Schürmann, „Bunte Bäume": Linux-Magazin 12/06, S.70

[3] Gleixner, Niehaus, „Hrtimers and beyond: Transforming the Linux Time Subsystem. Proceedings of the Linux Symposium": Volume One, 2006, S. 333: [http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf]

[4] Thomas Gleixner und Ingo Molnar, „High resolution timer design notes": [http://rt.wiki.kernel.org/index.php/High_resolution_timer_design_notes]

[5] Dyntick-Patch: [http://tglx.de/projects/hrtimers/2.6.18/patch-2.6.18-hrt-dyntick2.patch]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr Buch „Linux-Treiber entwickeln" ist seit Juni in zweiter Auflage im Handel.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 32

Kern-Technik

Mit Hilfe der Mmap-Funktion lassen sich Kontextwechsel und Kopieraktionen einsparen. Diese Folge der Kern-Technik verrät Treiber- und Anwendungsentwicklern, wie es funktioniert. Eva-Katharina Kunst, Jürgen Quade



Um handelsüblicher Hardware die auf der Packung angegebenen Höchstleistungen zu entlocken, muss der Programmierer einiges an Wissen über den Kernel mitbringen. Kennt er Komponenten und Betriebsabläufe, kann er in die Trickkiste greifen, Kontextwechsel-Zeiten vermeiden und Kopiervorgänge einsparen. Die Basistechnologie hierfür stellt »mmap()« bereit. Mmap ermöglicht es der Applikation, Speicherbereiche aus dem Kernelspace in den eigenen Adressraum einzublenden (siehe Abbildung 1).

Getrennte Adressbereiche

Aus Stabilitäts- und Sicherheitsgründen besitzen Applikationen und Kernel jeweils eigene Adressräume (zum Begriff Adresse siehe Kasten „Adresstypen“). Daher ist es einer Applikation nicht möglich, die (physischen) Speicherbereiche anzusprechen, in denen der Betriebssystemkern seine Variablen ablegt oder in denen Register von Peripheriegeräten liegen (Kernelspace). Der Applikation bleibt also nichts anderes übrig, als den Betriebssystemkern per Systemcall damit zu beauftragen, Manipulationen an den Speicherzellen vorzunehmen oder Daten zwischen dem eigenen Adressraum (Userspace) und dem Kernelspace zu transferieren. Die Nachteile liegen auf der Hand: unnötige Kopieraktionen beziehungsweise zeitlich teure Systemcalls durch den Kontextwechsel.

Wenn es beim Programmieren auf jede Millisekunde ankommt, hilft oft »mmap()« weiter. Es blendet Teile des Kernelspeichers in den Adressraum der Applikation ein. So kann die Applikation ohne Umwege auf die Daten respektive auf die hinter einem (Dual-Port-)Memory liegende Hardware zugreifen. Auf diese Art blendet beispielsweise auch der X-Server den Speicher der Grafikkarte in den eigenen Adressraum ein.

Interface: Gerätedatei

Applikationen können den Beschleunigungstrick allerdings nur dann nutzen, wenn der zugehörige Gerätetreiber »mmap()« unterstützt. Der Treiber übernimmt also die Verantwortung dafür, dass bestimmte Speicherbereiche einer Applikation direkt zugänglich gemacht werden. Listing 1 zeigt, wie Anwendungsentwickler den direkten Weg zur Hardware einschlagen. Sie öffnen dazu in gewohnter Weise den zugehörigen Treiber durch Angabe der Gerätedatei (in Listing 1 ist dies die Datei »/dev/mmap«). Die Gerätedatei repräsentiert den Treiber, der die Freigabe auf den Speicher erteilen soll. Danach rufen sie »mmap()« auf, wobei die Applikation mit dem ersten Parameter einen Vorschlag machen darf, an welche Adresse im eigenen Adressraum der Speicher eingeblendet werden soll. Der zweite Parameter spezifiziert die Länge des Speicherbereichs. Vorteilhaft ist es, wenn diese ein Vielfaches einer Seitengröße, also typischerweise ein Vielfaches von 4096 Byte ist.

Weiterhin lassen sich noch die Speicherschutzbits, die so genannten Protections, festlegen: »PROT_READ«, »PROT_WRITE«, »PROT_EXEC« und »PROT_NONE«. Der vierte Parameter »MAP_SHARED« schließlich legt fest, dass der Kernel den Speicherbereich schützt (lockt), sodass er die zugehörigen Speicherseiten nicht auslagert (swappt). Zum Schluss folgt der Offset innerhalb des Speicherbereichs, ab dem das Mapping passieren soll.

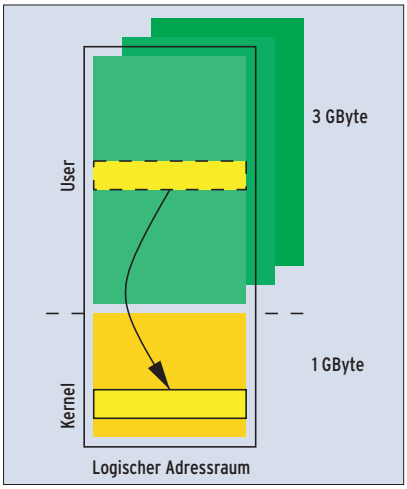


Abbildung 1: Die Funktion »mmap()« blendet die Speicherseite aus dem Kernelspace in den Userspace ein. Die Applikation kann nun direkt auf den Kernelspeicher zugreifen.

Um Treiber-seitig »mmap()« zu unterstützen, ist vom Treiber im Wesentlichen nur eine Funktion »drv_mmap()« zu implementieren. Diese Funktion bekommt zwei Parameter übergeben. Der erste repräsentiert die zugreifende Applikation, die Treiberinstanz. Der zweite zeigt auf eine so genannte Virtual Memory Area (VMA). Sie beschreibt als zentrale Datenstruktur einen semantisch zusammengehörigen Speicherbereich der Applikation. Ein solcher Speicherbereich ist durch identische Zugriffsrechte gekennzeichnet. Ein Rechenprozess hat unter Linux mindestens vier VMAs: das Codesegment, den Bereich der initialisierten Daten, den Bereich der uninitialisierte Daten (BSS, Block Started by Symbol) und den Stack.

Listing 2 zeigt die Speicherregionen eines Hello-World-Beispiels, ausgegeben durch »cat /proc/*Hello-World-Programm-PID*/maps«. Die erste Spalte gibt den Adressbereich im Adressraum der Applikation an, die zweite Spalte bezeichnet die Zugriffsrechte, die dritte Spalte den Offset innerhalb des Segments, die vierte schließlich die Major- und Minor-Nummer, als fünfte folgt die Inode. Ist der Inhalt des Speicherbereichs nicht aus einer normalen Datei heraus entstanden, wird im Feld »Dev« die Major-Nummer zu Null gesetzt.

Bei den Rechten bedeutet »r« Lesen, »w« Schreiben, »x« Ausführen, »s« eine gemeinsame Nutzung mit anderen Prozessen und »p«, dass der Speicherbereich bei einem ersten Schreibzugriff kopiert wird (COW, Copy on Write). Die in Listing 2 dargestellte Region »[vdso]« (Virtually Dynamically Shared Object) repräsentiert übrigens die Speicherseite, über die eine Applikation Systemcalls aufruft (siehe hierzu auch [1]).

Ruft eine Applikation die Funktion »mmap()« auf, erzeugt und initialisiert der Kernel ein VMA-Objekt. Dieses Objekt repräsentiert die neue Speicherregion und wird von der Funktion »drv_mmap()« vervollständigt. Dazu kann das Programm dem Kernel – durch Aufruf von »remap_pfn_range()« – die physikalische Adresse des einzublendenden Speicherbereichs plus dessen Länge übergeben. Der Kernel richtet dann das entsprechende Mapping ein.

Die physische Adresse muss sich an einer Seitengrenze orientieren und wird als so genannte Page Frame Number (PFN) übergeben. Diese Nummer erhält man durch Einteilung des Speichers in gleich große Blöcke – die Pages – und deren Zählung, beginnend mit 0.

Die PFN kann der Entwickler sehr einfach berechnen, indem er die physische Adresse um die Bits, die zur Adressierung innerhalb einer Page benötigt werden, nach rechts schiebt (»PAGE_SHIFT«, siehe Listing 3). Um die logische Adresse, über die die Applikation später auf die Seite zugreifen kann, muss er sich übrigens nicht kümmern. Der Kernel hat sie bereits bei Aufruf der Funktion »drv_mmap()« ausgewählt.

Probleme beim Verändern

Es gibt allerdings noch zwei Fallstricke. Die erste Warnung betrifft den Systemcall »mremap()«. Hiermit können Applikationsprogrammierer ein bestehendes Mapping in seiner Größe verkleinern oder erweitern. Der Kernel akzeptiert eine solche Anfrage zunächst klaglos und erweitert in seinen Datenstrukturen den eingeblendeten Speicherbereich. Erst wenn die Applikation auf den erwei-

Adresstypen

Der Begriff Adresse ist nicht eindeutig. Innerhalb des Kernels sind fünf Varianten zu unterscheiden (siehe Abbildung 2). Logische oder User Virtual Address nennen die Entwickler jene Adresse, die innerhalb einer Applikation verwendet wird. Die physische Adresse dagegen ist jene, mit der die CPU auf den physischen Speicher zugreift, also den von der Hardware zur Verfügung gestellten linearen Speicherbereich.

In einigen Fällen verwendet Peripherie für den Zugriff auf den physischen Speicher eine andere Adresse. Diese Bus-Adresse wird beispielsweise für den Zugriff per DMA benötigt. Bei den meisten Plattformen sind Bus-Adresse und physische Adresse identisch. Als Kernel-logische Adresse bezeichnen die Kernelhacker Adressen, die sie innerhalb des Kernels für den Zugriff auf den Speicher verwenden. Oft sind diese Adressen eins zu eins auf den physischen Speicher abgebildet.

Kernel Virtual Address heißen Adressen im Kernel, die nicht direkt auf den physischen Speicher abgebildet sind. Die Kernelfunktion »vmalloc()« beispielsweise gibt bei Aufruf eine solche Kernel-virtuelle Adresse zurück.

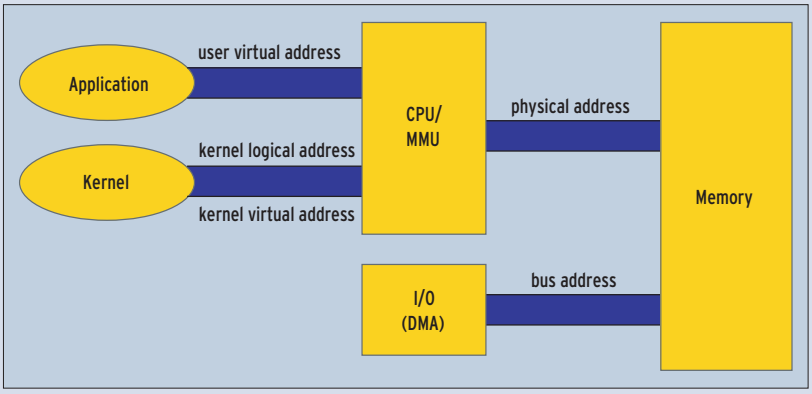


Abbildung 2: Das Betriebssystem arbeitet mit unterschiedlichen Adresstypen.

Listing 1: Mmap einer Speicherseite aus dem Kernelspace

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <unistd.h>
04 #include <sys/mman.h>
05
06 int main( int argc, char **argv )
07 {
08     int fd;
09     void *pageaddr;
10     int *ptr;
11
12     fd = open( "/dev/mmap", O_RDWR );
13     if( fd<0 ) {
14        perror( "/dev/mmap" );
15        return -1;
16     }
17     pageaddr = mmap( NULL, 4096, PROT_READ|
   PROT_WRITE, MAP_SHARED, fd, 0 );
18
19     printf("pageaddr mmap: %p\n", pageaddr );
20     if( pageaddr ) {
21        ptr = (int *)pageaddr;
22        printf("*pageaddr: %d\n", *ptr );
23        *ptr = 55;
24     }
25     munmap( pageaddr, 4096 );
26
27     return 0;
28 }
```

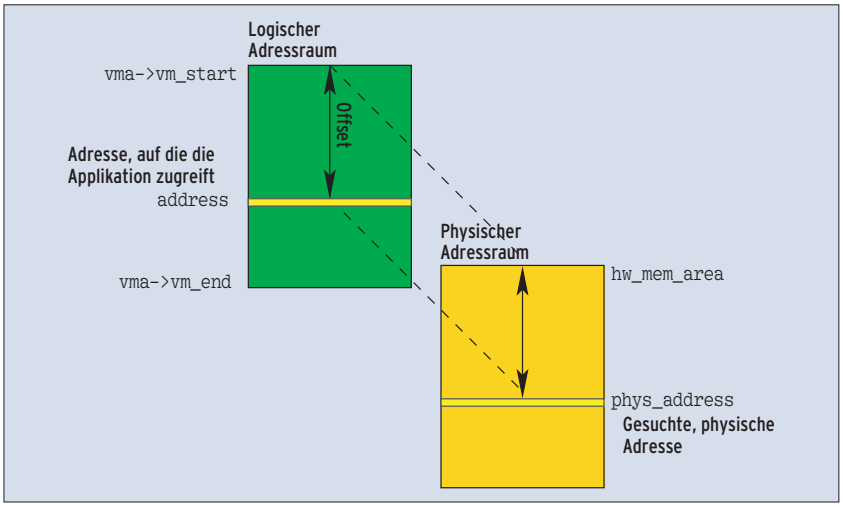


Abbildung 3: Eine Offset-Verschiebung berechnet die physische Adresse, auf die eine Applikation zugreifen möchte.

terten Bereich zugreift, wird es kritisch. Der Kernel stellt in diesem Fall fest, dass es zu der verwendeten logischen Adresse noch kein physisches Pendant gibt, das heißt, es gibt keine Zuordnung des Adressbereichs zu einer physischen Speicherseite (mit der Datenstruktur »struct page«).

Der Kernel benötigt für den Zugriff die zur logischen Adresse gehörende »struct page«. Um sie zu erhalten, ruft er die zur VMA gehörende Methode »nopage()« auf. Diese Funktion bekommt neben der VMA-Datenstruktur die Adresse im Adressraum der Applikation übergeben. Es ist also Aufgabe des Entwicklers, die Methode zu implementieren und innerhalb der Funktion »drv_mmap()« dem VMA-Objekt zuzuordnen. Im einfachsten Fall – der Treiber kann den eingeblendeten Speicherbereich nicht erweitern – ist die Methode so zu implementieren:

```
struct page *drv_vma_nopage( struct ➥
vm_area_struct *vma, unsigned long address,➥
int *type )
{
    return NOPAGE_SIGBUS;
}
```

Listing 2: Zwölf Speicherregionen bei Hello-World

```
# Adressbereich    Perms Offset  Dev   Inode      Bezeichnung
quade@ezs-mobil:/tmp$ cat /proc/21452/maps
08048000-08049000 r-xp 00000000 03:02 2829122    /tmp/hello_world
08049000-0804a000 rw-p 00000000 03:02 2829122    /tmp/hello_world
b7ddf000-b7de0000 rw-p b7ddf000 00:00 0
b7de0000-b7f0d000 r-xp 00000000 03:02 736410     /lib/tls/i686/cmov/libc-2.4.so
b7f0d000-b7f0f000 r--p 0012c000 03:02 736410     /lib/tls/i686/cmov/libc-2.4.so
b7f0f000-b7f11000 rw-p 0012e000 03:02 736410     /lib/tls/i686/cmov/libc-2.4.so
b7f11000-b7f14000 rw-p b7f11000 00:00 0
b7f23000-b7f26000 rw-p b7f23000 00:00 0
b7f26000-b7f27000 r-xp b7f26000 00:00 0          [vdso]
b7f27000-b7f40000 r-xp 00000000 03:02 703158     /lib/ld-2.4.so
b7f40000-b7f42000 rw-p 00018000 03:02 703158     /lib/ld-2.4.so
bf887000-bf89c000 rw-p bf887000 00:00 0          [stack]
```

Listing 3: Einblenden von Hardwarespeicher in den Adressraum einer Anwendung

```
01 ...
02 static int drv_mmap( struct file *instance,
03     struct vma_area_struct *vma )
04 {
05    if( (vma->vm_end-vma->vm_start) >
   hw_mem_area_size )
06       return -EINVAL;
07    if( remap_pfn_range( vma,
08       vma->start,                 // user
   virtual address
09       hw_mem_area>>PAGE_SHIFT,     // PFN -
   physical address
10       vma->vm_end - vma->vm_start, // length
11       vma->vm_page_prot ) )        // page
   protection
12       return -EAGAIN;
13
14    return 0;
15 }
16 ...
```

Die Applikation bekommt dann beim Zugriff das Signal »SIGBUS« geschickt. Unterstützt ein Treiber auch die nachträgliche Erweiterung des Speicherbereichs, sollte er die »struct page« zurückgeben, die den zusätzlichen physikalischen Speicherbereich repräsentiert.

Listing 4 zeigt, wie es geht. Zuerst bestimmt das Modul aus der übergebenen logischen Adresse die physische Adresse, auf die die Applikation zugreifen möchte. Dazu rechnet es den Offset im logischen Adressraum der Applikation aus und addiert diesen zur Startadresse des physischen Speicherbereichs (siehe Abbildung 3). Danach berechnet es die Page Frame Number, indem es das Ergebnis des ersten Schritts um »PAGE_SHIFT« Bits nach rechts schiebt.

Die erhaltene PFN überprüft es durch Aufruf der Funktion »pfn_valid()« auf Gültigkeit. Ist die PFN gültig, ruft es »pfn_to_page()« auf und erhält schließlich die gesuchte »struct page«. Bevor der Treiber diese dem Kernel übergibt, muss er durch Aufruf von »get_page()« unbedingt den Zugriffszähler auf die Seite erhöhen und »type« auf den Wert »VM_FAULT_MINOR« setzen.

Ohne Swapping

Vergisst der Treiberentwickler die Methode »drv_vma_nopage()« zu implementieren, bekommt die Applikation beim Zugriff auf den erweiterten Adressbereich keine Fehlermeldung, sondern eine Zero-Page eingeblendet. Sie greift dann also auf ganz normalen Hauptspeicher zu. Die Adresse der Methode »drv_vma_nopage()« assoziiert man übrigens mit der »struct vm_area_struct« wie in Listing 5, Zeile 23.

Die zweite Warnung betrifft den Adressraum, den die Funktion »remap_pfn_range()« einblendet. Es lassen sich nur jene Pages einblenden, die der normalen Speicherverwaltung nicht zur Verfügung stehen. Diese Seiten sind als reserviert markiert und werden daher bei der Auslagerung nicht berücksichtigt. Auf einer PC-Plattform beispielsweise sind dies der Adressbereich zwischen 640 KByte

und 1 MByte (ISA), der Adressbereich oberhalb des physischen Speichers (von PCI-Geräten) sowie der Codebereich des Kernels selbst.

Shared Mem zum Kernel

Wie das Relay-Subsystem [2] zeigt, ist es in einigen Situation dennoch sinnvoll, normalen Kernelspace in den Adressraum der Applikation einzublenden. Auch damit lassen sich sehr häufig Kopieraktionen vermeiden. Einen derart gemeinsamen Speicher zwischen Kernel- und Userland implementieren Treiberentwickler unter Auslassung der Funktion »remap_pfn_range()« wiederum mit Hilfe der Methode »drv_vma_nopage()«. Wird »drv_vma_nopage()« aufgerufen, genügt es, die entsprechende Page zurückzuliefern. Listing 5 zeigt dies in Form eines Treibers, der das virtuelle Gerät »/dev/mmap« implementiert. Als kleines Bonbon erstellt die Funktion »drv_init()« im Sys-Filesystem übrigens einen Eintrag, sodass Udev die Gerätedatei automatisch anlegt.

Der Code aus Listing 5 lässt sich leicht mit Hilfe des Makefile von [3] kompilieren. Wer das so entstehende Modul »mmap.ko« installiert, kann anschließend in der Applikation aus Listing 1 auf den eingeblendeten Kernelspeicher zugreifen. Beim ersten Zugriff auf die Speicherseite liest die Anwendung die vom Treiber abgelegte »99«. Danach schreibt sie dorthin den Wert »55« und liest diesen Wert beim Neustart direkt aus.

Viele Möglichkeiten

Das letzte Beispiel demonstriert, wie Anwendungen auf eingeblendete Speicherbereiche des Kernels zugreifen können. Richtige Programme werden normalerweise mehr Nutzdaten als nur einfache Zahlen austauschen, das Prinzip bleibt jedoch gleich. Gerade wer größere Datenmengen zwischen User- und Kernelspace austauschen und dabei Kopiervorgänge vermeiden möchte, sollte einen Blick auf Mmap werfen. *(ofr)*

Listing 4: Implementierung von »drv_nopage()«

```
01 struct page *drv_vma_nopage( struct
   vm_area_struct *vma,
02     unsigned long address, int *type )
03 {
04    unsigned long phys_address, pfn;
05    struct page *page_wanted;
06
07    phys_address = hw_mem_area +
   (address-vma->vm_start);
08       if( (vma->vm_end-vma->vm_start) >
   HW_MEM_AREA_SIZE )
09          return NOPAGE_SIGBUS;
10    pfn = phys_address >> PAGE_SHIFT;
11    if( !pfn_valid( pfn ) )
12       return NOPAGE_SIGBUS;
13    page_wanted = pfn_to_page( pfn );
14    get_page( page_wanted );
15    if( type )
16       *type = VM_FAULT_MINOR;
17    return page_wanted;
18 }
```

Infos

[1] Johan Petersson, „What is linux-gate-so.1?“: [http://www.trilithium.com/johan/2005/08/linux-gate]

[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 29: Linux-Magazin 9/06, S. 94

[3] Listings online: [http://www.linux-magazin.de/Service/Listings/2007/03/Kern-Technik]

Listing 5: Einblenden einer Kernel-Speicherseite in den Adressraum einer Anwendung

```
01 #include <linux/module.h>
02 #include <linux/device.h>
03 #include <linux/fs.h>
04 #include <linux/mm.h>
05
06 static int major;
07 static struct class *demo_class;
08 static struct page *common_page;
09
10 struct page *drv_vma_nopage( struct
   vm_area_struct *vma,
11        unsigned long address, int *type )
12 {
13    printk("drv_vma_nopage( %p, %lx, %p )\n",
   vma, address, type );
14    if( (address-vma->vm_start) > 4095 )
15       return NOPAGE_SIGBUS;
16    get_page( common_page );
17    if( type )
18       *type = VM_FAULT_MINOR;
19    return common_page;
20 }
21
22 static struct vm_operations_struct drv_vma_ops
   = {
23    .nopage = drv_vma_nopage,
24 };
25
26 static int drv_mmap( struct file *instance,
   struct vm_area_struct *vma )
27 {
28    printk("drv_mmap(%p, %p)\n", instance,
   vma);
29    vma->vm_ops = &drv_vma_ops;
30    vma->vm_flags |= VM_RESERVED;
31    return 0;
32 }
33
34 static struct file_operations fops = {
35    .mmap = drv_mmap,
36 };
37
38 static int __init drv_init(void)
39 {
40    int *ptr;
41
42    if( (major=register_chrdev(0,"mmap",&fops
   ))==0 ) {
43       printk("no major number available\n");
44       return -EIO;
45    }
46    demo_class = class_create(THIS_MODULE,
   "demo");
47    if( IS_ERR(demo_class) ) {
48       printk("no udev support\n");
49    } else {
50       printk("class_device_create...\n");
51       class_device_create(demo_class,NULL,
   MKDEV(major,0),NULL,"mmap");
52    }
53    common_page = alloc_page( GFP_USER );
54    if( !common_page ) {
55       if( !IS_ERR(demo_class) ) {
56          class_device_destroy( demo_class,
   MKDEV(major,0) );
57          class_destroy( demo_class );
58       }
59       unregister_chrdev( major, "mmap" );
60       return -ENOMEM;
61    }
62    ptr = (int *)page_address( common_page );
63    *ptr = 99; // Nur zur Demonstration!
64    return 0;
65 }
66
67 static void __exit drv_exit(void)
68 {
69    unregister_chrdev( major, "mmap" );
70    if( !IS_ERR(demo_class) ) {
71       class_device_destroy( demo_class,
   MKDEV(major,0) );
72       class_destroy( demo_class );
73    }
74 }
75
76 module_init( drv_init );
77 module_exit( drv_exit );
78 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 33

Kern-Technik

Der Linux-Kernel hat auch die Aufgabe, die Prozesse und Threads möglichst gleichmäßig auf die vorhandenen CPU-Cores zu verteilen. Einige Funktionen des Scheduling-API helfen dem Anwendungsprogrammierer dabei, den Kernel entsprechend zu beeinflussen. Eva-Katharina Kunst, Jürgen Quade



Entscheidend für die gute Skalierbarkeit eines Multicore-Systems ist die Verteilung der anstehenden Aufgaben auf die vorhandenen Prozessoren, das so genannte Load Balancing. Der von Linus Torvalds einst hierfür eingeführte große Kernel-Lock, der die Liste mit den lauffähigen Rechenprozessen (Run Queue) absichert, ist mit Kernel 2.6 einem ausgewachsenen und ausgefeilten Multi-CPU-Scheduling-Modell gewichen. Bei diesem Modell verwaltet nun jede CPU ihre eigene Run Queue über den O(1)-Scheduler (siehe Kasten „Ein-Prozessor-Scheduling“).

Stellt eine CPU eine signifikante Ungleichverteilung der Last zwischen sich selbst und einer anderen CPU fest, nimmt sie Rechenprozesse aus der Run Queue einer stark belasteten CPU und verschiebt sie in die eigene (siehe Abbildung 1). Dabei allein auf eine gleichmäßige Verteilung der vorhandenen Aufgaben zu achten, wäre kurzsichtig. Vielmehr muss der Scheduler auch die Leistungsverluste berücksichtigen, die beim Verschieben eines Thread von einem zum anderen Prozessor – der Prozessmigration – durch das unvermeidliche Flushen von Prozessor-Caches entstehen. Diese Leistungsverluste hängen stark von der verwendeten Multiprozessor-Architektur ab.

Prozessoren-Zoo

Der geringste Aufwand fällt bei einer Hyperthreading-Architektur (Symmetric Hyperthreading, SMT) an. In diesen Fällen stehen aus Software-Sicht zwei unabhängige Prozessorkerne – sprich zwei logische Prozessoren – zur Verfügung. Da sich beide Kerne aber den Hauptspeicher und die Prozessor-Caches teilen, gibt es erhebliche Abhängigkeiten zwischen den logischen Prozessoren. Die Migration eines Thread von einem logischen Kern zu dem zugehörigen Gegenstück ist so zwar kaum mit Kosten verbunden. Allerdings ist bei dieser Architektur der Leistungsgewinn auch vergleichsweise klein.

Anders beim klassischen symmetrischen Multiprozessing (SMP), aufgebaut etwa aus aktuellen Multicore-Prozessoren wie Athlon-X2 und Core 2 Duo. Bei jedem Prozesswechsel sind auch die eventuell noch in der CPU befindlichen Prozessinformationen obsolet. Noch dicker kommt es auf Numa-Systemen, bei denen Prozessoren auf die ihnen zugewiesenen Speicherbereiche schneller zugreifen können als auf das von anderen CPUs verwaltete Memory. In der Realität kommen die verschiedenen Basisarchitekturen oft in Kombination vor. Ein Numa-System besteht beispielsweise aus mehreren Knoten (Nodes), wobei jeder Knoten aus mehreren Prozessoren aufgebaut ist (SMP). Auch Hyperthreading-CPUs sind dabei einsetzbar.

Scheduling Domains

Um mit den unterschiedlichen Multiprozessor-Architekturen zurechtzukommen, setzt Linux auf das Konzept der Scheduling Domains und der Scheduling-Gruppen. Eine Scheduling Domain besteht aus mehreren Scheduling-Gruppen. Eine Gruppe repräsentiert dabei entweder eine CPU oder eine untergeordnete Scheduling Domain. Ein Zwei-Prozessor-System beispielsweise modelliert Linux in

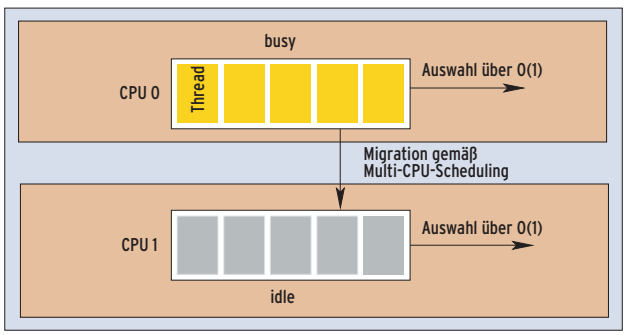


Abbildung 1: Jede CPU unterhält eine eigene Liste lauffähiger Rechenprozesse (Run Queues). Bei signifikant ungleicher Lastverteilung verschiebt der Scheduler Prozesse zwischen den Run Queues.

Ein-Prozessor-Scheduling

Die Auswahl des Rechenprozesses, der als nächster die (lokale) CPU zugeteilt bekommt, gestaltet Linux prioritätengesteuert. Die insgesamt 140 zur Verfügung stehenden Prioritäten sind in zwei Bereiche eingeteilt: In den Bereich der statischen und den Bereich der dynamischen Prioritäten. Der Unterschied: Der Scheduler erlaubt sich, die Priorität im dynamischen Bereich – von 100 bis 139 – schon mal zu verändern. Die Priorität eines Rechenprozesses im statischen Bereich – 0 bis 99 – modifiziert der Scheduler nicht. Diese Prioritäten sind für Aufgaben mit Echtzeit-Anforderungen vorgesehen, während normale Rechenprozesse typischerweise eine Priorität aus dem dynamischen Bereich erhalten.

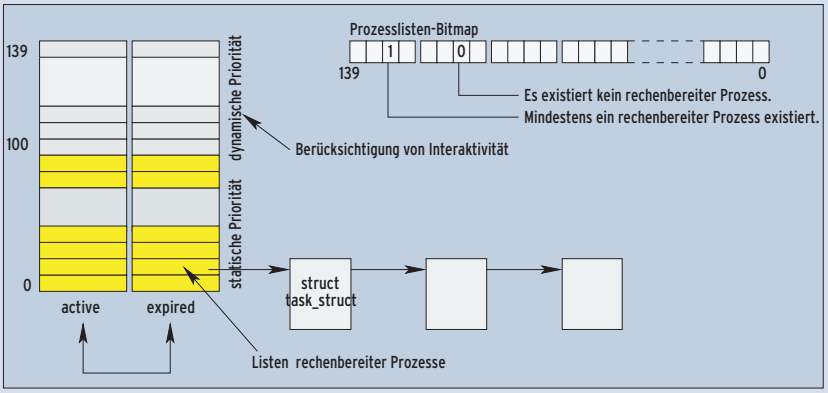


Abbildung 2: Der Kernel führt fürs Scheduling zwei Listen mit Prozessen, die mit normalen und Echtzeit-Prioritäten versehen sind.

Für jede Priorität gibt es zwei Listen, in denen die rechenbereiten Tasks und Threads eingetragen sind, die Listen »active« und »expired«. Der Scheduler wählt den ersten Prozess mit höchster Priorität aus der Active-Liste. Hat sich der Prozess nach Ablauf eines Zeitintervalls (Zeitscheibe) nicht beendet oder schlafen gelegt, hängt er ihn an die Expired-Liste an, aus der der Multi-CPU-Scheduler übrigens mit Vorliebe die zu migrierenden Rechenprozesse entnimmt.

Neben dem beschriebenen Round-Robin-Verfahren unterstützt Linux im Bereich der statischen Prioritäten (Echtzeit-Prioritäten) auch noch die Auswahl des nächsten Rechenprozesses nach dem Prinzip „Wer zuerst kommt, mahlt zuerst" (First Come First Serve).

einer Scheduling Domain mit zwei Gruppen: für jede CPU eine. Ein Rechner mit Hyperthreading-Prozessor wird übrigens ebenfalls auf eine Domain und zwei Gruppen abgebildet.

Ein heterogenes System dagegen besteht aus mehreren Domains, wobei die Scheduling-Gruppen der übergeordneten Container (Domains) nicht Prozessoren, sondern eben weitere Domains repräsentieren. Ein Beispiel für eine solche baumartige Topologie zeigt Abbildung 3.

Innerhalb der Scheduling Domain sorgt der Kernel für ausbalancierte Lastverteilung zwischen den Gruppen, das Verfahren hierzu ist konfigurierbar. So kann der Anwender beim Booten festlegen, ob beim Aufruf des Systemcalls »exec« oder »fork« ausbalanciert werden soll, ob dies beim Aufwecken eines schlafenden Thread geschieht oder nur, wenn die CPU leer läuft (siehe Tabelle 2).

Um auch einen rechenintensiven Prozess, der sich nicht schlafen legt und kein »exec« oder »fork« aufruft, lasttechnisch zu verteilen, ist für jede Scheduling Domain ein individuelles Intervall spezifizierbar, innerhalb dessen der Scheduler in jedem Fall neu ausbalanciert. Dies geschieht über den neuen Soft-IRQ »SCHED_SOFTIRQ« und die Funktion »run_rebalance_domain()«.

Linux unterscheidet prinzipiell drei Domain-Arten: SMT, SMP und Numa. Gleich beim Booten legt sich das System auf eine Domain-Topology fest. Identifiziert der Kernel eine Hyperthreading-CPU, ruft er zur Initialisierung der zugehörigen Domain das Makro »SD_SIBLING_INIT()« auf. Zur Initialisierung einer klassischen SMP-Domain dient »SD_CPU_INIT()«. Einen Numa-Node initialisiert »SD_NODE_INIT()«. Die Makros sind architekturspezifisch und finden sich zum Beispiel in »arch/i386/kernel/topology.h«. Die für die x86-Systeme verwendeten Default-Load-Balancing-Flags listet Tabelle 1 auf.

Grundsätzlich gilt: SMT-Domains balanciert der Kernel entsprechend den geringen Verlusten bei der Prozessmigration

Anzeige

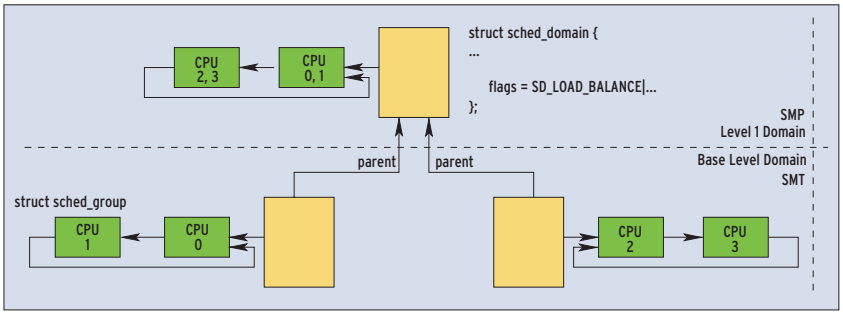


Abbildung 3: Linux bildet eine Architektur mit zwei Hyperthreading-Prozessoren auf drei Scheduling-Domains ab: zwei Basis-Domains und die übergeordnete Level-1-Domain, die alle vier CPUs umfasst. Die Basis-Domains enthalten jeweils einen Prozessor mit zwei logischen Kernen.

häufig aus, Numa-Domains dagegen eher selten. Zum Ausbalancieren ist es natürlich notwendig, die Last jeder Scheduling-Gruppe zu bestimmen. Hierfür skaliert der Kernel die Last »load« einer CPU mit Hilfe der für jede Scheduling-Gruppe vorhandenen Variablen »cpu_power«. Während eine Zwei-Prozessor-SMP-Scheduling-Gruppe eine »cpu_power« von 2 besitzt, kann ein Hyperthreading-Prozessor nur mit 1,1 aufwarten.

Kommt der Scheduler zu dem Schluss, dass eine Prozessmigration sinnvoll ist, aktiviert er den höchstprioren Kernelthread »migration/X«, der für jede CPU einmal im System vorhanden ist und die eigentliche Migration durchführt. Allerdings werden nur jene Rechenprozesse migriert, die gerade nicht aktiv sind und die nicht Cache-hot sind, von denen der Scheduler also annimmt, dass sich im CPU-Cache keine für die Weiterverarbeitung nützliche Information befindet. Sind diese Anforderungen erfüllt, verhindert die CPU-Affinität unter Umständen noch eine Prozessverschiebung. Jeder Rechenprozess kann nämlich für sich bestimmen, auf welchem Prozessor er ablaufen soll und damit auch, auf welchem eben nicht.

Die Systemcalls »sched_set_affinity()« und »sched_get_affinity()« lesen respektive verändern die CPU-Affinität. Damit lassen sich CPU-Blocker auf eine CPU festlegen und die Cache-Performance optimieren. Außerdem können die zu einer Gruppe gehörigen Threads fest einer CPU zugeordnet werden. Auch das führt, da sich die Threads ihre Daten im Datensegment und damit im CPU-Cache teilen, unter Umständen zu einer Performance-Steigerung.

Isolationshaft für Realzeit

Besonders sinnvoll ist das Setzen von CPU-Affinitäten im Umfeld von Realzeit-Applikationen. Der Kernel bietet nämlich mit der Bootoption

```
isolcpus=<CPU-Nummer>,...,<CPU-Nummer>
```

die Möglichkeit, einen oder mehrere Prozessoren vom Ausbalancieren auszunehmen. Der einzige Weg, einen Rechenprozess auf eine isolierte CPU zu verschieben respektive von dieser wegzunehmen, ist der Systemcall »sched_set_affinity()«. Wer also eine isolierte CPU für Realzeit-Applikationen reserviert, kann mit wesentlich deterministischerem Zeitverhalten rechnen.

Der Applikationscode in Listing 1 zeigt, wie man die Affinität auslesen, setzen und damit auch eine Prozessmigration erzwingen kann. Eine Beschreibung der

Listing 1: »getcpu.c«

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #define __USE_GNU
05 #include <sched.h>
06 #include <errno.h>
07
08 static unsigned int getcpu()
09 {
10     unsigned int i, cpuid;
11     FILE *fp;
12
13     fp=fopen( "/proc/self/stat", "r" );
14     if( fp==NULL ) {
15         perror( "/proc/self/stat" );
16         exit( -1 );
17     }
18     fscanf( fp, "%*d %*s %*s" );
19     for( i=3; i<39; i++ ) // Das 39. Feld enthaelt die CPU-Id.
20         fscanf( fp, "%d", &cpuid );
21     return cpuid;
22 }
23
24 static unsigned int getaffinity( cpu_set_t *mask )
25 {
26     CPU_ZERO( mask );
27     if( sched_getaffinity( getpid(), sizeof(*mask), mask) ) {
28         perror("sched_setaffinity");
29         exit( -2 );
30     }
31     return mask->__bits[0]; // XXX - quick hack
32 }
33
34 int main( int argc, char **argv )
35 {
36     cpu_set_t mask;
37     int cpuid;
38
39     cpuid = getcpu();
40     printf("Aktive CPU: %d\n", cpuid );
41     printf("Affinity-Maske: 0x%x\n", getaffinity( &mask ) );
42     CPU_CLR( cpuid, &mask ); // Aktuelle CPU ausschliessen.
43     printf("Setzen der Affinität auf: 0x%lx\n", mask.__bits[0] );
44     if( sched_setaffinity( getpid(), sizeof(mask), &mask ) ) {
45         perror("sched_setaffinity"); // UP-System???
46     }
47     printf("Affinity-Maske: 0x%x\n", getaffinity( &mask ) );
48     printf("Aktive CPU %d\n", getcpu() );
49
50     return 0;
51 }
```

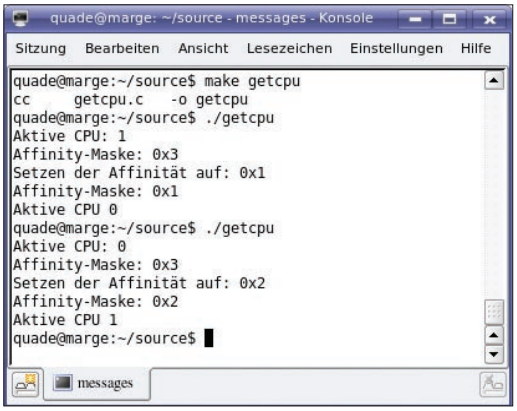


Abbildung 4: Mit Hilfe des Systemcalls »sched_setaffinity()« lässt sich eine Prozessmigration erzwingen.

Tabelle 1: Defaults für Multiprozessor-Architekturen

SMT	SMP	Numa
SD_SIBLING_INIT()	SD_CPU_INIT()	SD_NODE_INIT()
SD_BALANCE_NEWIDLE	SD_BALANCE_NEWIDLE	SD_BALANCE_FORK
SD_BALANCE_EXEC	SD_BALANCE_EXEC	SD_BALANCE_EXEC
SD_WAKE_IDLE	-	SD_WAKE_BALANCE
SD_WAKE_AFFINE	SD_WAKE_AFFINE	-
SD_SHARE_CPU_POWER	SD_SHARE_PKG_RESOURCES	SD_SERIALIZE

verwendeten Makros und Funktionen gibt [2]. Dieser Code gehört aber nicht in ein Produktivsystem. Der Einfachheit halber geht er nämlich einige Kompromisse ein: Er ist auf Zwei-Prozessor-Maschinen angepasst. Außerdem benutzt er nicht den mit Kernel 2.6.19 eingeführten Systemcall »get_cpu()«, der die CPU bestimmt, auf der der aktuelle Prozess läuft. Da der Systemcall neu ist, fehlt er in der Standard-C-Bibliothek.

Der universelle Funktionsaufruf »syscall()« hilft nur auf einer i386-Architektur weiter. Auf einer x86-64-Architektur ist »get_cpu()« als so genannter »vsyscall« realisiert, einem Systemcall, der allein im Userspace – ohne Übergang in die Kernelebene – abläuft. Vsyscalls setzen aber andere Aufrufmechanismen voraus. Der Beispielcode entnimmt deshalb die Information, auf welcher CPU der aktive Prozess läuft, der Proc-Datei »/proc/self/stat«. Die Information steht dort im 39. Feld, siehe Listing 1, Zeile 19.

Das Programm aus Listing 1 gibt die CPU aus, auf der es läuft. Danach erzwingt es eine Prozessmigration, indem es die aktuell genutzte CPU aus dem Bitfeld der affinen Prozessoren entfernt. Die erneute Ausgabe der CPU (Abbildung 4) bestätigt die Prozessverschiebung.

Handarbeit hilft

Bleibt abschließend anzumerken, dass vielfältige Methoden – die zudem von Kernelversion zu Kernelversion verfeinert werden – für eine bessere Lastverteilung zwischen den Prozessoren sorgen. Insbesondere wer hohe Anforderungen an das Zeitverhalten von Prozessen stellt, kann mit Hilfe der CPU-Affinität den Scheduler auf den richtigen Weg bringen. *(ofr)* ■

Infos

[1] Scheduling Domains und Load Balancing: [http://140.114.71.71/kerneltracing/slides/scheduling_domain.ppt]

[2] Dokumentation der GNU-C-Library: http://www.rdrs.net/document/c/gnuclibrary/html/CPU-Affinity.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Tabelle 2: Scheduler Domain Flags

Symbol	Bedeutung
SD_WAKE_IDLE	Dieses Flag ist dann relevant, wenn der Kernel einen schlafenden Rechenprozess aufweckt. Ist für die zugehörige Scheduling Domain das Flag gesetzt, verschiebt der Scheduler den aufgeweckten Prozess auf eine andere CPU innerhalb der Domain, wenn diese idle ist.
SD_WAKE_AFFINE	Kommt zur Wirkung, wenn ein schlafender Rechenprozess aufgeweckt wird. Wenn der geweckte Prozess cache-cold ist, verschiebt der Kernel ihn auf die den Scheduler-Code ausführende CPU, sonst verbleibt der Thread auf dem ihm bisher zugeordneten Prozessor.
SD_WAKE_BALANCE	Dieses Flag ist dann relevant, wenn ein schlafender Rechenprozess aufgeweckt wird. Der geweckte Rechenprozess wird dann auf die aktuelle, also den Scheduler-Code ausführende CPU verschoben, wenn diese idle ist.
SD_BALANCE_FORK	Das Flag wird bedeutsam, wenn ein Task den Systemcall »fork« aufruft. In diesem Fall balanciert der Scheduler die Rechenprozesse zwischen den nächsten Domains aus, die das Flag gesetzt haben.
SD_BALANCE_EXEC	Beeinflusst den Scheduler, wenn ein Task den Systemcall »exec« aufruft. Der Scheduler sucht die höchste Ebene im Domain-Baum, die dieses Flag aktiviert hat. Danach sucht er innerhalb dieser Domain die CPU, die am wenigsten Last aufweist, und ordnet den Prozess dieser CPU zu.
SD_BALANCE_NEWIDLE	Das Flag ist relevant, wenn eine Run Queue leer ist (der Prozessor ist idle). Wird eine Run Queue leer, sucht der Scheduler die oberste Domain, die dieses Flag gesetzt hat. Innerhalb der Domain wird die CPU gesucht, die die meiste Last fährt. Der Scheduler verschiebt einen Rechenprozess von dieser CPU auf die Idle-CPU (Heranholen von Rechenprozessen).
SD_SHARE_CPUPOWER	Dieses Flag ist typischerweise auf der SMT-Ebene gesetzt. Ist die aktive CPU idle, findet eine Prozessmigration statt. Eine Besonderheit: Verarbeitet der andere Kern des Hyperthreading-Prozessors einen Rechenprozess mit höherer Priorität, verzögert der Scheduler die Bearbeitung des Rechenprozesses mit niedrigerer Priorität. Damit steht die gesamte Rechenleistung uneingeschränkt dem Realzeit-Prozess zur Verfügung.
SD_SHARE_PKG_RESOURCES	Typischerweise auf der SMP-Ebene gesetzt. Es dient allein dazu, die Variable »cpu_power« zu initialisieren.
SD_SERIALIZE	Kommt normalerweise auf der Numa-Ebene zum Einsatz. Ist es gesetzt, balanciert der Kernel zu einem Zeitpunkt immer nur eine Domain per Zeitsteuerung (Soft-IRQ) aus.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 34

Kern-Technik

Spinlocks, die nicht spinnen, und Interrupt-Service-Routinen, die im Prozesskontext laufen: Das Realtime-Preemption-Patch stellt die Welt des Kernels scheinbar auf den Kopf. Eva-Katharina Kunst, Jürgen Quade



Virtualisierung auf der einen und Echtzeitverhalten auf der anderen Seite. Das sind die beiden großen Entwicklungsfelder am Linux-Kernel, die zurzeit spürbare Verbesserungen für den Nutzer mit sich bringen. Mit Kernel 2.6.21 hat Linus Torvalds nicht nur das in der Kern-Technik-Folge 31 (siehe **[1]**) beschriebene Dyntick-Patch übernommen, sondern gleich das dort in Aussicht gestellte Tickless-Patch mit dazu.

Doch damit nicht genug. In der Queue der Entwickler Thomas Gleixner und Ingo Molnar stehen weitere Änderungen, die Stück für Stück den Weg in den Mainstream-Kernel finden sollen. Entwickler von Systemen mit hohen Anforderungen an das Zeitverhalten nutzen das entwickelte Patchset »PREEMPT_RT« (**[2]**, **[3]**, **[4]**, **[5]**) bereits seit einiger Zeit als Latenzzeit-Killer und um das Timing berechenbarer zu gestalten. Der Trick: Nicht unterbrechbare Bereiche werden unterbrechbar. Die nicht unterbrechbaren Bereiche existieren in mehreren Formen: als Interrupt-Service-Routinen, als Soft-IRQs und als durch Spinlocks geschützte Codesequenzen.

Alles ein Thread

Die Preempt-Modifikationen stellen damit das bisherige Unterbrechungsverhalten auf den Kopf, mit weitreichenden Konsequenzen für die Kernelprogrammierer. Denn mit einem Mal scheint der Kernel Interrupt-Service-Routinen zu schedulen und Soft-IRQs schlafen zu legen.

Doch das ist in Wahrheit nur babylonische Sprachverwirrung. Denn technisch gesehen stellt der Linux-Kernel weiterhin die vier Unterbrechungsebenen zur Verfügung, wie sie im **Kasten „Unterbrechungsmodell“** dargestellt sind. Nur dass die Entwickler den Code aus den Interrupt-Service-Routinen in hoch priorisierte Threads, die ISR-Threads, verschoben haben. Der zugehörige Code wird damit nicht mehr auf der Interrupt-, sondern auf der Kernel-Ebene abgearbeitet und der dort ablaufende Code ist bekanntlich unterbrechbar. Die richtige ISR hat nur noch die Aufgabe, den zugehörigen ISR-Thread zu aktivieren.

Genauso sieht es mit den Soft-IRQs aus. Den hierzu gehörenden Code haben die Kernelentwickler ebenfalls in eigene Threads verlagert (Soft-IRQ-Threads), die der Scheduler aufgrund ihrer hohen Priorität allerdings bevorzugt zur Abarbeitung auswählt. Im Fall des Falles unterbrechen aber wichtigere Rechenprozesse einen Soft-IRQ-Thread. In Anlehnung an das alte Unix-Motto „Alles ist eine Datei“ heißt es jetzt also zusätzlich „Alles ist ein Thread“.

Auch den Spinlocks rücken die Entwickler zu Leibe. Diese führen unter Umständen zu einer Interrupt-Sperre, sodass hoch priore Rechenprozesse mit engen zeitlichen Anforderungen bisher chancenlos waren. Wenn aber alles ein Thread ist und die zugehörigen Codesequenzen, beispielsweise die ehemaligen Interrupt-Service-Routinen, schlafen können, dann bleibt für Spinlocks nichts weiter als ein kleiner Performance-Vorteil (kein aufwändiger Prozesswechsel bei einem kurzen, gesperrten kritischen Abschnitt) übrig. Den tauschen die Entwickler gerne gegen verkürzte Latenzzeiten ein und verwandeln konsequenterweise Spinlocks im RT_PREEMPT-Patch in Mutexe.

Diese tiefgreifende Modifikation in der Basistechnologie (Spinlock zu Mutex – aktives zu passivem Warten) ist für den Programmierer unsichtbar. Der Compiler tauscht dank ausgeklügelter Makros die Namen der Datentypen und der verwendeten Funktionen gegen Mutex-Typen und -Funktionen aus. So wird aus dem Datentyp »spinlock_t« ein »struct rt_mutex«, und die Funktionen »spin_lock()« und »spin_unlock()« firmieren danach unter »rt_mutex_lock()« und »rt_mutex_unlock()«.

Namensverwirrung

Dass das Kind nicht mehr den richtigen Namen trägt, hat wohl damit zu tun, dass Linus Torvalds Veränderungen am Kernel bekanntlich nur in kleinen Häppchen akzeptiert. Der Austausch der Datentypen und Funktionsnamen wäre zweifelsohne ein dicker Brocken. Dennoch: Wo »rt-mutex« drin ist, sollte auch

»rt-mutex« dranstehen! Alles andere führt über kurz oder lang nur zu Missverständnissen und zu unvorhersehbaren Fehlern.

Die Umwandlung von Mutexen zu Spinlocks führt auf Mehrprozessorsystemen zu einem Nebeneffekt, auf den der Programmierer achtgeben sollte: Während ein Spinlock sicherstellt, dass eine zusammenhängende Codesequenz komplett auf einer CPU abläuft, können beim Einsatz eines Mutex mehrere Prozessoren an der Abarbeitung des Programmabschnitts beteiligt sein. Wer Per-CPU-Variablen in seinem Code einsetzt, sollte also gut aufpassen. Nur wer die Preemption über die in [6] erläuterten Funktionen ausschaltet, schließt zusätzliche Race Conditions aus.

Auch wenn normale Kernelentwickler und Treiberprogrammierer wohl kaum einen Einsatzbereich dafür haben, lässt das PREEMPT_RT-Patch auch weiterhin die klassischen Interrupt-Service-Routinen zu. Um das alte Verhalten zu aktivieren, setzen Programmierer beim Aufruf von »request_irq()« das Flag »IRQF_NODELAY«. Damit läuft die ISR direkt beim Auftreten des Interrupts im Interrupt-Kontext ab.

Sind in einem solchen Fall kritische Abschnitte zu schützen, muss das gute alte Spinlock wieder herhalten, das jetzt unter dem Datentyp Raw-Spinlock (»raw_spinlock_t«) firmiert. In Verbindung mit den bekannten Spinlock-Funktionen werden beim Kompilieren daraus durch die erwähnten Makros echte Spinlocks.

Doch noch einmal zurück zum Realtime-Mutex, das nämlich noch mit einer weiteren Eigenschaft aufwarten kann: Es kennt und unterstützt die so genannte Prioritätsvererbung.

Prioritäts-Inversion und -Vererbung

Als Prioritätsinversion wird eine Situation bezeichnet, in der ein Job mit mittlerer Priorität die Abarbeitung eines Jobs mit hoher Priorität verzögert (siehe Abbildung 2a). Der hoch priore Job (A) kommt typischerweise deshalb nicht zum Zug, weil er einen durch Semaphor oder Mutex geschützten kritischen Abschnitt betreten möchte, den ein Rechenprozess mit niedriger Priorität (C) gerade

Unterbrechungsmodell

Codesequenzen im Linux-Kernel können auf vier unterschiedlichen Ebenen ablaufen (siehe Abbildung 1). Die Abarbeitungsebene entscheidet darüber, welche Funktionen des Kernels zum Einsatz kommen und welche (anderen) Codesequenzen sie unterbrechen dürfen. Die sich durch die Unterbrechung ergebenden kritischen Abschnitte sind durch entsprechende Methoden zu schützen.

Grundsätzlich gilt, dass Code, der auf einer niedrigeren Ebene lauffähig wird, höhere Priorität hat (vertikale Unterbrechbarkeit). Die Abarbeitung des Code auf einer darüberliegenden Ebene wird also zugunsten des lauffähig gewordenen Code unterbrochen. Eine Interrupt Service Routine (ISR) unterbricht einen Systemcall, der auf Kernel-Ebene arbeitet.

Die horizontale Preemption spezifiziert, ob Codesequenzen, die auf der gleichen Ebene ablaufen, sich gegenseitig unterbrechen können. Auf der ISR- und der Soft-IRQ-Ebene gibt es bei Linux typischerweise keine horizontale Unterbrechung. Werden mehrere konkurrierende Codesequenzen auf der Interrupt-Ebene oder der Soft-IRQ-Ebene lauffähig, findet konsequenterweise eine Sequenzialisierung im Sinne eines First Come First Serve (FCFS) statt: Erst wenn die gerade aktive Codesequenz beendet ist, arbeitet der Kernel die konkurrierende Codesequenz ab.

Auf der Kernel-Ebene hat Kernel 2.6 die Kernel-Preemption eingeführt. Dort abzuarbeitende Codesequenzen – Kernel-Threads und Systemcalls – sind priorisiert. Höher priore Jobs unterbrechen demnach niedrig priore Prozesse. Auf der Applikations-Ebene (Ebene 4) schließlich unterbrechen nicht nur die höher prioren Jobs die niedrig prioren, die gleich prioren Rechenprozesse geben nach dem Round-Robin-Prinzip (Zeitscheiben) die CPU nach einer endlichen Verarbeitungszeit ab. Diese Priorisierung arbeitet übrigens Ebenen-übergreifend, sodass ein hoch priorer Job auf Applikationsebene einen weniger wichtigen Job auf der Kernel-Ebene unterbrechen darf.

Lock-Typen

Entwickler müssen das Unterbrechungsmodell kennen, damit sie kritische Abschnitte korrekt schützen können. Der Kernel bietet hierzu diverse Methoden an, die den konkurrierenden Zugriff auf ein Betriebsmittel innerhalb einer Ebene, aber auch von unterschiedlichen Unterbrechungsebenen heraus absichern. Die bekanntesten Methoden sind das Semaphor, das im Linux-Kernel im Wesentlichen in der Variante Mutex eingesetzt wird, und schließlich das Spinlock. Auf Einprozessormaschinen lassen sich Spinlocks allerdings nicht einsetzen. Innerhalb des Kernels setzen Programmierer hier alternativ die Interruptsperre ein. Sie mussten sich um dieses Detail bisher nicht kümmern, das Kernel-Buildsystem hat für sie jeweils den richtigen Code eingesetzt.

Jedoch mussten sie sehr wohl bedenken, auf welcher Ebene die an einem zu schützenden kritischen Abschnitt beteiligten Codesequenzen ablaufen. Schützen beispielsweise eine Interrupt-Service-Routine und Code auf Kernel-Ebene einen kritischen Abschnitt, mussten die Programmierer das Spinlock mit einer Interrupt-Sperre kombinieren (»spin_lock_irq()«). Das einfache Spinlock, repräsentiert durch die Funktion »spin_lock()«, kann den gegenseitigen Ausschluss, also die sequenzielle Abarbeitung, nur dann garantieren, wenn die konkurrierenden Codesequenzen auf unterschiedlichen Prozessoren laufen. Die zusätzliche Interruptsperre schließt damit die Unterbrechung des Code auf Kernel-Ebene durch eine ISR aus, die auf dem gleichen Prozessor abläuft.

Übrigens: Man sollte sich bei der Wahl des Schutzmechanismus eines kritischen Abschnitts nicht allein vom Unterbrechungsmodell und damit von einer scheinbaren Parallelverarbeitung leiten lassen. Mehrprozessorsysteme, die eine echte Parallelverarbeitung bieten, sind heute Standard. Besser ist es also, grundsätzlich davon auszugehen, dass alle Routinen mehrfach parallel ablaufen.

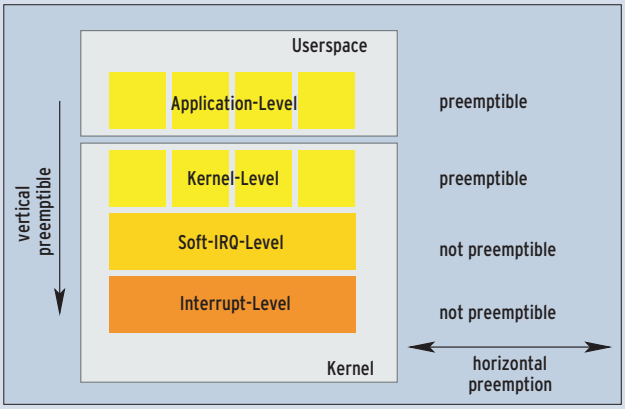


Abbildung 1: Das Unterbrechungsmodell des Kernels: Die Unterbrechbarkeit setzt sich aus horizontaler und vertikaler Preemption zusammen.

bearbeitet. Der an sich unbeteiligte Prozess mit mittlerer Priorität (B) hält dabei den niedrig prioren und somit auch den hoch prioren Job auf.

Zur Entschärfung der Prioritätsinversion setzen die Entwickler auf die so genannte Prioritätsvererbung. Sobald der hoch priore Job (C) die Semaphor- oder Mutex-Operation zum Betreten eines kritischen Abschnitts aufruft und das Semaphor/Mutex durch den niedrig prioren Rechenprozess (C) belegt ist, bekommt dieser die Priorität des hoch prioren Jobs vererbt (siehe Abbildung 2b). So ausgestattet lässt er sich nicht mehr durch den mittel priore Job (B) verdrängen. Gibt er das Semaphor/Mutex frei, nimmt er wieder seine ursprüngliche Priorität an und der hoch priore Job betritt – mit einer signifikant kürzeren Latenzzeit – den kritischen Abschnitt.

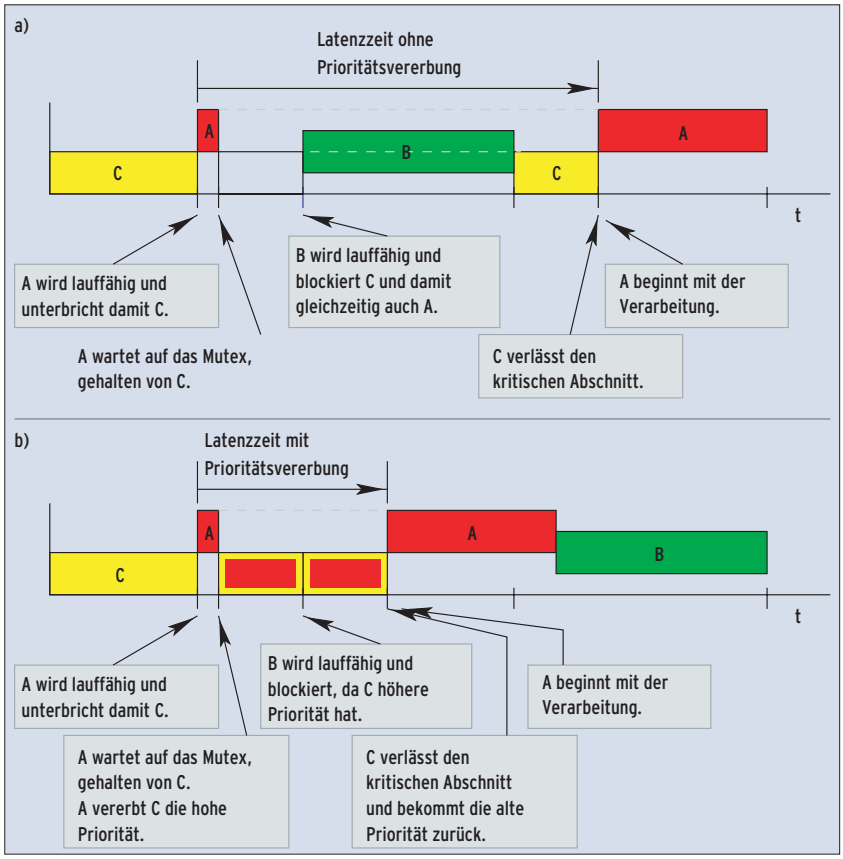


Abbildung 2: Prioritätsinversion und -vererbung: Belegung der CPU durch die drei Jobs A (rot, hohe Priorität), B (grün, mittlere Priorität) und C (gelb, niedrige Priorität).

Praxis

Um in den Genuss der verkürzten Latenzzeiten zu kommen, benötigt der Entwickler die Quellen eines aktuellen Linux-Kernels (zum Beispiel 2.6.21) und das passende PREEMPT_RT-Patch. Der Quellcode zum Linux-Kernel findet sich auf [http://www.kernel.org], das passende PREEMPT_RT-Patch (zum Beispiel »patch-2.6.21-rt1«) unter [7]. Nach dem Entpacken des Kernels im Verzeichnis »/usr/src/«, geht es ans Patchen und Konfigurieren. Auf dem Ubuntu-System der Autoren übernehmen das die Kommandos in Listing 1.

Bei der Kernelkonfiguration muss die Option »Complete Preemption (Real-Time)« aktiviert sein (siehe Abbildung 3). Alternativ lässt sich mit der Option »Preemptible Kernel (Low-Latency Desktop)« die Verschiebung der ISRs und der Soft-IRQs in Threads getrennt per SYS-Filesystem (Dateien »/proc/sys/kernel/softirq_preemption« und »/proc/sys/kernel/hardirq_preemption«) zur Laufzeit des Kernels kontrollieren. Bleibt nur noch, den neuen Kernel zu übersetzen und zu installieren. Unter Ubuntu geht das folgendermaßen:

```
/usr/src/linux-2.6.21# make-kpkg --initrd➥
 buildpackage
/usr/src/linux-2.6.21# cd ..
/usr/src# dpkg -i linux-image-2.6.21-rt1_➥
2.6.21-rt1-10.00.Custom_i386.deb
```

Mit einem folgenden Reboot ist der Latenzzeit-Killer schließlich aktiviert. Das zeigt sich bei einem Blick auf die Taskliste (siehe Abbildung 4). Hier tauchen die Interrupt-Service-Threads ebenso auf wie die Soft-IRQ-Threads. Da es sich nur bei den Prioritäten von 40 bis 139 um Echtzeitprioritäten handelt, laufen diese Threads augenscheinlich mit der mittleren Priorität 90.

Über Prioritäten legt der Anwender fest, dass der Scheduler Rechenprozesse mit hohen Anforderungen an eine kurze Latenzzeit bevorzugt. Das Kommando »chrt« kann die Prioritäten einzelner Threads gezielt anpassen. Unglücklicherweise stimmen aber die mit »ps« angezeigten Prioritäten nicht genau mit denen überein, die das Kommando »chrt« erwartet. Es setzt nämlich die Prioritäten getrennt einmal für den Bereich der normalen und einmal für den Bereich der Realzeit-Prioritäten. Jeder Bereich beginnt wiederum mit Priorität 0. Priorität 90 bei »ps« entspricht Priorität 50 und Scheduling Mode »SCHED_FIFO« bei »chrt«.

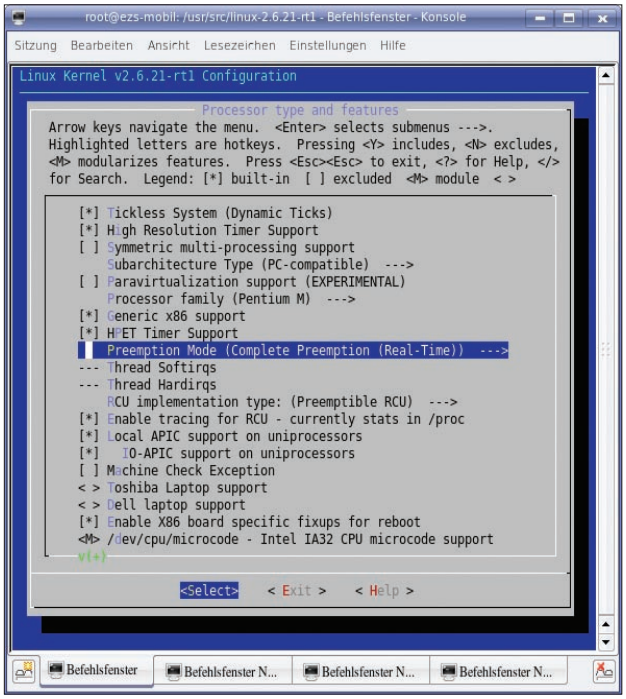
Wer die Priorität des in Abbildung 4 gezeigten Prozesses »IRQ-10« auf 91 anheben will, gibt diesen Befehl ein:

```
sudo chrt -vfp 51 1998
```

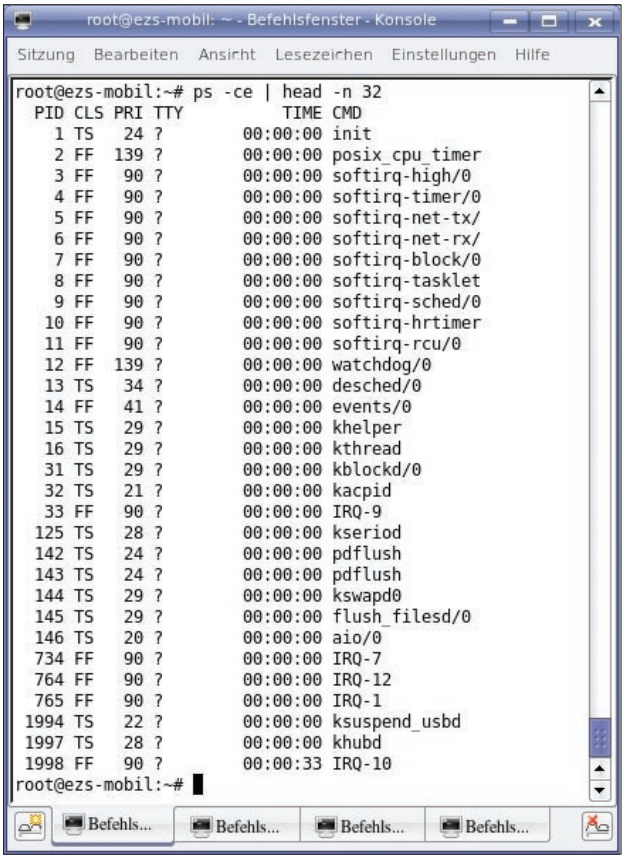
Das RT_PREEMPT-Patch bringt Linux einem richtigen Echtzeit-Betriebssystem einen erheblichen Schritt näher. Für viele Anwendungen sind damit die bisherigen Realtime-Erweiterungen wie RT-Linux und RTAI nicht mehr notwendig. Da auch auf dem Desktop dank Multimedia die Anforderungen an das Echtzeitverhalten zunehmend strenger werden und die Performance-Penalties des vorgestellten Patchsets sehr gering sind,

Listing 1: Realtime-Patch

```
01 /usr/src# wget http://www.kernel.org/pub/linux/
   kernel/v2.6/linux-2.6.21.tar.bz2
02 /usr/src# wget http://people.redhat.com/mingo/
   realtime-preempt/patch-2.6.21-rt1
03 /usr/src# tar xvfj linux-2.6.21.tar.bz2
04 /usr/src# cd linux-2.6.21
05 /usr/src/linux-2.6.21# patch -p1 <../patch-2.6.21-rt1
06 /usr/src/linux-2.6.21# cp /boot/config-`uname -r` .
07 /usr/src/linux-2.6.21# make menuconfig
```



▲ **Abbildung 3: Die richtige Kernelkonfiguration entscheidet über die Echtzeit-Eigenschaften. Der relevante Eintrag heißt »Complete Preemption«.**



▶ **Abbildung 4: Das Kommando »ps« verrät: ISRs und Soft-IRQs tauchen in der Taskliste auf. Die Spalte »PRI« zeigt die Priorität jedes Prozesses.**

spricht nicht viel gegen eine Aufnahme in den Mainline-Kernel. Linus Torvalds sollte allerdings für eine korrekte Namensgebung der verwendeten Synchronisationselemente sorgen – selbst wenn dies eine Textersetzung in vielen Codepassagen bedeuten würde. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik" (31): Linux-Magazin 1/07, S. 110

[2] Initiale Ankündigung der RT-Preemption auf der Kernel-Mailingliste: [http://kerneltrap.org/node/3995]

[3] Linux-Realtime-Wiki: [http://rt.wiki.kernel.org/index.php/Main_Page]

[4] Paul McKenney, „A realtime preemption overview": [http://lwn.net/Articles/146861/]

[5] Paul McKenney, „SMP and embedded real time systems": [http://www.linuxjournal.com/article/9361]

[6] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik" (26): Linux-Magazin 2/06, S. 102

[7] Realtime Preemption Patch: [http://people.redhat.com/mingo/realtime-preempt/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Anzeigen

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 35

Kern-Technik

Kopieren, ohne die CPU damit zu belasten? Linux kann's. Fehlen nur noch Applikationen, die das neue Feature nutzen. Die Kern-Technik verschafft Anwendungsprogrammierern das Know-how. Eva-Katharina Kunst, Jürgen Quade



Die Datenmengen bei Multimedia & Co. sind für jedes Betriebssystem eine mächtige Herausforderung. Oft müssen mal eben ein paar GByte von der Festplatte gelesen, in die Grafikkarte transportiert oder über das Netzwerkinterface versandt werden. Typischerweise kopiert das OS die Daten dazu gleich mehrfach hin und her (Abbildung 1), was den ganzen Vorgang selbstverständlich verlangsamt.

Zum Beispiel bei einem Server, der eine Filmdatei an einen über TCP/IP angebundenen Client sendet: Zuerst wandern die Daten (also der Film) von der Festplatte in den Speicher des Betriebssystemkerns, den Kernelspace. Dann sind die Daten ein zweites Mal unterwegs, um in den Speicherbereich der Server-Applikation zu wandern, den Userspace.

Die Applikation schließlich übergibt die Daten wieder dem Kernel, damit er sie über das Netzwerkinterface an den Client-Rechner schickt. Dazu kopiert der Netzwerktreiber die im Kernel deponierten Daten in den Speicher des Ethernet-Controllers.

Zum Glück erfolgen zumindest zwei der Kopieraktionen häufig per DMA, nämlich das Lesen von der Platte und die Übergabe an den Ethernet-Controller. Das lindert das Problem aber nur zum Teil. Mindestens zweimal fasst die CPU die Daten an – das können sich auch moderne (HPC, High Performance Computing) Systeme nicht leisten.

Schon seit Längerem verwenden pfiffige Programmierer daher den Systemcall »sendfile()« [1]. Dieser Systemaufruf kopiert Daten von einem Stream in den nächsten, ohne sie dabei im Userspace zwischenzuparken. Leider ist »sendfile()« nicht universell verwendbar. Es lässt nämlich – vereinfacht gesagt – nur das Kopieren von einer auf der Platte liegenden Datei auf einen Socket (Netzwerk) zu. Datei auf Datei, Socket auf Socket oder Socket auf Datei – für »sendfile()« ist das alles unmöglich.

Turbolader

Seit Kernel 2.6.17 tragen die Entwickler den steten Nachfragen nach optimierten Datenkopierfunktionen mit den neuen Systemcalls »splice()«, »tee()« und »vmsplice()« Rechnung. Sie implementieren so genanntes Zero-Copying, was bedeutet, dass die CPU beim Kopieren von Daten nicht mehr beteiligt ist, eine CPU-Last also durch das Kopieren nicht anfällt (Abbildung 2).

Der Trick besteht darin, die Daten – meist per DMA – von der Quelle (etwa der Festplatte) in einen Kernel-Buffer zu transportieren. Statt die Daten danach

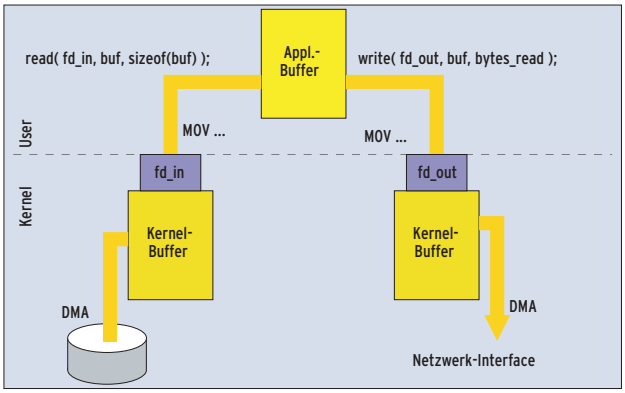


Abbildung 1: Beim traditionellen Kopieren wird eine Datei bis zu viermal im Speicher kopiert. Unter günstigen Umständen kommt zweimal direkter Speicherzugriff (DMA) zum Einsatz, aber mindestens zweimal durchlaufen die Daten die CPU.

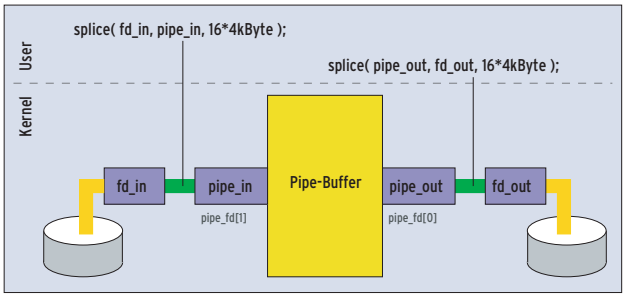
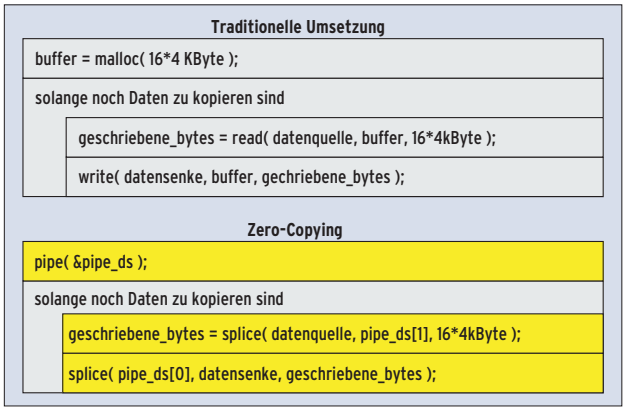


Abbildung 2: Disk-to-Disk-Copy: Der erste Aufruf von »splice()« füllt den Kernel-Buffer, der zweite Aufruf überführt die Daten in die neue Datei.



◀ Abbildung 3: Statt des Puffers im Userspace legt der Kernel beim Zero-Copying per »pipe()«-Aufruf einen eigenen Puffer an. Der Syscall »splice()« stößt aus dem Userspace das Lesen von Platte in den Kernel-Buffer und das Schreiben aus dem Kernel-Buffer auf die Platte an.

weiter in den Userspace und von dort zurück an eine andere Stelle im Kernel zu kopieren, entnimmt der Kernel (etwa der Netzwerktreiber) die Daten direkt dem Kernel-Buffer. Gesteuert wird dieser Vorgang, bei dem die CPU kein einziges Datenbyte anfasst, durch die User-Applikation (siehe Abbildung 2).

Alles, was »splice()« & Co. also brauchen, ist – neben der Eingangsdatenquelle und der Ausgangsdatensenke – ein Kernel-Buffer. Den stellt eine klassische Unix-Pipe zur Verfügung, weshalb sie auch der Dreh- und Angelpunkt der Kernel-Implementierung ist. Der Systemcall »splice()« verbindet einen Filedeskriptor mit dem Anfang (oder Ende) einer solchen Pipe und füllt respektive entleert den zugehörigen Kernel-Buffer. Insgesamt sind bei einer derartigen Kopieraktion also vier Deskriptoren beteiligt (Abbildung 3).

Pipes und Puffer

Wer beispielsweise mit Hilfe von Zero-Copying eine Datei in eine andere kopieren möchte, verbindet mit »splice()« den File-Deskriptor der Quelldatei mit dem einen Ende der Pipe und füllt den Pipe-Buffer. Ein zweiter Aufruf von »splice()« verbindet das andere Ende der Pipe mit der Ausgabedatei und transferiert dadurch den Inhalt des Pipe-Buffers zurück auf die Festplatte. Diese Sequenz wiederholt sich so lange, bis alle Daten kopiert sind. Abbildung 3 verdeutlicht den Unterschied zur traditionellen Kopierfunktion: An die Stelle des Puffers im Userspace tritt der per Aufruf »pipe()« reservierte Kernel-Buffer. Und statt »read()« und »write()« transferiert der neue Systemcall »splice()« die Daten.

Um mehreren Programmen parallel Zugriff auf die im Kernel-Buffer befindlichen Daten zu ermöglichen, bietet Linux den Systemcall »tee()« an. In seiner Funktion ähnelt er dem von der Shell bekannten Tee-Kommando. Er erlaubt es, Daten aus dem Pipe-Buffer auszulesen und an eine (andere) Pipe weiterzureichen (siehe Abbildung 4). Damit lassen sich beispielsweise die von einem DVB-T-Gerät empfangenen Daten per »tee« gleichzeitig mit einen Videoplayer anschauen und auf Festplatte speichern.

Listing 1: »splice()«

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <fcntl.h>
04 #include <sys/stat.h>
05 #include <asm/page.h>
06 #include <asm/unistd.h>
07
08 // defines from linux/pipe_fs_i.h
09 #define SPLICE_F_MOVE     (0x01)
   /* move pages instead of copying */
10 #define SPLICE_F_MORE     (0x04)
   /* expect more data */
11 #define PIPE_BUFFERS (16)
12
13 #define SPLICE_BUF_SIZE (PIPE_BUFFERS*PAGE_
   SIZE)
14
15 static inline int sys_splice(int fdin, loff_t
   *off_in,
16     int fdout, loff_t *off_out, size_t len,
   unsigned int flags)
17 {
18     return syscall( __NR_splice,fdin,off_in,
   fdout,off_out,len,flags );
19 }
20
21 int main( int argc, char **argv )
22 {
23     int fd_in, fd_out, ret;
24     int fd_pipe[2];
25     int bytes_to_copy, bytes_copied;
26     unsigned long bytes_left;
27     struct stat stat_buf;
28
29     if( argc != 3 ) {
30         fprintf(stderr,"usage: %s from to\n",
   argv[0] );
31         return -1;
32     }
33     fd_in = open( argv[1], O_RDONLY );
34     if( fd_in < 0 ) {
35         perror( argv[1] );
36         return -1;
37     }
38     fd_out = open( argv[2], O_WRONLY|O_CREAT,
   0644 );
39     if( fd_out < 0 ) {
40         perror( argv[2] );
41         return -1;
42     }
43     if( pipe(fd_pipe) < 0 ) {
44         close( fd_out );
45         close( fd_in );
46         return -1;
47     }
48     fstat( fd_in, &stat_buf );
49     bytes_left = stat_buf.st_size;
50
51     while( bytes_left ) {
52         bytes_to_copy = bytes_left <
   SPLICE_BUF_SIZE ?
53             bytes_left:SPLICE_BUF_SIZE;
54         bytes_copied=sys_splice( fd_in, NULL,
   fd_pipe[1],NULL,
55             bytes_to_copy, SPLICE_F_
   MOVE|SPLICE_F_MORE );
56         if( bytes_copied==-1 ) {
57             perror( "splice read" );
58             return -1;
59         }
60         bytes_left -= bytes_copied;
61         while( bytes_copied ) {
62             ret=sys_splice( fd_pipe[0], NULL,
   fd_out, NULL,
63                 bytes_copied, SPLICE_F_
   MOVE|SPLICE_F_MORE );
64             if( ret==-1 ) {
65                 perror( "splice write" );
66                 return -1;
67             }
68             bytes_copied -= ret;
69         }
70     }
71     close( fd_out );
72     close( fd_in );
73     close( fd_pipe[0] );
74     close( fd_pipe[1] );
75     return 0;
76 }
```

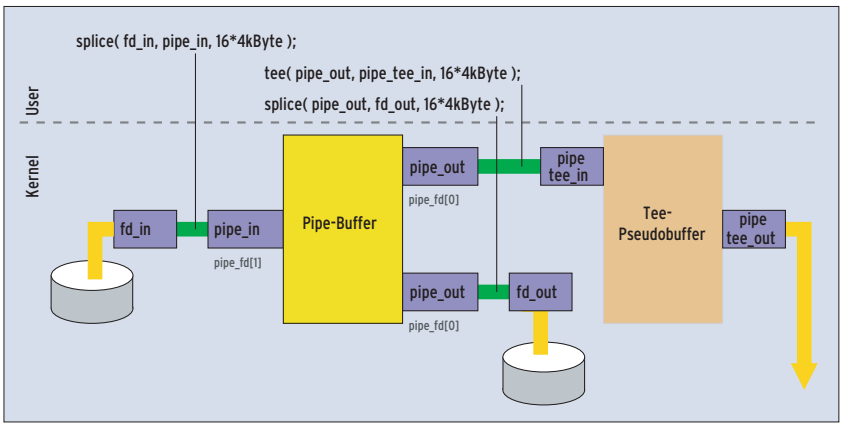


Abbildung 4: Der Systemcall »tee()« bindet noch einen zweiten Kernel-Buffer zwischen die Eingabequelle und den duplizierten Ausgabekanal. Doch keine Angst vor unnötigen Datenduplikaten: Der Kernel arbeitet intern nicht mit einer Kopie, sondern nur mit einer Referenz.

»tee()« kopiert die Daten dabei nicht, sondern arbeitet allein mit Referenzen. Häufig ist die Pipe, an die »tee()« die Daten weiterreicht, die Standardausgabe Stdout (»STDOUT_FILENO«).
Auf der Kommandozeile sieht eine solche Befehls-Pipeline so aus:

```
splice-in /dev/dvbt | splice-tee ➥
film.mpeg | splice-out /dev/tvout
```

Doch sollten Programmierer aufpassen: Wer von einer Quelle zwei Kopien ziehen möchte – wobei keine der Kopien nach Stdout geht –, braucht auch zwei Pipes! Der Pseudo-Code in Abbildung 5 gibt die Struktur einer solchen Kopierapplikation wieder.
Der dritte hier vorgestellte Systemcall »vmsplice()« macht die im Userspace bereits vorhandenen Speicherbereiche dem Kernel zugänglich. Ohne Kopieraktion der Daten vom User- in den Kernelspace überträgt der Kernel die ihm anvertrauten Daten zur Peripherie per DMA und umgekehrt. Es handelt sich damit – abgesehen von der mit dem Systemcall verbundenen Kopieraktion – gewissermaßen um ein Gegenstück zu »mmap()« [3], das Speicherbereiche im Kernelspace einer Applikation zugänglich macht. Der zugehörige Filedeskriptor ist wieder einmal eine Pipe. Die genaue Aufrufsyntax der Systemcalls und der zugehörigen Parameter beschreibt der Kasten „Neue Systemcalls im Überblick“.

Implementationsprobleme

Nicht alles ist Gold, was glänzt. Leider weist die Implementierung von Splice sowohl im Kernel als auch im Userland noch Schwächen auf. Im Userland ist es die Glibc, die auf verschiedenen Plattformen (zum Beispiel auf Ubuntu und Suse) die Systemcalls »splice()«, »tee()« und »vmsplice()« nicht richtig implementiert. Hier bleibt nur, die Systemcalls – wie in [4] beschrieben oder in Listing 1 (Zeile 18) gezeigt – direkt aufzurufen.

Warten und hoffen

Im Kernel sieht es leider noch trauriger aus, hier müssen Performance-hungrige Programmierer auf Linux 2.6.23 warten. Erst dann ist der Code für »SPLICE_F_MOVE« (siehe Kasten „Neue Systemcalls im Überblick“) implementiert.
In der aktuellen Version 2.6.21 kopiert der Kernel bei Splice-Zugriffen auf Dateien nämlich die Daten der Pipe-Buffer in den Page-Cache, bis sie von dort den Weg auf die Festplatte finden. In Version 2.6.23 – so die Planung – wird der fragliche Pipe-Buffer, wenn »SPLICE_F_MOVE« gesetzt ist, zum Page-Cache hinzugefügt und dann nur noch mit Referenzen gearbeitet.

Ins Netz kopieren

Aber immerhin: Die Splice-Kopplung zum Netzwerk-Stack ist auf einem guten Weg. Vom eingangs erwähnten Veteranen »sendfile()« ist nur noch die Schnittstelle übrig geblieben. Hinter den Kulissen ist »sendfile()« auch längst auf der Basis von »splice()« realisiert. Denn »splice()« bringt nicht nur wegen der Universalität Vorteile, auch das Programmierinterface ist klarer.
Will ein Programmierer beispielsweise den Daten auf der Platte vor dem Versen-

Neue Systemcalls im Überblick

Drei Systemaufrufe geben dem Anwendungsprogrammierer eine Eintrittskarte zu den neuen Kernelfunktionen.
Der erste Systemcall verbindet den über die Pipe repräsentierten Kernel-Buffer mit einem File-, Pipe- oder Socket-Deskriptor:

```
int splice(int fdin, loff_t *off_in, ➥
int fdout, loff_t *off_out, size_t len,➥
unsigned int flags)
```

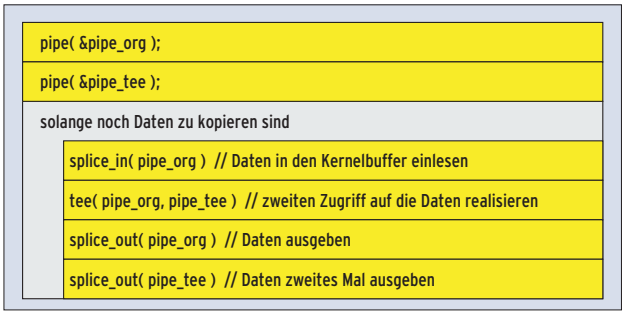
Er überträgt von »fdin« ab dem »off_in«-Byte maximal »len« Bytes Daten nach »fdout«. Über das Bitfeld »flags« lässt sich der Transfer parametrisieren. Die Bedeutung der Flags sieht folgendermaßen aus:
»SPLICE_F_MOVE«: Ist dieses Flag gesetzt, versucht der Kernel anstelle einer Kopie den Pipe-Buffer in den Page-Cache zu verschieben, falls er die Daten auf die Festplatte transferieren soll.
»SPLICE_F_MORE«: Ist dieses Flag gesetzt, weiß den Kernel, dass nach dem Aufruf von »splice()« weitere Daten folgen.
»SPLICE_F_NONBLOCK«: Dieses Flag verhindert, dass die Applikation am »splice()«-Aufruf schlafen gelegt wird. Achtung: Ein (interner) Zugriff auf die File-Deskriptoren kann dennoch blockieren.
»SPLICE_F_GIFT«: Das Flag ist nur im Kontext von »vmsplice« interessant. Damit überlässt eine Applikation dem Kernel die Kontrolle über die übergebenen Speicherbereiche, die Applikation darf den Speicher nicht weiter verwenden. Der übergebene Speicher muss einer Seitenlänge (typisch 4 KByte) entsprechen und auf eine Seitenadresse ausgerichtet sein.
Der zweite Systemcall dupliziert mindestens »len« Bytes von »pipe_in« nach »pipe_out«:

```
int tee(int pipe_in, int pipe_out, size_t ➥
len, unsigned int flags)
```

Der dritte Systemcall schließlich blendet »nr_segs« Speicherbereiche (referenziert über »iov«) in den durch »fd« repräsentierten Pipe-Buffer ein:

```
int vmsplice(int fd, const struct iovec ➥
*iov, unsigned long nr_segs, unsigned ➥
int flags)
```

Die Struktur »struct iovec *« ist in der Headerdatei »sys/uio.h« deklariert.



◀ **Abbildung 5: Der vorgestellte Pseudo-Code legt zwei Kopien einer Datei an.**

den per TCP/IP einen Header voranstellen, schreibt er ihn mit »write()« als Erstes in den Pipe-Buffer. Danach „splict" er die Datei an den Kernelbuffer und initiiert so den Transfer. Genauso fügt er Daten in den Strom ein oder hängt welche an. Das Flag »TCP_CORK«, das bei »sendfile()« dazu dient, Daten in den Strom einzufügen, hat damit ausgedient.

Einstiegshilfe

Ein erstes Codebeispiel in Listing 1 demonstriert, wie Linux-Anwendungen »splice()« verwenden. Es veranschaulicht die Abläufe und die Mechanismen beim Kopieren der Daten von Platte auf Platte. Aber solange die Splice-Implementierung der Glibc noch fehlerhaft und die notwendigen Headerdateien nicht in dem Standardpfad »/usr/include/« verfügbar sind, muss man die notwendigen Definitionen selbst vornehmen und den Systemcall direkt aufrufen (Listing 1, Zeilen 9 bis 19).

Die Größe des Pipe-Buffers und damit die Anzahl der in einem Rutsch mit »splice()« maximal zu kopierenden Bytes ist mit 16 Pages festgelegt, auf einer x86-Plattform also 64 KByte. Sollte der Compiler die Headerdatei »page.h« nicht finden, ist der Pfad komplett anzugeben:

```
#include </usr/src/linux-2.6.21-rt1/➥
include/asm/page.h> // Zeile 5
```

Weitere Programmierbeispiele, insbesondere auch in Kombination mit Netzwerkzugriffen und den neuen Systemcalls »tee()« und »vmsplice()«, finden sich im Git-Archiv des Splice-Implementierers Jens Axboe unter der URL [git://git.kernel.dk/data/git/splice.git]. Dort – ebenso wie zum Beispiel unter der Adresse [2] – liegen auch die Manpages zu den neuen Systemcalls. Wie Interessenten mit Hilfe von Git an den Quellcode kommen, verrät der Kasten „Beispielcode im Git-Archiv".

Dem Chef gefällt's

Weitere Infos zu »splice()« – auch vom Linux-Chefentwickler Linus Torvalds persönlich – sind auf der Website [5] zu finden. Der zeigt sich übrigens ganz angetan von den neuen Systemcalls, wie er in einer E-Mail an die Linux-Kernel-Mailingliste mitteilt [6]: „Ganz offensichtlich bin ich von diesem Ansatz begeistert. Ich denke, es ist eine dieser ‚Richtigen Sachen(tm)', die einem nicht allzu häufig über den Weg laufen. Ich kenne sonst niemanden, der das so macht, aber es ist sowohl nützlich als auch clever. Wenn du [Oleg Nesterov] mich jetzt vom Gegenteil überzeugst, werde ich dich für immer hassen ;)." *(ofr)* ■

Beispielcode im Git-Archiv

Jens Axboe, einer der Väter von Splice, verwaltet seine Dateien mit dem Sourcecode-Konstrollsystem Git [7]. Wer seine Beispielprogramme selber ausprobieren will, muss also Git installiert haben. Auf einem Debian- beziehungsweise einem (K)Ubuntu-System sind dazu die Pakete »cogito« oder respektive und »git-core« erforderlich. Axboes Quellcode lässt sich nach dem Auschecken einfach per »make« übersetzen:

```
quade@ezs-mobil:/tmp/git$ sudo apt-get install git-core
Password: <Geheimes Passwort>
...
quade@ezs-mobil:/tmp/git$ git clone git://git.kernel.dk/data/git/splice.git
Generating pack...
Done counting 268 objects.
Deltifying 268 objects.
 100% (268/268) done
Total 268 (delta 162), reused 0 (delta 0)
quade@ezs-mobil:/tmp/git$ ls
splice
quade@ezs-mobil:/tmp/git$ cd splice/
quade@ezs-mobil:/tmp/git/splice$ ls
ktee.c      splice.2          splice.h         splice-test4s.c vmsplice2.c
ktee-net.c  splice-bench.c    splice-in.c      splice-tonet.c  vmsplice.c
Makefile    splice-cp.c       splice-out.c     tee.2
README      splice-fromnet.c  splice-test4c.c  vmsplice.2
quade@ezs-mobil:/tmp/git/splice$ make
...
```

Nach wenigen Augenblicken liegen die fertig übersetzten Beispielprogramme auf der Platte und warten auf ihren Einsatz.

Infos

[1] Jeff Tranter, „Exploring The Sendfile System Call": Linux Gazette, Ausgabe 91/2003; [http://linuxgazette.net/issue91/tranter.html]
[2] Manpage zu »splice()«: [http://www.die.net/doc/linux/man/man2/splice.2.html]
[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 32: Linux-Magazin 3/07
[4] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 13: Linux-Magazin 8/04
[5] Linus Torvalds, Splice: Diskussion auf der LKML; [http://kerneltrap.org/node/6505]
[6] Linus Torvalds, Kommentar zur Splice-Implementierung: [http://lwn.net/Articles/118760/]
[7] Alberto Planas, „Git, das Versionskontrollsystem des Kernels": Linux-Magazin 3/06, S. 98

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 36

Kern-Technik

Für den nächsten Kernel sieht Linus Torvalds optional ein besonderes Userspace-Treibermodell vor, das auch proprietäre Treiber einbinden könnte. Doch schon die Tücken der Technik könnten den Durchbruch in der Praxis verhindern. Eva-Katharina Kunst, Jürgen Quade



Jahrelang versuchten mehrere Entwickler Linus Torvalds von der Notwendigkeit eines Programmier-Interface für so genannte Userspace-Treiber zu überzeugen – ohne Erfolg. Ein Userspace-Treiber soll – wie jeder andere auch – Anwendungsprogrammen Schnittstellen für den Hardwarezugriff bereitstellen, er steuert die Hardware aber von der Applikationsebene, arbeitet also im nicht-privilegierten Modus.

Doch jetzt hat Linus seine Blockade aufgegeben: Er erlaubt in dem kommenden Kernel 2.6.23 solche Userspace-Treiber und stellt dafür ein Interface im Kernel bereit [1]. Der Code geht auf Greg Kroah-Hartman und sein Interface Industrial IO zurück [2]. In den einschlägigen Foren diskutieren nun Linux-Anhänger, ob einerseits vielleicht bald GPL-allergische Firmen endlich Binärtreiber für ihre Hardware schreiben und ob andererseits Linux damit langsam zu der von Linus Torvalds stets abgelehnten Mikrokernel-Architektur mutiere. Denn, so die These der reinen Informatik-Lehre, je mehr Code im Userspace, desto sicherer und stabiler das System.

Um einen Userspace-Treiber für Linux zu schreiben, muss der Entwickler allerdings im Kernel anfangen. Der dort erforderliche Code verbindet das reale Gerät mit dem Userspace-Teil des Treibers (siehe Abbildung 1). Dieser Part sorgt für die Vergabe einer Minornummer und reserviert bei Bedarf Hardwareressourcen. Außerdem legt der Kernel Pseudodateien im Sys-Filesystem an, über die sich der Userspace-Part über die Adressen für den späteren Zugriff informiert.

Userland greift durch

Für den Hardwarezugriff blendet der Userpart nämlich die Speicherbereiche der Hardware per Mmap in seinen Adressraum ein. Danach darf er die betreffenden Register lesen und schreiben. Falls die Hardware Interrupts erzeugt, muss der Kernelteil zudem eine Interrupt-Service-Routine (ISR) aufweisen.

Das Userland kann sich über das Auslösen eines Interrupts informieren lassen, indem es blockierend von einer Gerätedatei (»/dev/uio0«) liest, die der Kernel erzeugt. Sobald ein Interrupt eintrifft, weckt das den wartenden Rechenprozess im Userspace. Die eigentlichen Daten tauschen beide Treiberteile dann über die eingeblendeten Hardwareregister (Adressen) aus.

Das Userspace-Treibermodell spezifiziert also im Wesentlichen nur das Einblenden von Hardwareressourcen in den Adressenraum einer Applikation und die Benachrichtigung über ausgelöste Events (Interrupts) – per Blocking Read oder über »select()« auf eine Gerätedatei. Wie letztlich Applikationen über einen Userspace-Treiber auf das Gerät zugreifen, bleibt komplett dem Treiberentwickler überlassen (siehe Abbildung 1). Folglich nennen die Kernelentwickler den Mechanismus auch nur User I/O (UIO).

Der Kernelpart jongliert mit drei Objekten

Listing 1 zeigt den Kernelpart eines minimalistischen Userspace-Treibers ohne Interruptbehandlung. Insgesamt füllt der Code drei Datenstrukturen (Objekte) mit Inhalt und übergibt sie dem Kernel: Das

Tabelle 1: Funktionen des UIO-Subsystems

Kernelfunktion	Beschreibung
extern int uio_register_device(struct device *parent, struct uio_info *info);	Meldet den über »info« referenzierten Userspace-Treiber beim UIO-Subsystem an. »parent« spezifiziert den Eintrag im Sys-Filesystem, der das zugehörige Gerät identifiziert.
extern void uio_unregister_device(struct uio_info *info);	Meldet den über »info« referenzierten Userspace-Treiber beim UIO-Subsystem ab.
extern void uio_event_notify(struct uio_info *info);	Weckt einen schlafenden Rechenprozess auf, der auf die zum Userspace-Treiber gehörende Gerätedatei referenziert.

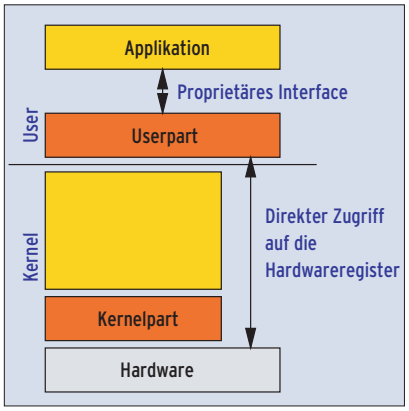


Abbildung 1: Neben dem Userpart benötigt der Userspace-Treiber immer auch einen Kernelpart. Wie eine Applikation auf die angebundene Hardware zugreift, ist nicht festgelegt.

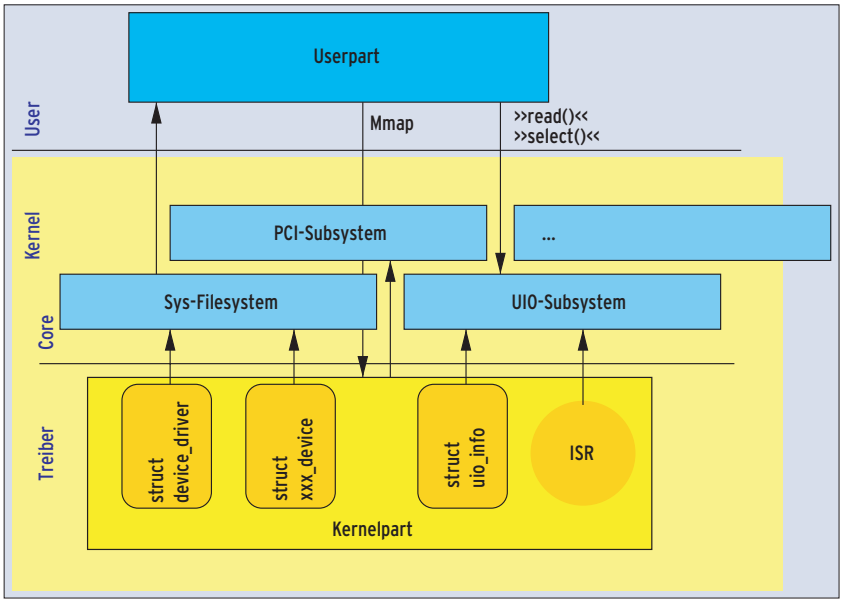


Abbildung 2: Im Kernelteil instanziiert der Treiber drei Objekte und übergibt sie dem Kernelcore.

Gerätemodell respektive Sys-Filesystem benötigt ein Treiber- (»struct device_driver«) und ein Geräteobjekt (zum Beispiel »struct platform_device«), das UIO-Subsystem ein Info-Objekt »uio_info« (Abbildung 2).

Wer für das Sys-FS-Treiberobjekt die Methoden »probe()« und »remove()« implementiert, kann hier die Hardware-Adressen identifizieren und ins Info-Objekt eintragen. Bei einer PCI-Hardware eignen sich zum Identifizieren der Adresslagen die aus **[3]** bekannten Kernelfunktionen. In diesem Fall lässt sich auch statt des Plattform-Device ein PCI-Device instanziieren. Ein Aufruf der Funktion »uio_register_device()« (Syntax siehe Tabelle 1) übergibt das initialisierte Info-Objekt schließlich dem UIO-Subsystem.

Leistungsshow

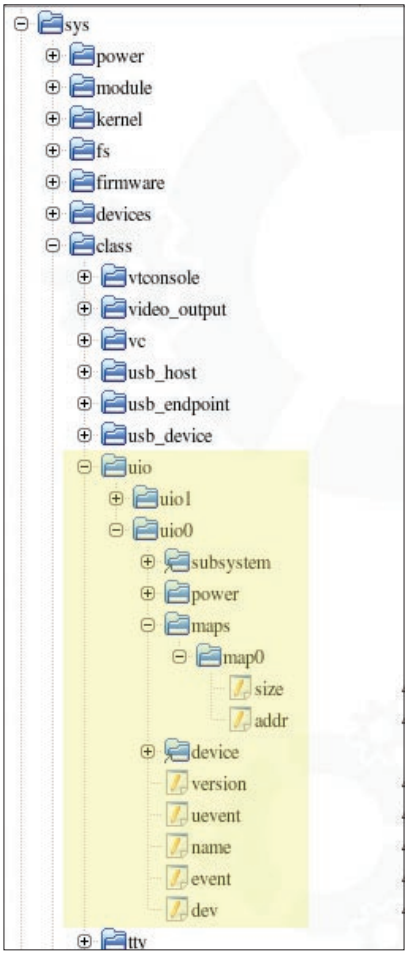
Im Rahmen dieser Registrierung überprüft das UIO-Subsystem, ob es im Gerätemodell die Geräteklasse »uio« bereits gibt. Ist dies nicht der Fall, legt es die Klasse an. Außerdem stellt das UIO-Subsystem sicher, dass das Modul eine Majornummer zugewiesen bekommen hat. Danach reserviert das Subsystem für den neuen Treiber eine Minornummer; Udev erzeugt die zugehörige Gerätedatei, zum Beispiel »/dev/uio0«.

Schließlich legt die Routine die entsprechenden Attributdateien an und hängt – falls vom Treiberentwickler implementiert – die Interrupt-Service-Routine ein. Um diesen Kernelcode zu schreiben, benötigt der Programmierer also recht gute Kenntnisse über den Kernel, besonders im Umgang mit dem Gerätemodell, dem Sys-FS und der Implementierung von Interrupt-Service-Routinen.

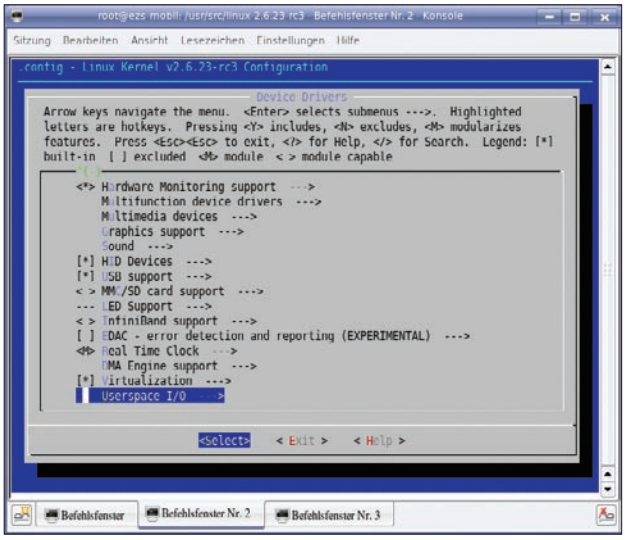
Für jeden Treiber, der sich per »uio_register_device()« beim UIO-Subsystem

Listing 1: »kernel_part.c« des Userspace-Treibers

```
01 #include <linux/module.h>
02 #include <linux/platform_device.h>
03 #include <linux/uio_driver.h>
04
05 struct uio_info kpart_info = {
06   .name = "kpart",
07   .version = "0.1",
08   .irq = UIO_IRQ_NONE,
09 };
10
11 static int drv_kpart_probe(struct device
   *dev);
12 static int drv_kpart_remove(struct device
   *dev);
13 static struct device_driver uio_dummy_driver
   = {
14   .name   = "kpart",
15   .bus    = &platform_bus_type,
16   .probe  = drv_kpart_probe,
17   .remove = drv_kpart_remove,
18 };
19
20 static int drv_kpart_probe(struct device *dev)
21 {
22   printk("drv_kpart_probe( %p )\n", dev );
23   kpart_info.mem[0].addr = (unsigned long)
   kmalloc(512, GFP_KERNEL);
24   if( kpart_info.mem[0].addr==0 )
25       return -ENOMEM;
26   kpart_info.mem[0].memtype =
    UIO_MEM_LOGICAL;
27   kpart_info.mem[0].size = 512;
28   if( uio_register_device(dev,&kpart_info) )
29       return -ENODEV;
30   return 0;
31 }
32
33 static int drv_kpart_remove(struct
   device *dev)
34 {
35   uio_unregister_device(&kpart_info);
36   return 0;
37 }
38
39 static struct platform_device
    *uio_dummy_device;
40 static int __init uio_kpart_init(void)
41 {
42   uio_dummy_device = platform_device_
   register_simple("kpart", -1,
43
            NULL, 0);
44   return driver_register(&uio_dummy_driver);
45 }
46
47 static void __exit uio_kpart_exit(void)
48 {
49   platform_device_unregister
   (uio_dummy_device);
50   driver_unregister(&uio_dummy_driver);
51 }
52
53 module_init( uio_kpart_init );
54 module_exit( uio_kpart_exit );
55
56 MODULE_LICENSE("GPL");
```



◀ Abbildung 3: Sobald der Kernelpart eines Userspace-Treibers geladen ist, legt UIO Informationen im Sys-Filesystem an. Für den Zugriff auf die Hardware muss sich der Userpart die Informationen holen und sie auswerten.



▶ Abbildung 4: Die ab Kernel 2.6.23 geplanten Userspace-Treiber sind in der Kernelkonfiguration anfangs deaktiviert.

anmeldet, legt dieses – wie oben gerade beschrieben – eine Gerätedatei nach dem Schema »/dev/uio0«, »/dev/uio1«, »/dev/uio2« und so weiter an. Jetzt ist es die Aufgabe des Userparts festzustellen, hinter welcher der gelisteten Gerätedateien sich die betreffende Hardware befindet. Dazu kann sie wieder einmal die Pseudodateien im Sys-Filesystem nutzen (Abbildung 3).

Unter dem Pfad »/sys/class/uio/uio*X*« (»*X*« durch »0«, »1« und so weiter ersetzen) lässt sich die Datei »name« auslesen, die den hoffentlich eindeutigen Treibernamen enthält. Unter dem entsprechenden Verzeichnis findet der Userpart dann die vom Kernelpart abgelegten Informationen bezüglich der Adressbereiche. Die blendet der Userpart per »mmap()« in den eigenen Adressraum ein. Den dafür benötigten Filedeskriptor erhält, wer die zugehörige Gerätedatei, zum Beispiel »/dev/uio0«, öffnet. War »mmap()« erfolgreich, darf der Userpart (siehe Beispielcode in Listing 2) auf die Hardwareregister zugreifen.

Eine Routine, die über das Auftreten von Interrupts informiert werden will, ruft »select()« oder »read()« im nicht-blockierenden Modus auf. »read()« liefert übrigens die Anzahl der aufgetretenen Events (Interrupts) als 4-Byte-Wert zurück. Zur Sicherheit noch mal: Der Mechanismus funktioniert nur dann, wenn man vom Kernel genau 4 Bytes mit dem Datentyp »s32« anfordert! Ein Zugriff über das Kommando »cat« beispielsweise, das grundsätzlich 4096 Bytes anfordert, geht schief.

Die ISR ist schnell gemacht

Die im Kernel hierzu notwendigerweise zu kodierende Interrupt-Service-Routine darf schlank ausfallen. Sie muss meist nur feststellen, ob die eigene Hardware den Interrupt ausgelöst hat, und wenn ja, den Interrupt Hardware-seitig quittieren und »IRQ_HANDLED« zurückgeben. Wer innerhalb des Kernels auf einem an-

Listing 2: »user_part.c« ohne Identifikation von »/dev/udev*X*«

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <stdlib.h>
04 #include <unistd.h>
05 #include <sys/mman.h>
06
07 #define UIO_DEV "/dev/uio0"
08 #define UIO_ADDR "/sys/class/uio/uio0/maps/
   map0/addr"
09 #define UIO_SIZE "/sys/class/uio/uio0/maps/
   map0/size"
10
11 static char uio_addr_buf[16],
    uio_size_buf[16];
12
13 int main( int argc, char **argv )
14 {
15   int uio_fd, addr_fd, size_fd;
16   int uio_size;
17   void *uio_addr, *access_address;
18
19   addr_fd = open( UIO_ADDR, O_RDONLY );
20   size_fd = open( UIO_SIZE, O_RDONLY );
21   uio_fd  = open( UIO_DEV, O_RDONLY );
22   if( addr_fd<0 || size_fd<0 || uio_fd<0 ) {
23       fprintf(stderr,"Kann UIO-Dateien nicht
   öffnen...\n");
24       return -1;
25   }
26
27   read( addr_fd, uio_addr_buf, sizeof
   (uio_addr_buf) );
28   read( size_fd, uio_size_buf, sizeof
   (uio_size_buf) );
29   uio_addr = (void *)strtoul( uio_addr_buf,
   NULL, 0 );
30   uio_size = (int)strtol( uio_size_buf,
    NULL, 0 );
31
32   access_address = mmap(NULL, uio_size,
33   PROT_READ, MAP_SHARED, uio_fd, 0);
34   printf("Auf die HW-Adresse %p (Länge %d) "
35          "kann über die logischer Adresse %p"
36          " zugegriffen werden\n", uio_addr,
37          uio_size, access_address);
38
39   // Ab hier kann der Zugriff auf die
   Hardware-Register erfolgen...
40   // ...
41   return 0;
42 }
```

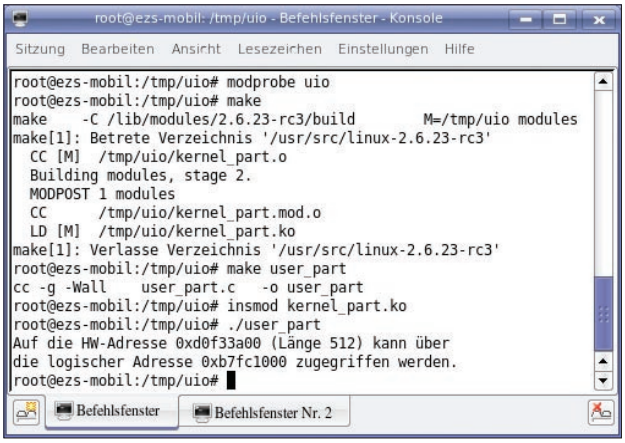


Abbildung 5: Der Beispielcode demonstriert das Zusammenspiel zwischen Kernel- und Userpart des Userspace-Treibers.

Vor- und Nachteile des Userspace-Treibermodells

Vorteile:
- Versionswechsel: Der Anwender muss nur den Kernelpart rekompilieren und zuvor eventuell anpassen.
- Stabilität: Der Kernel ist vor fehlerhaften Treibern geschützt.
- Floating-Point steht zur Verfügung.
- Gute Effizienz, da kein Prozesswechsel notwendig ist.
- Lizenz: Das Offenlegen des Userpart-Quellcode ist nicht zwingend. (Diese Auslegung der GPL ist jedoch umstritten.)

Nachteile:
- Kernel-Know-how nötig in den Bereichen Standardtreiber, Sys-FS, IRQ und PCI.
- Das Timing ist ungenauer als im Kernel.
- Reaktion auf Interrupts: Antwortzeiten sind länger als im Kernel (Prozesswechsel).
- Funktionalität im Userland stark eingeschränkt, so steht kein DMA zur Verfügung.
- API: Die Applikation kann kein definiertes Interface zum Zugriff auf das Gerät nutzen.
- Eingeschränkte, nicht immanente Sicherheit.

deren Wege den schlafenden User-Part aufwecken will, ruft die Funktion »uio_event_notify()« auf.

Selbst Hand anlegen

Wer das neue Interface ausprobieren will, muss sich einen sehr neuen Kernel zulegen, zum Beispiel 2.6.23-rc3, und in dessen Kernelkonfiguration das UIO-Subsystem unter »Device Drivers | Userspace I/O | Userspace I/O Drivers« einschalten (siehe Abbildung 4). Ist UIO im Kernel als Modul übersetzt, lädt Root es durch Aufruf von »modprobe uio«.

Danach geht es daran, den Userspace-Treiber zu generieren, also Kernelmodul und den Userspace-Part. Falls die Quelldateien von Listing 1, 2 und 3 (Makefile) im Verzeichnis »/tmp/uio« liegen, funktionieren Erzeugen und Testen wie in Abbildung 5 dargestellt.

Listing 3: Makefile für Kernel- und Userpart

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m := kernel_part.o
03
04 else
05 KDIR  := /lib/modules/$(shell uname -r)/build
06 PWD   := $(shell pwd)
07
08 default:
09   $(MAKE)   -C $(KDIR)   M=$(PWD) modules
10 endif
11
12 clean:
13   rm -f *.ko *.o user_part
14
15 user_part: user_part.c
16   cc -g -Wall    user_part.c   -o user_part
```

Um das Beispiel überschaubar zu halten, koppelt es weder eine PCI-Hardware an noch verwendet es Interrupts. Informationen dazu gibt es unter [3] und [4]. Der letzte Link liefert zudem Hinweise, dass und wie der Programmierer eigene »open()«-, »release()«- und »mmap()«-Funktionen im Kernel einhängt, um beispielsweise einen ausgefeilten Zugriffsschutz zu realisieren. Fragen bezüglich der Sicherheit sind nämlich nicht wirklich gelöst.

Auch die Autoren des UIO-Subsystems haben Beispielcode veröffentlicht. Anders als in der Kerneldokumentation angegeben [4] befindet sich aber kein Template-Treiber in den Kernelquellen. Der auf Kernel 2.6.23-rc3 angepasste Treiber »uio_dummy.c« mit dem zugehörigen Userpart »uio_events.c« lässt sich aber über [5] herunterladen.

Was soll's?

Fazit: Mit der Aufnahme des UIO-Subsystems in den aktuellen Kernel vollzieht Linus Torvalds nicht die große Wendung in Sachen Userspace-Treiber. Der keine, 1000 Zeilen umfassende Code des UIO-Subsystems bietet bei genauer Betrachtung technisch kaum mehr, als mit Mmap (siehe [6]) und normalen Treiberinterfaces schon vorher möglich war. Das Einbinden von Hardware in die vorhandenen Kernelsubsysteme, beispielsweise für Netzwerk- oder Blockgerätetreiber, ist überhaupt nicht vorgesehen (siehe Kasten „Vor- und Nachteile des Userspace-Treibermodells“).

Die eingeschränkte Funktionalität, das Fehlen eines definierten Interface zwischen Applikation und Userspace-Treiber und das notwendige umfangreiche Wissen über den Kernel lassen eine große Verbreitung außerhalb des ursächlichen Einsatzes für Embedded Linux als unwahrscheinlich erscheinen. *(jk)* ■

Infos

[1] Linus Torvalds Ankündigung zum Kernel 2.6.23-rc1: [http://article.gmane.org/gmane.linux.kernel/559066]
[2] Simple userspace interface for PCI drivers: [http://www.uwsg.indiana.edu/hypermail/linux/kernel/0608.3/1908.html]
[3] Eva-Katharina Kunst, Jürgen Quade, „Linux-Treiber entwickeln“: Dpunkt-Verlag, 2. Auflage, Januar 2006
[4] Hans-Jürgen Koch, „The Userspace I/O-Howto“: [/usr/src/linux-2.6.23-rc3/Documentation/DocBook/uio-howto.tmpl]
[5] Vollständige Listings zum Artikel: [ftp://www.linux-magazin.de/pub/listings/magazin/2007/11/Kern-Technik]
[6] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 32: Linux-Magazin 03/07, S. 104

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 37

Kern-Technik

Das Firmware-Subsystem fristet ein Dasein als Mauerblümchen. Zu Unrecht, denn hinter dem eleganten Kernel- und User-Interface steckt sehr moderne Betriebssystem-Technologie. Eva-Katharina Kunst, Jürgen Quade



Ein heikles Thema war zur Laufzeit nachzuladende Firmware bei Linux schon immer. Für Hardware-nahe Software liegt oft kein Quellcode vor. Folglich – so die Konsequenz aus der GPL – darf ein Entwickler sie nicht direkt in den Linux-Kernel respektive einen Linux-Treiber integrieren. Dabei setzen Hardwarehersteller oft auf Firmware: Intels Centrino-WLAN-Adapter funken erst, wenn der auf dem Adapter eingesetzte Prozessor seine Firmware geladen hat. Und ohne Extra-Software weigert sich der DVB-T-Empfänger standhaft, aktuelle Bilder zu präsentieren.

Für Hersteller und Kunden ist auswechselbare Systemsoftware vorteilhaft: Fehler lassen sich bei bereits ausgelieferten und im Einsatz befindlichen Komponenten ausbessern und sie aktiviert auch auf älterer Hardware neue Funktionen.

Entsprechend den Lizenzbestimmungen haben Entwickler aber schon seit Längerem die Methode fallen gelassen, die Firmware als obskuren Hexstring im Quellcode des zugehörigen Gerätetreibers zu verstecken. Stattdessen lädt der Kernel das Firmware-Image aus dem Userspace nach und kopiert es dann in die zugehörige Hardware. Anfänglich hatte jeder Entwickler für sich dazu eine eigene Schnittstelle geschaffen und implementiert, doch bereits in den Kerneln der Reihe 2.4 führte Torvalds eine einfach zu nutzende, standardisierte Schnittstelle für Firmware ein.

Für den Anwender ist die Schnittstelle erfreulich einfach: Er legt die Firmware-Datei an zentraler Stelle ab, typisch sind »/lib/firmware« oder »/usr/local/lib/firmware«. Sobald Linux den passenden Treiber lädt, übergibt es ihm das Firmware-Image. Der Treiber sorgt für den Transport in die Hardware.

Im Detail betrachtet sind die Abläufe etwas komplexer, da Kernel und Userland harmonisch zusammenspielen müssen, bis sich die richtigen Daten in der Hardware befinden.

Zwischen Kern und Userland

Der Ablauf beginnt, sobald ein Benutzer einen Gerätetreiber für eine Hardware lädt, die eine Firmware benötigt (siehe Abbildung 1). Der Treiber teilt dem Kernel den Namen der an zentraler Stelle (»/lib/firmware«) im Filesystem abgelegten Firmware-Datei mit. Das Firmware-Subsystem des Kernels legt daraufhin dynamisch im Sys-Filesystem ein Verzeichnis an. Es repräsentiert den Treiber und enthält die virtuellen Dateien »loading« und »data«.

Der Treiber informiert den Udev-Daemon (»udevd«), der Gerätedateien automatisch anlegt [1]. Der Daemon erkennt am übermittelten »uevent«, dass eine Firmware zu laden ist, und startet das Helferprogramm »/lib/udevd/firmware.sh« oder »/lib/udevd/firmware_helper« (siehe Kasten „Ubuntu und die Firmware“). Diese Software übergibt über die virtuellen Dateien »loading« und »data« das Firmware-Image an den Kernel.

Das definierte Protokoll sieht vor, dass die Software zunächst eine »1« in die Datei »loading« schreibt und danach die Firmware-Daten in die virtuelle Datei »data« kopiert. Das Firmware-Subsystem übernimmt die Daten und legt sie in einen mitwachsenden, zusammenhängenden Bereich des Kernelspeichers ab. Nachdem »firmware.sh« alle Daten kopiert hat, signalisiert das Programm das Ende der Aktion, indem es eine »0« in »loading« schreibt (Abbildung 2).

Eine »-1« dagegen signalisiert den Abbruch des Ladevorgangs. Das Firmware-Subsystem entfernt daraufhin die virtuel-

Ubuntu und die Firmware

Ubuntu verwendet auch in der neuesten Version 7.10 nicht das den Orginalquellen beiliegende Skript »firmware.sh«, sondern den älteren C-Code »firmware_helper.c«. Anders als in den Orginalquellen von »firmware_helper.c« vorgesehen, ermöglicht das von den Ubuntu-Entwicklern modifizierte Programm, für unterschiedliche Kernel auch unterschiedliche Firmware-Versionen zu verwenden. Kernel-spezifische Firmware-Images erwartet es in einem Verzeichnis unterhalb von »/lib/firmware«, das den Namen der Kernelversion trägt – bei Ubuntu 7.10 beispielsweise »/lib/firmware/2.6.22-14-generic«. Firmware, die Kernel-unabhängig geladen werden soll, findet sich weiterhin im Verzeichnis »/lib/firmware«.

Schnittstellenfunktionen im Kernel

Um das Firmware-Subsystem zu veranlassen, das zum Gerät »device« gehörende und unter dem Namen »name« abgelegte Firmware-Image in einen Speicherbereich des Kernels zu kopieren, bietet es eine Schnittstelle an:

```
int
request_firmware(const struct firmware **fw,
                 const char *name,
                 struct device *device);
```

Sowohl das den Speicherbereich beschreibende Objekt vom Typ »struct firmware« als auch der entsprechende Speicher reserviert das Firmware-Subsystem dynamisch. Das Objekt »struct firmware« enthält die Startadresse des Speicherbereichs, in dem das Firmware-Image liegt, und die zugehörige Länge. Die Adresse des Objektes ist in »fw« gespeichert.
Während des Ladens der Firmware legt sich der zugehörige Thread schlafen. Bei erfolgreichem Laden der Firmware gibt die Funktion »0«, sonst einen negativen Fehlercode zurück. »EINVAL« bedeutet, dass die Adresse für die Aufnahme der Speicherdaten des Firmware-Image ungültig war, »ENOMEM«, dass kein Speicher im Kernel für die Firmware-Struktur oder für die Aufnahme des Firmware-Image ist. Bei »ENOENT« findet die Funktion das Firmware-Image nicht. Die folgende Funktion gibt die mit »fw« assoziierten Ressourcen wieder frei:

```
void
release_firmware(const struct firmware *fw);
```

So veranlasst das Firmware-Subsystem, einen Kernel-Thread zu starten und in ihm die Kernelfunktion »request_firmware()« aufzurufen:

```
int
request_firmware_nowait(struct module *mod,
  int uevent, const char *name,
  struct device *device, void *context,
  void (*cont)
(const struct firmware *fw, void *context));
```

Der Parameter »mod« spezifiziert das zugehörige Modul, typischerweise das Makro »THIS_MODULE«, und »name« das zu ladende Firmware-Image. Der Parameter »device« gibt das zugehörige Objekt im Gerätemodell (Sys-Filesystem) an. Nach einem erfolgreichen oder erfolglosen Versuch, die Firmware zu laden, ruft das Subsystem die Callback-Funktion »cont« auf. Neben der Spezifikation, wo das Firmware-Image im Speicher zu finden ist, bekommt die Funktion den frei definierbaren Parameter »context« übergeben.
Beim erfolgreichen Starten des Kernel-Thread gibt die Funktion »0«, sonst einen negativen Fehlercode zurück. »ENOMEM« signalisiert, dass kein Speicher im Kernel vorhanden ist, »EFAULT«, dass das Modul bereits entladen ist.

len Dateien und das angelegte Verzeichnis wieder und übergibt dem Treiber die Adresse des Speicherbereichs, in dem das Firmware-Image nun liegt.
Der Treiber sorgt anschließend noch dafür, die Daten der Hardware zu übergeben und den Speicher wieder freizugeben. Der Kernel überwacht den ganzen Vorgang: Wenn der Vorgang nicht binnen 60 Sekunden abgeschlossen ist und das Image im Speicher liegt, sorgt der Kernel für einen Abbruch.

Glücklicherweise sind die beschriebenen Vorgänge alle bereits im Firmware-Subsystem implementiert. Der Treiberprogrammierer bedient ein recht einfaches Interface. Es besteht im Wesentlichen aus den beiden Funktionen »request_firmware()« und »release_firmware()« und den beiden Objekten »struct firmware« und »struct device« (siehe Kasten „Schnittstellenfunktionen im Kernel“). Das Device-Objekt ist bei den meisten Treibern ohnehin implizit vorhanden.

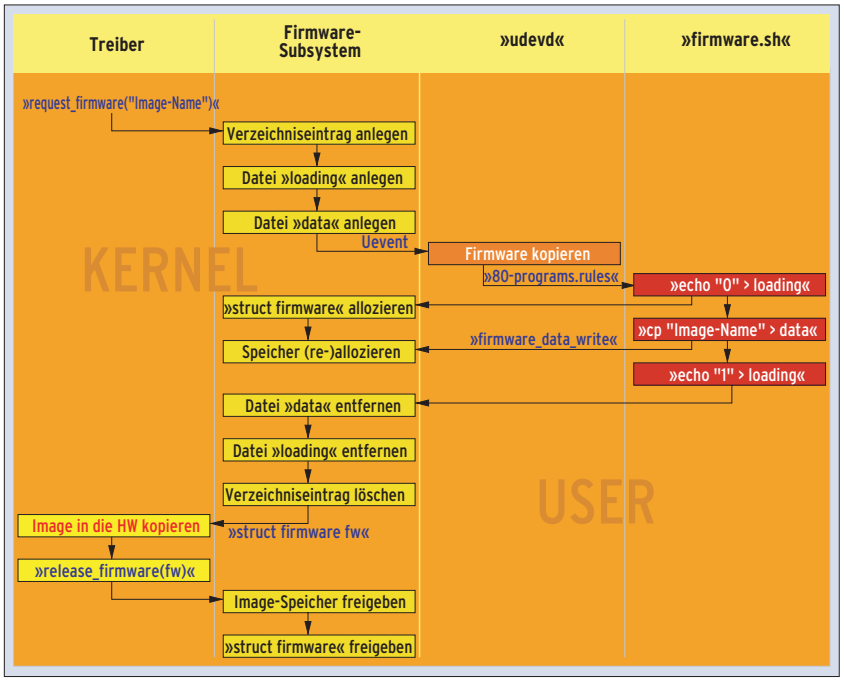


Abbildung 1: Die Funktion zum Laden der Firmware löst innerhalb des Betriebssystems umfangreiche Aktivitäten zwischen Treiber, Firmware-Subsystem und dem kontrollierenden Userprogramm aus.

Wenn eine Karte beispielsweise per PCI angebunden ist, stellt der Kernel ein Objekt vom Typ »struct pci_dev« bereit, das seinerseits im Element »dev« das gesuchte Device-Objekt enthält.
Sollte ein Treiber noch kein Device-Objekt haben, erhält er es mit Hilfe der Funktion »device_register()« und muss es aktivieren (siehe Listing 1, Zeile 33). Speicher wird reserviert und ein Objektname bereitgestellt. Die Release-Funktion in den Zeilen 7 bis 11 sorgt dafür, dass niemand den Treiber entladen kann, solange jemand noch auf das Device-Objekt zugreift, zum Beispiel über das Sys-Filesystem.
Die Funktion »request_firmware()« übergibt dem Kernel-Subsystem den Namen

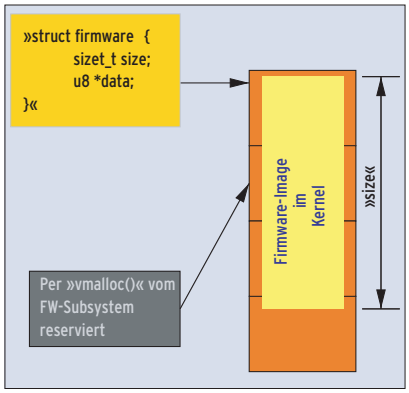


Abbildung 2: Das Firmware-Subsystem reserviert Kernelspeicher für das Image und ein Objekt, das es beschreibt. Die Größe des Speichers wird – wenn es sein muss durch Umkopieren – dynamisch als ein Vielfaches der Pagesize angepasst.

der Firmware-Datei und das Device-Objekt. Der Name der Firmware-Datei darf nicht länger als 30 Zeichen sein. Außerdem wichtig: Die Funktion blockiert so lange, bis »udevd« entweder das erfolgreiche Kopieren der Firmware-Datei oder einen Fehler meldet, beispielsweise wenn es die Datei nicht findet.

Entwickler dürfen diese Funktion also nicht innerhalb einer Interrupt-Service-Routine, eines Tasklets oder eines Timers verwenden. Diese Einschränkung ist aber nicht gravierend, da ohnehin typischerweise der Treiber die Firmware anfordert, wenn er geladen wird, also in der durch das Makro »mod_init()« angegebenen Funktion. Wem das nicht reicht, für den gibt es einen bisher noch nicht erwähnten asynchronen Modus.

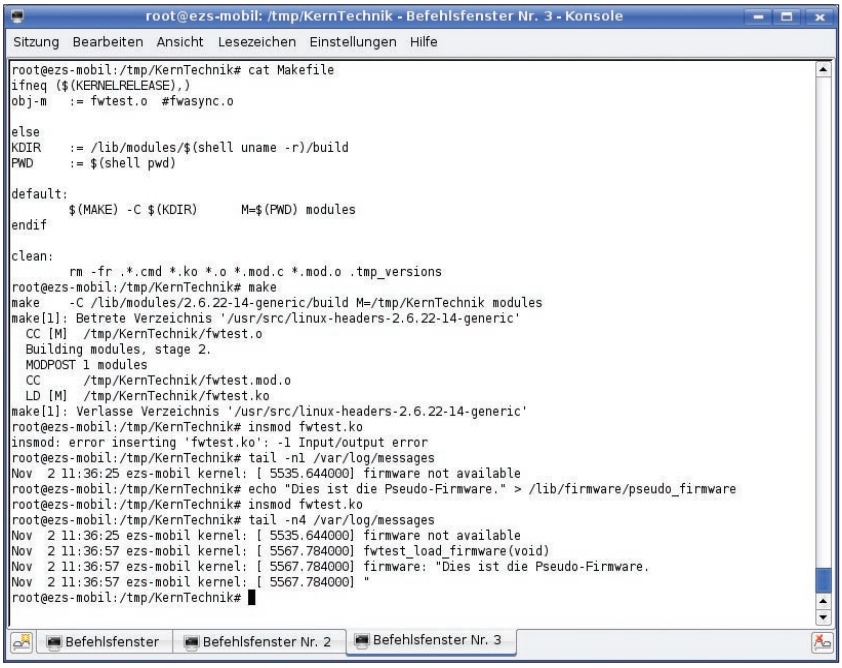


Abbildung 3: Bevor sich das Bespiel-Modul in den Kernel lädt, muss vorher eine Firmware im Verzeichnis »/lib/firmware« bereitstehen. Nach geglücktem Start gibt das Modul die Firmware im Kernel-Log aus.

Treiber-Angelegenheit

Konnte eines der erwähnten Helferprogramme die Firmware in den Kernel transportieren, gibt die Funktion »request_firmware()« die Adresse des »struct firmware«-Objekts zurück (siehe Abbildung 2). Dieses Objekt enthält die Startadresse der Daten und die zugehörige Länge. Meist wird der Treiber jetzt die Firmware in das Gerät kopieren. Dazu greift er mit Hilfe der Funktionen »in[bwl]« oder »out[bwl]« bei Port-mapped-IO beziehungsweise »read[bwl]« oder »write[bwl]« bei Memory-mapped-IO direkt auf die Register der Hardware zu. Dann ruft er die Funktion »release_firmware()« auf. Sie gibt den Speicherbereich wieder frei, in den der Treiber die Firmware temporär geladen hat.

Listing 1 zeigt als Beispiel den Einsatz der beiden Funktionen. Wer den Code testet, muss natürlich eine Firmware-Datei zur Verfügung stellen. Der Code verlangt hier eine Firmware, die Daten in Textform enthält. So lässt sich das Laden gut veranschaulichen. Das in Abbildung 3 demonstrierte Beispiel erzeugt die benötigte Pseudo-Firmware. Ein Anwender generiert und lädt dort ebenfalls den Treiber, der über das Kernel-Log Informatonen ausgibt. Dieses Beispiel läuft unter Ubuntu.

Wie bereits erwähnt, bietet das Firmware-Subsytem auch die Möglichkeit, Firmware asynchron, also auch aus Interrup-Service-Routinen, Tasklets oder einem Timer heraus zu laden. In diesem Fall ruft der Gerätetreiber die Funktion »request_firmware_nowait()« statt »request_firmware()« auf (siehe Listing 2). Bevor die Funktion die Kontrolle wieder an die aufrufende Funktion zurückgibt,

Listing 1: Anfordern von Firmware (»fwtest.c«)

```
01 #include <linux/device.h>
02 #include <linux/firmware.h>
03
04 static DECLARE_COMPLETION(obj_not_in_use);
05 static const struct firmware *fw_entry;
06
07 static void
08 firmware_device_release(struct device *dev)
09 {
10     complete(&obj_not_in_use);
11 }
12
13 static struct device fwtest_device = {
14     .bus_id    = "fwtest",
15     .release   = firmware_device_release,
16 };
17
18 static void
19 fwtest_load_firmware(char *firmware, int size)
20 {
21     unsigned char buf[80+1];
22     int to_copy;
23
24     printk("fwtest_load_firmware(void)\n");
25     to_copy = min((int)(sizeof(buf)-1), size);
26     memcpy(buf, firmware, to_copy);
27     buf[to_copy] = '\0';
28     printk("firmware: \"%s\"\n", buf);
29 }
30
31 static int __init fwtest_init(void)
32 {
33     if (device_register(&fwtest_device)) {
34         printk("device_register failed\n");
35         return -EIO;
36     }
37     if (request_firmware(&fw_entry,
38          "pseudo_firmware", &fwtest_device)) {
39         printk("firmware not available\n");
40         device_unregister(&fwtest_device);
41         return -EIO;
42     }
43
44     fwtest_load_firmware(fw_entry->data,
45                          fw_entry->size);
46
47     release_firmware(fw_entry);
48     return 0;
49 }
50
51 static void __exit fwtest_exit(void)
52 {
53     device_unregister(&fwtest_device);
54     wait_for_completion(&obj_not_in_use);
55     printk("fwtest_exit()\n");
56 }
57
58 module_init(fwtest_init);
59 module_exit(fwtest_exit);
60 MODULE_LICENSE("GPL");
```

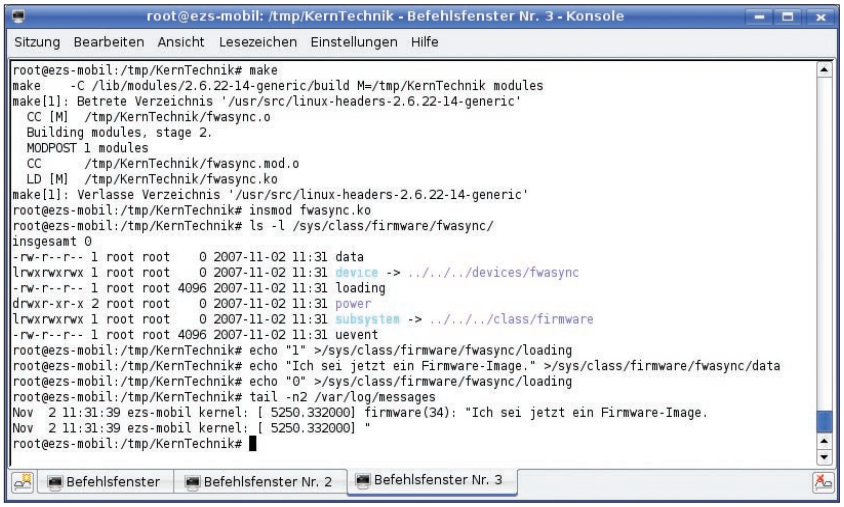


Abbildung 4: Um eine Firmware zu laden, ist der asynchrone Ansatz ohne Verwendung des »udevd« robuster gegen Unterbrechungen im Ladevorgang. Er erfordert nur wenig mehr Aufwand bei der Implementierung.

startet sie für den Entwickler transparent einen Kernel-Thread. In dessen Kontext ruft sie zunächst »request_firmware()« und danach eine vom Programmierer bereitgestellte Callback-Funktion auf. Die Funktion kann das inzwischen in den Kernel kopierte Firmware-Image zum Beispiel zur Hardware übertragen.

Hierbei ist aber zu beachten, dass nach Abschluss der Callback-Funktion der Kernel noch den Speicher, in dem sich das Firmware-Image befindet, wieder freigibt. Der Entwickler darf also, wenn er asynchron lädt, die Release-Funktion »release_firmware()« nicht aufrufen. Das Subsystem geht beim asynchronen Modus davon aus, dass nach Ablauf der Callback-Funktion das Image der Firmware nicht mehr benötigt wird, und räumt es selbst auf.

Asynchrones Laden

Das asynchrone Laden bietet darüber hinaus die Möglichkeit, den »uevent« zu unterdrücken. Da einige eingebettete Geräte auf einen »udevd« verzichten oder weil einfaches Kopieren den Ladevorgang nicht realisieren kann, ist dies gelegentlich erforderlich. Dazu erhält die Funktion »request_firmware_nowait()« das Flag »FW_ACTION_NOHOTPLUG« statt des sonst üblichen Flag »FW_ACTION_HOTPLUG«. Dann legt das Firmware-Subsystem nur die Dateien im Sys-Filesystem an. Die Treiberentwickler sind selbst aufgefordert, die Firmware nach dem beschriebenen Protokoll dem Gerätetreiber zu übergeben.

Listing 2 verdeutlicht, dass sich ein Image auch asynchron einfach lädt. Bei den Daten muss es sich übrigens nicht unbedingt um eine Firmware handeln. Abbildung 4 zeigt dazu, wie ein Anwender per Hand ein Image aus dem Userland dem Kernel übergibt.

Saubere Schnittstelle

Einfach, übersichtlich und funktional – mit diesen drei Attribute darf sich das Firmware-Subsystem bezüglich seiner Schnittstellen für den Treiberentwickler und für den Anwender schmücken. Um den Bilderkreis wieder zu schließen: Das Interface ist selbstredend auch für Firmware gedacht, die offengelegt ist; am besten für Firmware, die selbst auf einem Linux-System basiert. *(mg)* ■

Infos

[1] Oliver Frommel, Jens-Christoph Brendel, „Udev: Dynamisches Gerätemanagement“: Linux-Magazin 09/06, S. 32

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux-Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 2: Asynchrones Laden (»fwasync.c«)

```
01 #include <linux/device.h>
02 #include <linux/firmware.h>
03
04 static DECLARE_COMPLETION(obj_not_in_use);
05
06 static void
07 firmware_device_release(struct device *dev)
08 {
09     complete(&obj_not_in_use);
10 }
11 static struct device fwasync_device = {
12     .bus_id    = "fwasync",
13     .release   = firmware_device_release,
14 };
15
16 static void fwasync_cont(const struct
17           firmware *fw, void *private_data)
18 {
19     unsigned char buf[80+1];
20     int to_copy;
21
22     if (fw == NULL) {
23         printk("fwasync_cont:
24                 firmware load failed\n");
25         return;
26     }
27     to_copy = min((sizeof(buf)-1), fw->size);
28     memcpy(buf, fw->data, to_copy);
29     buf[to_copy] = '\0';
30     printk("firmware(%d): \"%s\"\n",
31           (int)fw->size, buf);
32 }
33
34 static int __init fwasync_init(void)
35 {
36     if (device_register(&fwasync_device)) {
37         printk("device_register failed\n");
38         return -EIO;
39     }
40     if (request_firmware_nowait(THIS_MODULE,
41           FW_ACTION_NOHOTPLUG,
42           "pseudo_firmware", &fwasync_device,
43           "private_data", fwasync_cont)) {
44         printk("fwasync:
45           request_firmware_nowait failed\n");
46         device_unregister( &fwasync_device );
47         return -EIO;
48     }
49     return 0;
50 }
51
52 static void __exit fwasync_exit(void)
53 {
54     device_unregister( &fwasync_device );
55     wait_for_completion( &obj_not_in_use );
56 }
57
58 module_init(fwasync_init);
59 module_exit(fwasync_exit);
60 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 38

Kern-Technik

Angriffe über Buffer Overflows haben es schwerer, wenn Address-Space Layout Randomization (ASLR) oder die Hardware-Unterstützung durch NX- und XD-Bits aktiv ist. Die Implementation der 32-Bit-Architektur bietet Linux-Hackern viele Möglichkeiten zur Absicherung. Eva-Katharina Kunst, Jürgen Quade



Als Multiusersystem ist Unix aus Tradition der Sicherheit verpflichtet. Zutrittsschutz mit Benutzername und Passwort, die Unterscheidung zwischen Superuser und übrigen Benutzern sowie Dateizugriffsrechte, die lesenden, schreibenden und ausführenden Zugriff für den Besitzer, eine Gruppe und den Rest getrennt festlegen, waren Unix schon in die Wiege gelegt.

Heute reichen diese Schutzmechanismen nicht mehr aus, um aktuelle Methoden von Angreifern abzuwehren. Häufiger Angriffsvektor sind Buffer Overflows. Bei diesem Angriff nutzen die Exploits der Black Hats zumeist Programmierfehler aus, die es ihnen ermöglichen, Speicherbereiche zu überschreiben.

Tun sie dies auf dem Stack und modifizieren die dort befindliche Rücksprungadresse einer Unterfunktion, bringen sie die CPU dazu, beim »return« der Funktion eigenen Code anzuspringen. Dieser liegt meist auf dem Stack. Noch bequemer ist es, eine Bibliotheksfunktion – zum Beispiel »system()« der Libc – anzuspringen. So gewinnt der Angreifer Kontrolle über die Maschine (siehe Abbildung 1). Zwei Voraussetzungen spielen den Angreifern in die Hände: Die starre Aufteilung des Adressraums und die Möglichkeit, auf dem Stack befindlichen Code auszuführen.

Angriffsmuster

Genau an diesen Stellen setzt der Linux-Kernel an, um Black Hats die Arbeit zu erschweren. Richtig konfiguriert verhindert es der Kernel mit Hardwarehilfe des Prozessors, Code auf dem Stack auszuführen. Außerdem kann er jeder Applikation ein eigenes, durch den Zufall bestimmtes Speichermapping verpassen. Diese Eigenschaften muss der Anwender jedoch auch nutzen.

Speichermapping bezeichnet die Aufteilung des virtuellen Adressraums in Code, Daten, Stack, Heap und Bibliotheken eines Programms. Jeder Applikation stehen auf einem 32-Bit-Linux typischerweise 3 GByte Hauptspeicher zur Verfügung, unabhängig davon, ob nur 64 MByte oder 4 GByte RAM im Rechner stecken. Ohnehin nutzen nur wenige Programme den virtuellen Adressraum vollkommen aus. Somit verteilt Linux die einzelnen Teile des Kernels sinnvoll über den Adressraum von 3 GByte, wie Abbildung 2 illustriert.

Ziemlich am Anfang befinden sich der Code des Programms, die vorinitialisierten (Data) und die nicht vorinitialisierten Daten (BSS), der Stack und der Heap. Der Heap ist der Speicherbereich, den etwa »malloc()« als Basis für dynamisch angeforderten Speicher verwendet. Die gemeinsam genutzten Bibliotheksfunktionen (Dynamic Shared Objects, DSO) blendet der Kernel ebenfalls in den Adressraum ein, etwa Speicherseiten, die eine Applikation per »mmap()« von anderen Prozessen angefordert hat.

Da beim Start eines Prozesses nicht bekannt ist, wie viel Arbeitsspeicher er dynamisch anfordern wird oder wie viele Speicherseiten er per »mmap()« verwenden will, wachsen die Heap- und die Mmap-Region aufeinander zu. Für den Stack, dessen Verbrauch beim Start ebenfalls nicht bekannt ist, reserviert der Kernel innerhalb des Adressraums einfach 128 MByte, falls nichts anderes konfiguriert ist.

Hackerfreundlich

Bei einer derart starren Aufteilung des virtuellen Adressraums ist es für Angreifer leicht, Rücksprungadressen auf dem Stack mit eigenen Adressen zu überschreiben. Daher bringt der Linux-Kernel – sofern gewünscht – mehr Variationen bei der Lage der Adressbereiche ein und verschiebt die Position des Stacks und der Mmap-Region, in der sich auch die gemeinsam genutzten Bibliotheken (DSO) befinden. Bei so verwürfeltem Speichermapping kommen Angreifer nur noch mit Ausprobieren weiter. Raten sie dabei falsch, stürzt die angegriffene Applikation mit hoher Wahrscheinlichkeit ab. Aufmerksamen Anwendern und Admins fällt das auf.

Abbildung 3 zeigt, wie der Kernel eine Speicherlücke vor das obere Ende des

Stacks einfügt. Ein Blick in die Kernelquellen offenbart, dass Linux beim Systemaufruf »execve()« 11 Bits (»0x7ff«) der oberen Startadresse des neuen Stacksegments auf einem 32-Bit-System variiert (Listing 1). Da das Stacksegment immer auf einer Seitengrenze liegt, ist die Startadresse bei 4 KByte großen Speicherseiten (12 Bit) grundsätzlich innerhalb eines 8 MByte (2^{23} Byte) großen Bereichs zu suchen. Mehr noch: Zusätzlich initialisiert Linux den Stackpointer per Zufall auf eine Adresse innerhalb eines 8 KByte großen Bereichs (Listing 2). Da das Betriebssystem den Stackpointer allerdings auf eine 16-Byte-Adresse ausrichtet, ergibt sich eine weitere Variation um ld(8192) - 4 = 9 Bit.

Summa summarum sprechen Security-Fachleute davon, dass die Entropie der Stackadresse 11 Bit + 9 Bit = 20 Bit beträgt und ein Angreifer im für ihn ungünstigsten Fall also 2^{20} (ungefähr eine Million) Versuche braucht, um eine Schwachstelle auszunutzen.

Besser als nichts

Die Entropie der Lage der Mmap-Region ist leider nicht ganz so groß. Sie ist jedoch wichtig, um den „Return to Libc“ genannten Angriff zu verhindern. Der Start dieser Region variiert nur innerhalb eines 1 MByte großen Bereichs (Listing 3). Der Kernel richtet die Region an einer Pagegrenze (typischerweise 4 KByte) aus, also ergeben sich 256 unterschiedliche Startadressen (20 - 12 = 8, $2^8 = 256$). Theoretisch ließe sich die Entropie sowohl des Stacks als auch der Mmap-Region erhöhen, das ginge aber zu Lasten des zur Verfügung stehenden virtuellen Adressraums.

Eine besondere Rolle nimmt die VDSO-Page (Virtual Dynamic Shared Object) ein. Diese Seite ist für den schnellen Aufruf von Systemcalls zuständig [1]. Der Kernel blendete sie früher ans Ende des virtuellen Adressraums bei »0xffffe000« ein (siehe Abbildung 4). Zwischenzeitlich wurde diese Seite aber in die Mmap-Region verschoben, wodurch sie jetzt typischerweise eine zufällige Adresse erhält. Applikationen kommen mit einer zufälligen VDSO-Page nur zurecht, wenn eine aktuelle Glibc (Version 2.3.3 und neuer) installiert ist.

Daher hat sich Ubuntu für eine konservative Konfiguration entschieden. Mit

```
if grep vdso /proc/self/maps; then
    echo "feste vDSO-Page"
fi
```

findet der Admin durch eine Abfrage im »/proc«-Filesystem heraus, ob sich die VDSO-Page an der kompatiblen, festen Adresse oder in der Mmap-Region befindet. Mit der ersten Variante kommen auch ältere Programme zurecht.

Auf aktuellen Kerneln kann der Administrator beim Booten den Parameter »vdso = 1« übergeben und damit die VDSO-Page in die variable Mmap-Region

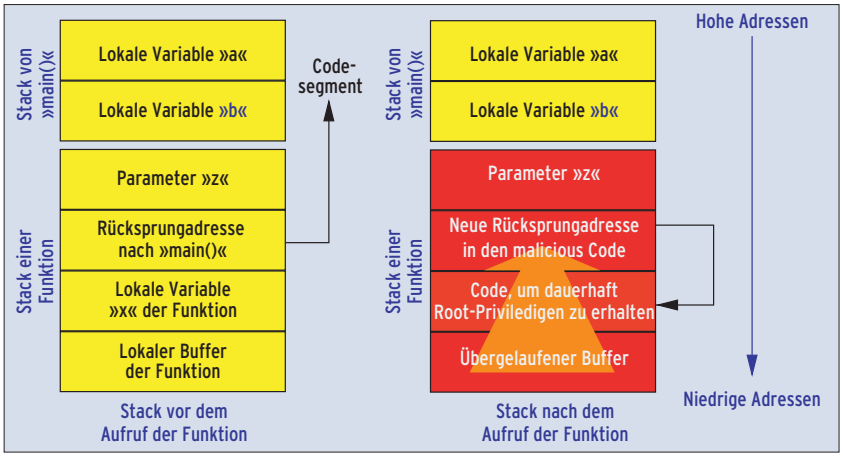


Abbildung 1: Gelingt es einem Angreifer, mehr Daten in einem Buffer auf dem Stack abzulegen, als der Programmierer dafür vorgesehen hat, kann er beim Buffer Overflow die gültige Rücksprungadresse überschreiben. Beim Rücksprung führt die CPU dann mitunter bösartigen Code aus.

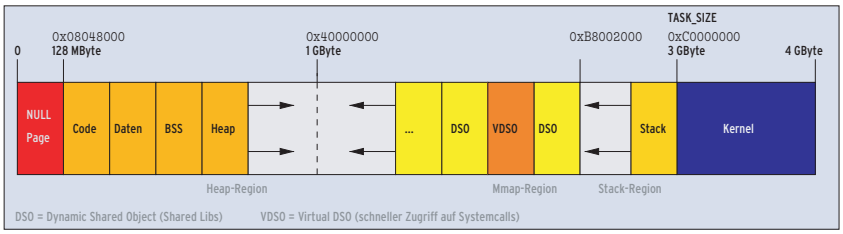


Abbildung 2: Im Normalfall ist die Einteilung des Adressraums auf einem 32-Bit-System der I-386-Architektur festgelegt. Heap und Stack wachsen von zwei Seiten aufeinander zu, da ihre Größe dynamisch wächst.

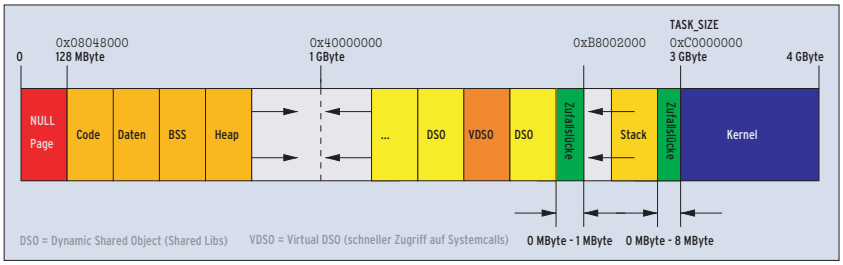


Abbildung 3: Aktiviertes ASLR erschwert es Angreifern, die Adresslage des Stacks und der Mmap-Region vorherzusagen. Vor dem oberen Ende des Stacks fügt der Kernel zufällige Leerräume ein.

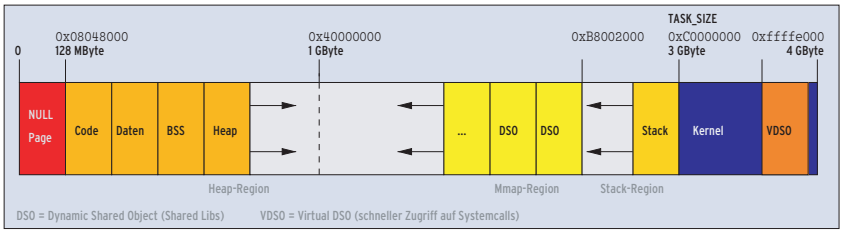


Abbildung 4: Im Kompatibilitätsmodus ist die VDSO-Page auf einem 32-Bit-Linux fest an die Adresse »0xffffe000« gebunden. Wird er deaktiviert, bekommt sie einen zufällen Platz in der Mmap-Region.

verschieben. Alternativ kann er das auch per Kernelkonfiguration erreichen, wenn er die Option »COMPAT_VDSO« deaktiviert (Abbildung 6).

Heap bleibt Heap

Gänzlich außen vor bleibt bei einem Standard-Linux der Heap-Bereich. Wer auch dessen Adresslage per Zufall variieren möchte, kommt um das Patchen des Kernels nicht herum. Das Pax-Team bietet dazu Vorschläge an [3]. Address-Space Layout Randomization (ASLR) – so der offizielle Name der vorgestellten Technik – lässt sich beim Starten des Kernels oder auch während des Betriebs ein- oder ausschalten. Beim Booten übergibt der Anwender dazu den Parameter »norandmaps« an den Kernel. Während des Betriebs steuert der Eintrag »/proc/sys/kernel/randomize_va_space« ASLR (siehe Abbildung 5).

Der Effekt von ASLR lässt sich anschaulich visualisieren. Das im Proc-Filesystem abgelegte Speicherlayout offenbart sich (als Hoffnungsschimmer für Angreifer) mit dem Befehl »cat /proc/self/maps«. Bei aktiviertem ASLR ändert sich die Adresslage, wie das untere Beispiel von Abbildung 5 zeigt.

Listing 1: »fs/binfmt_elf.c« (Auszug)

```
#ifndef STACK_RND_MASK                  /* 8MB of VA */
#define STACK_RND_MASK (0x7ff >> (PAGE_SHIFT - 12))
#endif

static unsigned long
randomize_stack_top(unsigned long stack_top)
{
    unsigned int random_variable = 0;

    if ((current->flags & PF_RANDOMIZE) &&
       !(current->personality & ADDR_NO_RANDOMIZE)) {
        random_variable = get_random_int() &
                          STACK_RND_MASK;
       random_variable <<= PAGE_SHIFT;
    }
```

Listing 2: »arch/i386/kernel/process.c« (Auszug)

```
unsigned long arch_align_stack(unsigned long sp)
{
    if (!(current->personality & ADDR_NO_RANDOMIZE)
        && randomize_va_space)
            sp -= get_random_int() % 8192;
    return sp & ~0xf;
}
```

Listing 3: »arch/i386/mm/mmap.c« (Auszug)

```
static inline unsigned long
mmap_base(struct mm_struct *mm)
{
   unsigned long gap =
        current->signal->rlim[RLIMIT_STACK].rlim_cur;
   unsigned long random_factor = 0;

   if (current->flags & PF_RANDOMIZE)
        random_factor = get_random_int() %
                        (1024*1024);

   if (gap < MIN_GAP)
        gap = MIN_GAP;
   else if (gap > MAX_GAP)
        gap = MAX_GAP;

   return PAGE_ALIGN(TASK_SIZE - gap - random_factor);
}
```

NX- und XD-Bits

Ein Standard-Linux hat zum Schutz vor Buffer Overflows noch mehr zu bieten, allerdings nur, wenn es auf einem Prozessor läuft, der nicht älter als drei oder vier Jahre ist. Neuere CPUs verfügen nämlich über ein zusätzliches Bit im Seitendeskriptor. AMD spricht vom No Execute (NX), Intel vom Execute Disable (XD) Bit. Mit ihnen markiert Linux Speicherseiten (Pages), die nur Daten und keinen Code enthalten.

Jeder Versuch, so markierten Code auszuführen, führt zu einer Exception (Abbildung 6). Diese Technik verhindert es, auf dem Stack abgelegten (bösartigen) Code auszuführen (Abbildung 7). Leider stehen das NX- oder das XD-Bit nur bei aktivierter PAE (Physical Address Extension) bereit. Wie dies festzustellen ist, beschreibt der Kasten „Stack-Execution per Hardware verbieten".

Um PAE und damit das NX-Bit zu aktivieren, muss der Admin bei der Kernelkonfiguration unter »Processor Type and Features« den »High Memory Support« entweder auf »off« oder auf »64 GByte« stellen. Ist das System mit maximal 1 GByte Hauptspeicher bestückt, sollte er zunächst »off« und dann die Option »PAE« aktivieren (Abbildung 8). Ist mehr Speicher vorhanden, empfiehlt sich die Auswahl »64 GByte«. Diese Wahl impliziert automatisch PAE und damit die Unterstützung des NX-Bits.

Bei der außerdem noch möglichen Auswahl »4 GByte« – gedacht für Systeme, die zwischen 1 und 4 GByte Hauptspeicher vorzuweisen haben – unterstützt der Kernel übrigens kein PAE! Natürlich muss der Admin den so konfigurierten Kernel anschließend noch übersetzen, installieren und neu booten.

Heile Welt

Eigentlich könnte der Kernel den Stack auf einer x86-Architektur mit 32 Bits auch ohne NX- oder XD-Bit schützen. Schließlich beinhalten die Segmentdeskriptoren seit Urzeiten vergleichbare Mechanismen. Es gibt unter dem Stichwort »SEGMEXEC« ein Patch des Pax-Teams, das diese Mechanismen nutzt [3]. Da es aber den virtuellen Adressraum reduziert, ist es bis dato nicht Teil des Standardkernels geworden. ▶

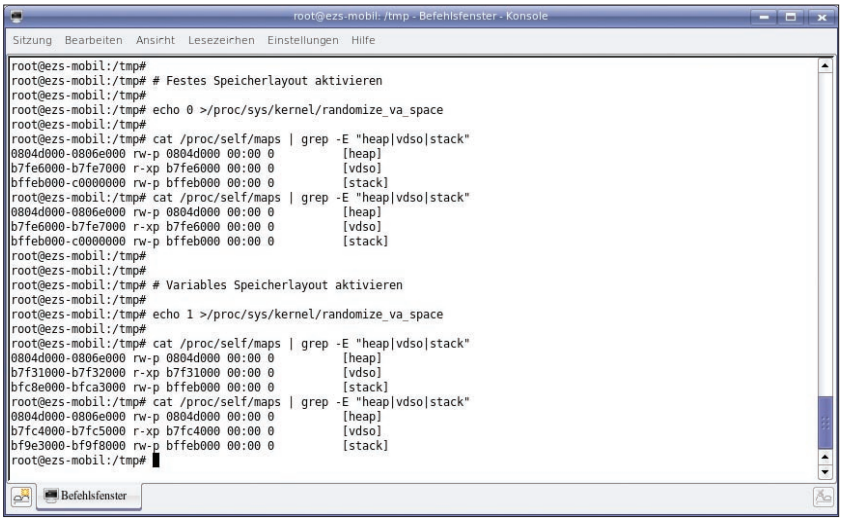
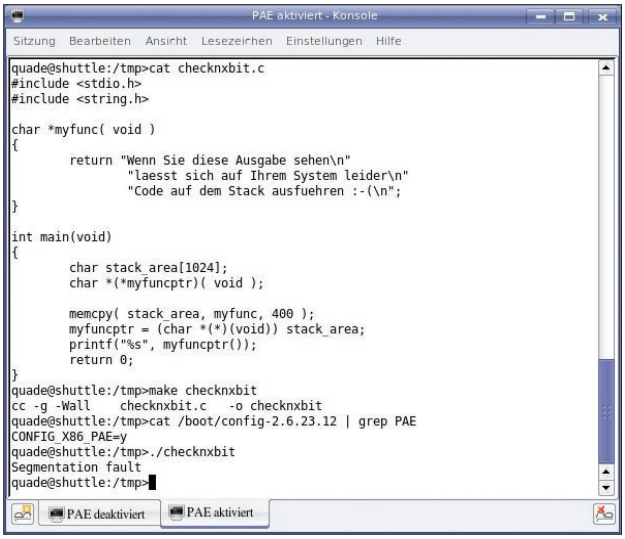
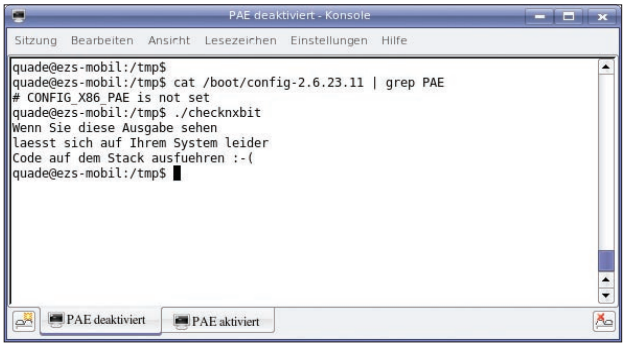


Abbildung 5: Bei aktiviertem ASLR liegt die VDSO-Page bei jedem Prozess an anderer Stelle im Hauptspeicher.





▲ **Abbildung 7: Diese Meldung bedeutet, dass Angreifer Code auf dem Stack ausführen und so etwas leichter die Kontrolle über das System gewinnen können.**

◀ **Abbildung 6: Mit diesem Programm lässt sich überprüfen, ob Code auf dem Stack ausführbar ist oder nicht. Erzeugt das Programm einen Segmentation-Fault, ist das ein gutes Zeichen.**

Es bleibt die Frage, was die vorgestellten Ansätze bringen und was sie kosten. Die von Linus Torvalds gewählte Lösung zum Schutz vor Buffer Overflows bleibt Applikationen und Anwendern weitgehend verborgen. Sie zwackt vergleichsweise wenig vom virtuellen Adressraum für die Zufallslücken ab. Performance-Einbußen sind gering, da der Kernel jeden Prozess beim Ausführen der »exec«-Funktionsfamilie randomisiert. Andererseits wird es zwar für den Angreifer schwieriger, einen Programmierfehler für seine Machenschaften auszunutzen, nicht aber unmöglich.

Shacham hat zusammen mit seinen Mitstreitern in einem Paper gezeigt, dass für einen effektiven Schutz eine Entropie von 8 Bit oder auch 20 Bit nicht ausreichend ist [2]. Anders sieht es übrigens in der 64-Bit-Welt aus: Hier genügt eine Entropie von 28 Bit für die Mmap-Region und ebenfalls 28 Bit für den Stack. Daran – so die Autoren – beißen sich Angreifer lange die Zähne aus, wenn sie durch Raten den Weg ins System finden wollen. Dass unter der 64-Bit-Version von Linux der Stack grundsätzlich mit dem NX-Bit geschützt ist, sollte Grund genug für jeden Systembetreiber der entsprechenden Architektur sein, den Linux-Kernel mit 64 Bit auch zu nutzen.

Schutzschilde aktivieren

Neben dem eingeschränkten Support durch die Prozessoren ist auch der Performance-Verlust bei aktiviertem PAE ein Grund für Distributoren, das NX-Bit zunächst ungenutzt zu lassen. Letztlich entscheidet jedoch der Benutzer, wie viel Sicherheit er braucht. *(mg)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik, Folge 13, Systemaufrufe“: Linux-Magazin 08/04

[2] Shacham et. al., „On the Effectiveness of Address-Space Randomization“: Proceedings of the 11th ACM conference on Computer and communications security, 2004, [http://www.stanford.edu/~blp/papers/asrandom.pdf]

[3] Patch des Pax-Teams: [http://pax.grsecurity.net/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux-Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Stack-Execution per Hardware verbieten

Neuere CPUs unterstützen das NX- (Intel) oder das XD-Bit (AMD). Ist diese Prozessorerweiterung vorhanden, kann der Standardkernel verhindern, auf dem Stack liegende Daten als Code zu interpretieren. Da es für die Bits im ursprünglichen Seitendeskriptor der x86-Architektur für 32-Bit-Systeme keinen Platz mehr gab, haben die Hersteller es an Position 63 des erweiterten Pagedeskriptors positioniert. Der Zugriff auf diesen Deskriptor ist im 32-Bit-Modus allerdings nur dann möglich, wenn die Physical Address Extension (PAE), mit der auch ein 32-Bit-System 64 GByte physikalischen Speicher adressieren kann, aktiv ist. Ob PAE unterstützt wird, zeigt ein Eintrag in »/proc/cpuinfo«:

```
if grep pae /proc/cpuinfo; then
    echo "PAE aktiviert"
fi
```

Ist PAE aktiv, nutzt der Kernel die Möglichkeiten der NX- oder XD-Bits aus. Overflow-Exploits haben auf diese Weise weniger Erfolg.

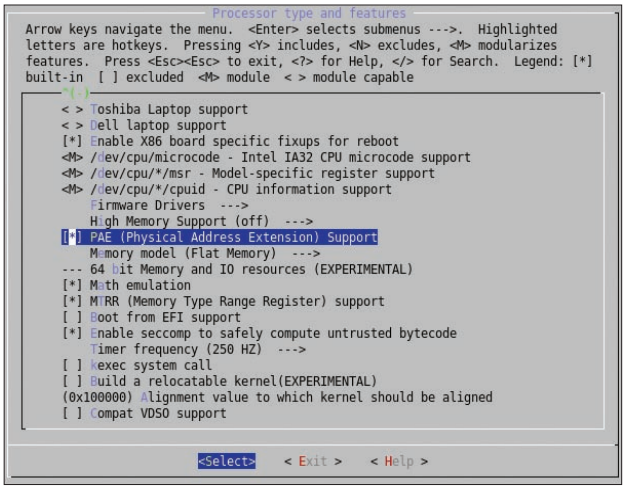


Abbildung 8: Wenn »High Memory Support« auf »off« steht, muss der Admin PAE explizit anwählen, um die Stack Protection zu aktivieren.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 39

Kern-Technik

RAM- und Temp-Filesysteme sparen Code, Hauptspeicher und Rechenzeit. Beste Voraussetzungen, um damit Linux beim Booten auf die Sprünge zu helfen. Eva-Katharina Kunst, Jürgen Quade



Einen richtigen Namen hat die Technik nicht. Von Early Userland und RAM-Filesystem sprechen die Entwickler. Der eigentlich unkorrekte Begriff Ramdisk beschreibt aber am besten, wie sie funktioniert: Der Computer legt Dateien im Hauptspeicher ab, ist so unabhängig von einer Festplatte und hat schnellen Zugriff auf sie. Gerade Entwickler von eingebetteten Systemen setzen schon lange auf Ramdisks, da sie helfen – genügend RAM vorausgesetzt – die Zahl der Zugriffe auf empfindliche Flashspeicher zu verringern, die nur eine begrenzte Anzahl Schreiboperationen erlauben. Für den Nutzer beinahe unbemerkt bootet heute kein aktuelles Linux-System mehr ohne RAM-Filesystem [1].

Ein in viele Module aufgeteiltes Linux unterliegt einem Henne-Ei-Problem, das erst das Early Userland löst: Treiber und Gerätedateien wie »/dev/console« liegen auf dem Hintergrundspeicher, der Kern benötigt die Treiber jedoch, um auf das Dateisystem zuzugreifen. Mindestens zwei Module sind nötig: Ein Treiber für den Hintergrundspeicher und eines, um das Dateisystem zu interpretieren, beispielsweise »sata« und »ext3«. Dieses Dilemma löst Linux via Virtual File System (VFS) und RAM-FS.

Ohne Treiber

Der Trick ist simpel: Ein Filesystemtreiber ist im Grunde eine Software, die spezifische Eigenschaften der einzelnen Dateisysteme auf das intern verwendete Virtual-Filesystem abbildet und damit dem Kernel ein einheitliches Bild vermittelt. Erst so kann Linux unterschiedliche Dateisysteme homogen als gemeinsamen Dateibaum darstellen.

Die dem VFS zugrunde liegenden Strukturen vermag der Kernel aber auch direkt aufzubauen. Dieser Überlegung folgend erlaubt es das Kernel-Buildsystem, später notwendige Treiber, Gerätedateien und Bootprogramme in ein CPIO-Archiv zu packen und es quasi huckepack an das Kernelimage zu kleben. So schaufelt der Bootloader beides in den Hauptspeicher. Da er und nicht der Kernel diese Arbeit durchführt, benötigt der Datentransport den Kernel gar nicht.

Sobald der Bootloader die Kontrolle an den Kernel überträgt, initialisiert dieser das Prozess- und Memory-Management und konstruiert aus dem im Hauptspeicher befindlichen Archiv das für den Betrieb notwendige Root-Filesystem (Abbildung 1, rechts). Dieser Weg ist viel eleganter, als einen Umweg über eine Ramdisk zu nehmen, bei der der Kernel Hauptspeicher einer bestimmten Größe reserviert, darin ein Filesystem erzeugt und in dieses schließlich Daten aus einem Archiv kopiert (Abbildung 1, links). Dass er stattdessen die Dateien direkt in die Kernel-internen Datenstrukturen des VFS umsetzt, spart Code, Speicherplatz und Rechenzeit.

Anders als eine Ramdisk, die immer eine feste Größe von 4, 8 oder 16 MByte hat, benötigt das RAM-Filesystem genau so viel Speicher wie die darin abgelegten Daten. Für eine Datei mit beispielsweise 12 KByte verwendet das Filesystem auch nur 12 KByte RAM. Legt der Anwender mehr Dateien an, wächst das RAM-Filesystem entsprechend mit. Sollten jedoch Amok laufende oder böswillige Threads ein solches Filesystem mit Daten fluten, kann das auch ein Nachteil sein.

Frühes Userland

Desktop- und Server-Linuxe nutzen das Early Userland, um dort Treiber für Festplatten- und Netzwerk-Controller und eventuell zugehörige Firmware abzulegen. Außerdem befindet sich dort ein Programm oder ein Shellskript mit dem Namen »init«, das der Kernel als ersten Prozess mit der PID 1 startet, wenn er das System hochfährt.

Dieses Gespann ermöglicht, zum Beispiel in einem festplattenlosen System, die Root-Partition über NFS per WLAN zu mounten. Der Init-Prozess lädt aus dem RAM-FS eventuell notwendige Firmware, den NFS- und den WLAN-Treiber. Damit ausgerüstet mountet er das neue Root-Filesystem mit Hilfe des Systemcalls »pivot_root«. Das kann durchaus auch wieder ein RAM-Filesystem sein (siehe Kasten „Schlanke Root-Filesysteme“). Jetzt darf er das RAM-Filesystem wieder aushängen und sich selbst per »excec()«

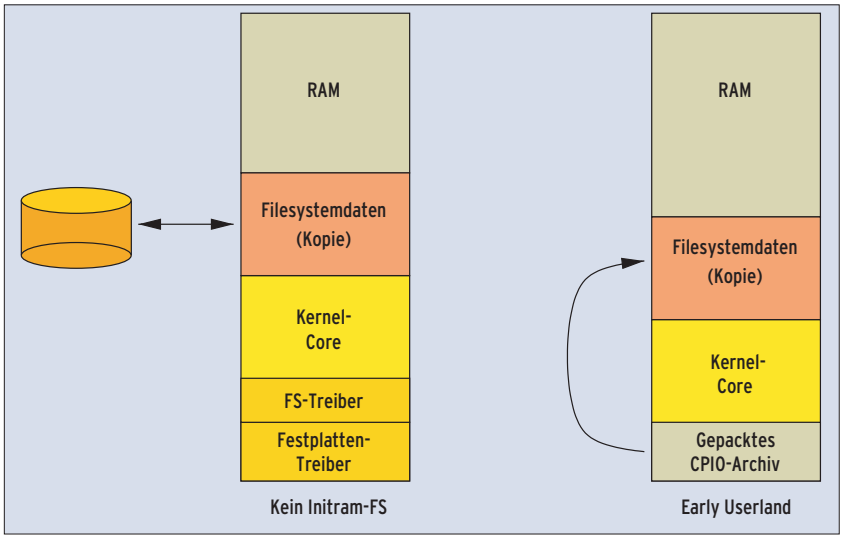


Abbildung 1: Das Early Userland (rechts) entnimmt die Daten seines Root-Filesystems einem komprimierten CPIO-Archiv. Den Speicher, den das Archiv belegt, gibt der Kernel wieder frei, sobald er sich vollständig initialisiert hat. Links ist im Vergleich ein herkömmlicher Filesystem-Zugriff auf die Ramdisk zu sehen.

in den neuen Init-Prozess verwandeln, dessen Programm auf dem neuen Root-Filesystem liegt (Abbildung 2). Das als Sprungbrett verwendete RAM-Filesystem ist für den Nutzer fortan unsichtbar.

Spezialfall: Embedded

Bei eingebetteten Systemen ist das Early Userland nicht das Mittel, sondern das Ziel. Embedded-Systeme müssen oft – der Einsatzumgebung geschuldet – ohne Festplatte auskommen und besitzen ersatzweise einen Flashspeicher. Auch moderne Flashsysteme lassen nur eine endliche Anzahl von Schreibzyklen zu. So erstaunt es manchen Architekten, wie leichtfertig viele Hersteller das Root-Filesystem auf dem Flashspeicher ablegen. Tatsächlich ist dies dank Early Userland nur in den Fällen notwendig, in denen Hauptspeicher Mangelware ist.

Stattdessen stellt der umsichtige Entwickler sein finales Root-Filesystem in Form eines Gzip-komprimierten CPIO-Archivs zusammen. Viele Bootloader sind in der Lage, es unabhängig vom Kernel in den Hauptspeicher zu laden. Besonders praktisch in der Embedded-Welt: Dieses Verfahren ermöglicht während der Entwicklung schnelle Testzyklen mit einfachen Buildverfahren.

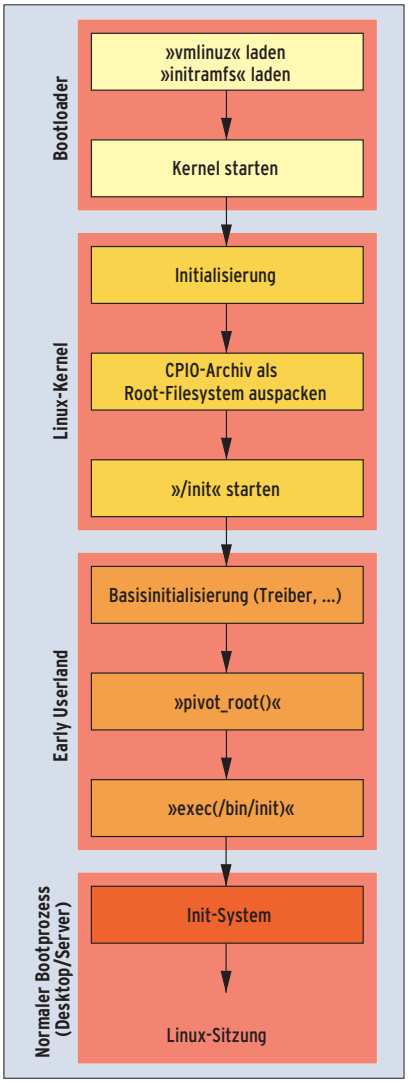


Abbildung 2: Desktop und Server nutzen das Early Userland als Zwischenstufe, bevor sie das eigentliche Init-System aktivieren.

Initrd-Archiv ein- und auspacken

Beim Booten füllt der Kernel das initiale RAM-FS mit Daten. Das beim Übersetzen des Kernels erzeugte und komprimierte CPIO-Archiv findet sich in den Kernelquellen unter ».../usr/initramfs_data.cpio.gz«. Ubuntu und Debian legen das Archiv typischerweise im Verzeichnis »boot« ab (»/boot/initrd.img-2.6.22-14-generic«). Um es zu inspizieren, legt der Admin am besten ein eigenes Verzeichnis an und führt folgende Kommandos aus:

```
mkdir /tmp/cpio
cd /tmp/cpio
zcat /usr/src/linux/usr/➥
initramfs_data.cpio.gz | ➥
 sudo cpio -i -d -H newc ➥
 --no-absolute-filenames
```

Das Archiv ist im neuen CPIO-Format gespeichert, das die Option »-H newc« erfordert. Da es Gerätedateien enthält, muss der Anwender Root-Rechte besitzen, um diese anzulegen. Weil »cpio« normalerweise absolute Pfadangaben verwendet, sollte der Admin es mit der Option »--no-absolute-filenames« aufrufen. Andernfalls könnte das Tool wesentliche Systemprogramme überschreiben und das System damit unbrauchbar machen.

Archiv für die Initrd erstellen

Um ein eigenes Archiv zusammenzustellen – wie es bei eingebetteten Systemen häufig notwendig ist –, hat Rob Landley Listing 1 »mkinitramfs« zur Verfügung gestellt, das aus einem Verzeichnis ein Archiv erzeugt. Wie er begleitend schreibt, ist das im Skript verwendete »find«-Kommando wichtig, damit der einfach gestrickte »cpio«-Code im Kernel das erzeugte Archiv auch richtig dekodiert.

Listing 1: »mkinitramfs«

```
01 #!/bin/sh
02 # Copyright 2006 Rob Landley <rob@landley.net> and
   TimeSys Corporation.
03 # Licensed under GPL version 2
04
05 if [ $# -ne 2 ]
06 then
07   echo "usage: mkinitramfs directory
   imagename.cpio.gz"
08   exit 1
09 fi
10
11 if [ -d "$1" ]
12 then
13   echo "creating $2 from $1"
14   (cd "$1"; find . | cpio -o -H newc | gzip) > "$2"
15 else
16   echo "First argument must be a directory"
17   exit 1
18 fi
```

Schlanke Root-Filesysteme

Root-Filesysteme, die der Kernel im Hauptspeicher instanziiert und die dort auch bleiben sollen, enthalten typischerweise nicht mehr Funktionalität als unbedingt notwendig. Schließlich steht jedes Byte, das das Root-Filesystem belegt, den Applikationen nicht mehr zur Verfügung. Wann immer es eine Auswahl gibt (etwa »ash« oder »bash«), empfiehlt sich die einfachere und schlankere Variante.

Abgelegte Software lässt sich noch entrümpeln, denn oft braucht ein initiales Root-Filesystem nicht alle Funktionen. Besonders hoch ist das Einsparpotenzial bei Bibliotheken, vor allem der Standard-C-Bibliothek. Da Shared Librarys mit dem notwendigen Framework viel Speicher belegen, setzen einige Entwickler – back to the roots – auf statisch gelinkte Applikationen (siehe Abbildung 5).

Da hierfür die »glibc« zu aufgebläht ist, haben die Kernelhacker eigene handliche Varianten entwickelt, die die wesentlichen Systemfunktionen beinhalten: Die »klibc« [4], »dietlibc« [5] und »newlib« [6] erfüllen auch ihren Zweck. Gerade Entwickler von Distributionen packen zusätzlich die notwendigsten Applikationen in ein Multi-Call-Binary wie beispielsweise »busybox«. Diese Applikation erledigt je nach dem Namen, mit dem sie der Anwender aufruft, unterschiedliche Aufgaben.

Der Kernel übergibt jeder Anwendung ihren Aufrufnamen (zum Beispiel aus der Shell) als ersten Parameter »argv[0]«. Dank Links haben manche Programme viele Namen und der Entwickler muss die Standardbibliothek nur einmal statisch an den Code binden. Mehr dazu erläutert ein eigener Artikel in diesem Heft.

Die Konfigurationsvariable »CONFIG_INITRAMFS_SOURCE« des Kernel-Buildsystems steuert, ob das Archiv vom Kernelimage separiert bleibt (Abbildung 3). Ist die Variable nicht belegt, erwartet der Kernel ein externes Archiv für das Root-Filesystem. Bei aktiviertem »CONFIG_BLK_DEV_INITRD« erzeugt der Kernel trotzdem ein Archiv, das aber leer bleibt. Der Kernel wächst dadurch nur um 157 Bytes gegenüber einem Kernel, der Initram-FS nicht unterstützt (Abbildung 4).

Kernel bauen

Unter Ubuntu und Debian erzeugt das Skript »update-initramfs« ein Archiv und legt es unter einem Namen wie »initrd.img-2.6.22-14-generic« in dem Verzeichnis »/boot« ab. Die Konfiguration ist in der Datei »/etc/initramfs-tools/initramfs.conf« zu finden. Wer beispielsweise die Anzahl der ins Archiv zu integrierenden Module reduzieren möchte, ist hier an der richtigen Stelle. Der Kasten „Initrd-Archiv ein- und auspacken“ erläutert, wie man das so entstandene Archiv inspiziert und ändert.

Alternativ gibt »CONFIG_INITRAMFS_SOURCE« an, welche Dateien ins initiale RAM-Filesystem gehören. Der Kernel erlaubt es, sie in einem Verzeichnis abzulegen oder sie per Liste anzugeben. Wer »CONFIG_INITRAMFS_SOURCE« mit einem Verzeichnisnamen belegt, überträgt beim Bau des Kernels den Inhalt des angegebenen Ordners ins Archiv. Andernfalls erwartet das Kernelskript »gen_initcpio« eine Dateiliste. Sie hat den Vorteil, dass für diesen Weg keine Superuser-Rechte notwendig sind.

Jede Zeile der Dateiliste bestimmt ein Element des Archivs. Die erste Spalte gibt dessen Typ an: »dir« für ein Verzeichnis, »nod« für ein Device-File, »slink« für symbolische Links und »file« für normale Dateien. Danach folgen der Name und bei Links und Files noch die Quelle. Die nächste Spalte listet die Zugriffsrechte in Oktal-Notation, danach Owner- und Gruppen-ID. Bei Device-Files stehen abschließend noch Gerätetyp (»b« oder »c«), Major- und Minor-Nummer.

Die Vorteile des RAM-Filesystems legen nahe, es auch für andere Zwecke zu nutzen, beispielsweise für temporäre Daten (»/tmp«). Dass das Filesystem keine

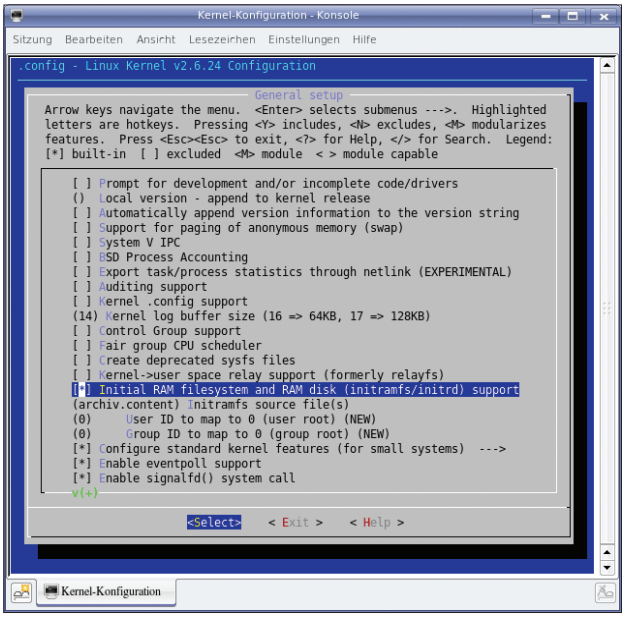


Abbildung 3: Mit aktivierter Option »Initial RAM filesystem...« unterstützt der Kernel das Root-Filesystem im RAM. Damit Linux ein generiertes CPIO-Archiv übernimmt, muss der Anwender zusätzliche »Initramfs source file(s)« – wie hier exemplarisch gezeigt – auswählen.

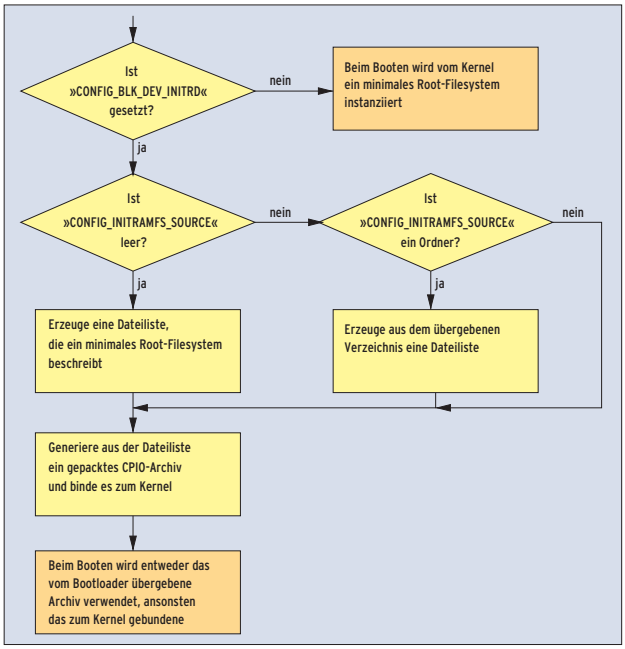


Abbildung 4: Die Kernelkonfiguration entscheidet durch die Variablen »CONFIG_BLK_DEV_INITRD« und »CONFIG_INITRAMFS_SOURCE«, ob sie ein gepacktes CPIO-Archiv erzeugt und mit welchen Inhalten sie es füllt.

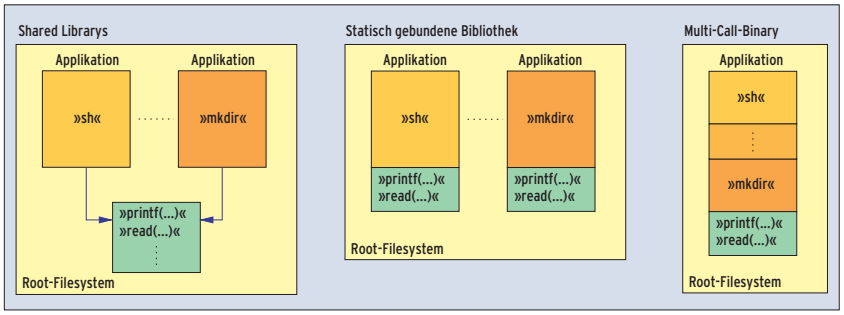


Abbildung 5: Für das Early Userland ist der mit Shared Librarys verbundene Speicherplatz-Overhead zu groß (links). Hier sind Mini-Applikationen gefragt, die schlanke Bibliotheken statisch linken (Mitte). Noch mehr Platz sparen Multi-Call-Binarys, die die Aufgabe mehrerer Programme in einem File vereinen.

Größenbegrenzung kennt, gerät jetzt allerdings zum Nachteil: Eine Denial-of-Service-Attacke flutet leicht den Hauptspeicher. Daher haben die Kernelentwickler mit dem »tmpfs« eine Variante programmiert, die ihre Größen überwacht. Standardmäßig beschränkt sie sich auf 50 Prozent des Hauptspeichers, per Mount-Option lassen sich aber auch andere Größen einstellen:

```
mount -t tmpfs -o size=512M,mode=1777 none ➥
  /tmp
```

Die Ausgabe von »mount« zeigt, dass der 2.6er Kernel »tmpfs« fleißig verwendet. Ubuntu etwa nutzt es für Verzeichnisse mit temporären Daten, etwa »/var/run«, »/var/lock« und »/dev/shm«. Insbesondere das heute dynamisch mit Inhalt gefüllte Verzeichnis »/dev« legt der Geräte-Daemon »udev« als »tmpfs« an und füllt es mit Gerätedateien.

Mit »tmpfs« ist die Variantenvielfalt virtueller Dateisysteme im Linux-Kernel bei Weitem nicht erschöpft. Auf Basis von Cram-FS lassen sich beispielsweise Dateien komprimiert im Hauptspeicher ablegen. Für eingebettete Systeme mit wenig Hauptspeicher ist das attraktiv. Allerdings hat die Sache einen Haken: Die Daten lassen sich nur lesen, aber nicht verändern.

Dass sich virtuelle Dateisysteme mit spezifischen Eigenschaften auch sehr leicht selbst entwickeln lassen, hat bereits eine frühere Kerntechnik [2] demonstriert. Den dort entwickelten und auf einen aktuellen Kernel (2.6.24) angepassten Quelltext bietet der FTP-Server des Linux-Magazins [3] zum Download an.

Es war einmal

Neben diversen virtuellen Filesystemen existiert weiterhin die klassische Ramdisk im Linux-Kernel. Die Komponenten, die das Root-Filesystem im RAM ermöglichen, befinden sich in den Kernelquellen an den verschiedensten Stellen. Tabelle 1 dient als Wegweiser.

Die klassische Ramdisk lässt sich aktivieren, indem der Admin bei der Kernelkonfiguration unter »Block Devices« den Punkt »Ram Disk Support« auswählt. Damit setzt er die Variable »CONFIG_BLK_DEV_RAM«. Ein so übersetzter Kernel stellt Gerätedateien für Blockdevices wie »/dev/ram0« oder »/dev/ram1« bereit. Bekannte Werkzeuge erlauben es, Dateisysteme zu erzeugen (formatieren) und zu mounten:

```
mke2fs /dev/ram0
mount /dev/ram0 /mnt
```

Im Hinblick auf die Vorteile der anderen RAM-Filesysteme steht jedoch zur Diskussion, welche Vorteile dieser Ahnvater der moderneren Subsysteme noch hat.

Nützlicher Helfer

Auf den ersten Blick erscheint die Vielfalt der RAM-Filesysteme verwirrend. Dass dies dem Kernel erlaubt von nahezu jedem Gerät zu booten, ist mehr als nur eine Entschädigung. Gerade für volatile Daten, die nur eine kurze Lebensdauer im Filesystem haben, sind sie ein nützlicher Helfer. Der Zugriff auf den Hauptspeicher läuft viel schneller als auf eine Festplatte – wenn überhaupt eine vorhanden ist – oder auf einen sonstigen Hintergrundspeicher. *(mg)* ■

Tabelle 1: RAM-FS und Ramdisks im Quellcode

Datei	Bedeutung
fs/ramfs/*	Quellcode zu »ramfs«
fs/cramfs/*	Quellcode zu »cramfs«
drivers/block/rd.c	Quellcode der Ramdisk
init/initramfs.c	Quellcode zum Auspacken des CPIO-Archivs
scripts/gen_initramfs.sh	Skript zum Erzeugen der Archiv-Dateiliste
usr/gen_cpio.c	Programm zum Erzeugen eines CPIO-Archivs aus einer Dateiliste
Documentation/early-userspace	Erläuterungen zum Kernelverzeichnis »usr«
Documentation/initrd.txt	Erläuterungen zu Initrd (1999/2000)
Documentation/ramdisk.txt	Erläuterungen zur alten Ramdisk (1995/2004)
Documentation/filesystems/ramfs-rootfs-initramfs.txt	Erläuterungen zu »ramfs«
Documentation/filesystems/cramfs	Erläuterungen zu »cramfs«
Documentation/filesystems/tmpfs	Erklärung zu »tmpfs« und dessen Mountparameter

Infos:

[1] Rob Landley, „Introducing initramfs, a new model for initial RAM disks“: [http://www.linuxdevices.com/articles/AT4017834659.html]

[2] Eva-Katharina Kunst und Jürgen Quade: „Kerntechnik, Folge 23“: Linux-Magazin 10/05, S. 90

[3] Sourcecode: [ftp://ftp.linux-magazin.de/pub/magazin/2008/05/Kerntechnik]

[4] Git-Archiv zu »klibc«: [http://git.kernel.org/?p=libs/klibc/klibc.git;a=summary]

[5] Wikipedia-Artikel zur Dietlibc: [http://en.wikipedia.org/wiki/Dietlibc]

[6] Newlib: [http://www.sourceware.org/newlib/]

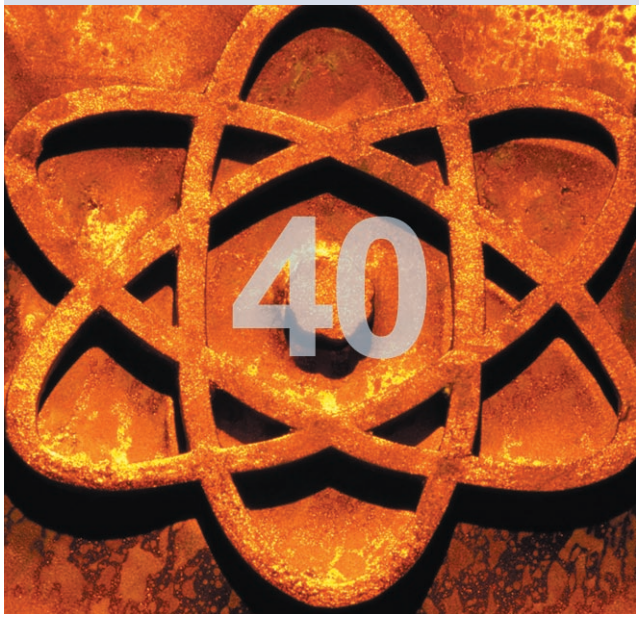
Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 40

Kern-Technik

Alternativ zum bekannten »ioctl()« setzen Userspace-Tools Routen und Firewall-Regeln per Netlink-Interface. Diese Methode eignet sich auch für die Kommunikation mit anderen Subsystemen. Eva-Katharina Kunst, Jürgen Quade



Viele Wege führen in den Kernel. Bekannt sind vor allem die Systemcalls und von denen die Funktionen »open()«, »close()«, »read()«, »write()« und »ioctl()«, die alle Daten aufnehmen oder zurückliefern. Die Unix-Urväter haben dem Aufruf »ioctl()« die Rolle einer Universalschnittstelle zugedacht, die Ein- und Ausgabegeräte steuert. Das war mit den übrigen Systemaufrufen nur schwer abzubilden.

Professionelle Programmierer mögen die Ioctls jedoch nicht. Zum einen sind die Aufrufe nicht portabel, zum anderen fehlerträchtig. Nachlässige Programmierer dokumentieren nur selten das Verhalten der Funktion im fehlerfreien und noch seltener im fehlerbehafteten Fall. Anwendungsentwickler gleichen Schlages verweigern ihrerseits hartnäckig das Lesen der Dokumentation.

Seit Langem schon gibt es auch andere, weniger bekannte Pfade ins Innere von Linux: Das Netlink-Interface [1] hat seinen Ursprung im Netzwerk-Subsystem. Hier dient es dazu, dem Kernel Konfigurationsdaten – etwa neue Routen – zu übergeben. Der umgekehrte Weg funktioniert ebenso. Über dieses Interface teilt der Kernel interessierten Applikationen mit, welche Routen er geändert hat.

Es geht aber noch weiter: Netlink ist eine Methode der Interprozess-Kommunikation (IPC), mit der Applikationen Daten mit sich selbst oder dem Kernel und der Kernel wiederum mit den Applikationen und ebenfalls mit sich selbst austauschen (siehe Abbildung 1).

Da das Interface zudem auf BSD-Sockets beruht, arbeitet es asynchron und unterstützt Multicasts. Ein einzelner Kernelthread kann daher gleich eine ganze Reihe von Prozesses mit identischen Daten versorgen. Und nicht zuletzt lassen sich auf diese Weise auch größere Datenmengen einfach transferieren. Angesichts dieser Vorteile verwundert es nicht, dass das Netlink-Interface auch außerhalb des Netzwerk-Subsystems zu Ruhm, Ehre und vor allem zum Einsatz gekommen ist.

Server im Kernel

Die angesprochene Kommunikation über Sockets folgt dem Client-Server-Modell. Im Fall von Netlink liegt der Server typischerweise im Kernel. Analog zur IP-Kommunikation benötigt auch Netlink einen Socket, allerdings wählt der Admin statt »PF_INET« hier »PF_NETLINK« (siehe Abbildung 2). Die Protokollfamilie »PF_NETLINK« hat zur Zeit 20 offizielle Mitglieder. Darunter befinden sich solche zum Handling von Routing-Informationen, zur Konfiguration der Firewall oder von IPsec. Die Headerdatei »netlink.h« listet die offiziellen Varianten auf.

Den adressierten Kommunikationspartner spricht der Entwickler über eine eindeutige Kennung an, typischerweise die Prozess-ID. Ein Netlink-Server im Kernel bindet seinen Socket an eine Adresse und hinterlegt eine Callback-Funktion. Das Subsystem ruft die Funktion auf, sobald eine Nachricht mit dem entsprechenden Protokoll und der Zieladresse eingeht. Die Adressen bei Netlink sind jeweils PIDs; wer den Kernel adressieren will, wählt die Null.

Flexibles Protokoll

Prinzipiell besteht eine Nachricht, die zu einem beliebigen Netlink-Protokoll gehört, aus einem Kommando und den zum Kommando gehörigen Attributen (siehe Abbildung 3). Attribute wiederum bestehen aus einem Attribut-Typ und dem Attribut-Wert. Typ kann beispielsweise Integer oder String, der zugehörige Wert 42 oder „Hello World“ sein. Makros, die alle den Namensvorsatz »nla_« tragen, dröseln die Nachricht in die verschiedenen Attribute auf.

Um einen Netlink-Server zu schreiben, definiert der Entwickler ein Netlink-

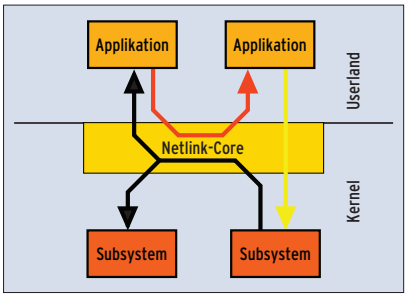


Abbildung 1: Netlink ermöglicht Kommunikation zwischen Komponenten aus Userland und Kernel.

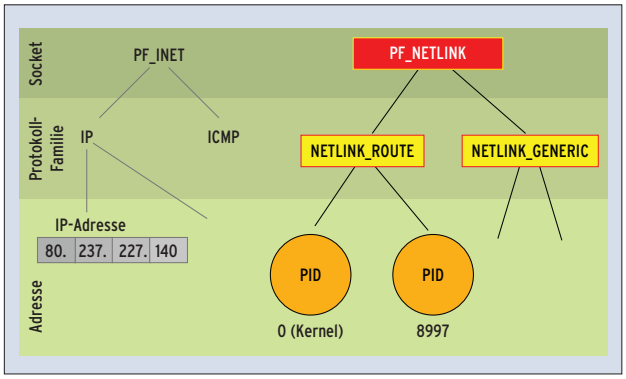


Abbildung 2: Das Schema der Adressierung folgt bei der Protokollfamilie Netlink dem der bekannteren Familien, etwa »PF_INET« mit seinen BSD-Sockets. Was dort IP-Adressen sind, entspricht bei Netlink Prozess-IDs.

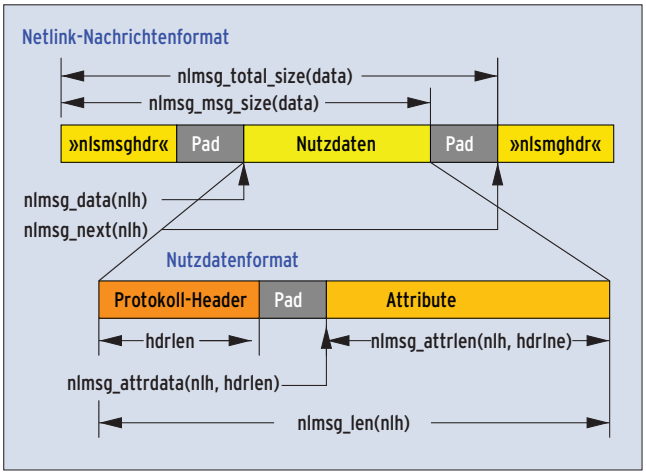


Abbildung 3: Eine Netlink-Nachricht besteht aus Header und Nutzdaten. Letztere können wiederum einen protokollspezifischen Header und die Attribute enthalten.

Protokoll. Für Experimente eignen sich »NETLINK_GENERIC« oder eine selbst erdachte Nummer wie in Listing 1, Zeile 10. Soll der Code in den offiziellen Kernel in »include/linux/netlink.h« aufgenommen werden, muss Linus Torvalds freilich ein entsprechendes Patch akzeptieren.

Warten auf Rückruf

Die Kernelfunktion »netlink_kernel_create()« alloziert einen Socket, bindet ihn an eine Adresse (0 für den Kernel) und hängt eine Callback-Funktion in das gewählte Netlink-Protokoll ein (Listing 1, ab Zeile 59). Der erste Funktionsparameter definiert den vor Kurzem in den Kernel eingeführten Namensraum für das Netzwerk, der in fast allen Fällen »&init_net« lautet. Weiter darf der Entwickler dem Netlink-Subsystem einen Mutex übergeben. Das Subsystem stellt so sicher, dass nur jeweils ein Prozess die Callback-Funktion durchläuft. Den Socket erhält der Programmierer als Rückgabe der Funktion und gibt ihn mittels »sock_release()« frei, sobald er das Modul entfernt.

Der Kernel ruft die Callback-Funktion ab Zeile 18 im Kontext des Nachrichtensenders auf. Innerhalb des Callback steht eine Reihe von Funktionen und Makros zur Verfügung (siehe Tabelle 1), um die eigentliche Nachricht auszulesen und um anschließend zu antworten: Die Funktion »nlmsg_hdr()« erhält die Adresse des Socket-Buffer und gibt einen Zeiger auf den Header der Netlink-Nachricht zurück (Zeile 28, siehe Abbildung 4).

Damit hat der Code Zugriff auf die PID des Absenders, das verwendete Kommando, seinen Nachrichten-Typ und auf die Sequenznummer (Listing 1, Zeilen 28 bis 32). Der Client inkrementiert die Sequenznummer mit jeder Abfrage. Es liegt nun bei der Callback-Funktion, die Daten auszuwerten. Dabei ist die Funktion »nlmsg_data()« behilflich. Genaue Angaben zu Prototypen und Erläuterungen liefert der Kernelquelltext in der Datei »include/net/netlink.h«.

Um zu antworten, benötigt der Kernelserver als Erstes einen Socket-Buffer, in dem er Header und Daten ablegt. Dafür reicht der Aufruf der Funktion »nlmsg_new()« in Zeile 35. Die Funktion erwartet zwei Parameter: Die Größe des für die Daten zu reservierenden Speicherbereichs – der Defaultwert »NLMSG_GOODSIZE« reicht meist aus – und ein Flag.

Tabelle 1: Auswahl wichtiger Netlink-Funktionen

Funktion	Beschreibung	Funktion	Beschreibung
Nachrichten aufbauen		**Zugriff auf die Payload**	
nlmsg_new()	Erzeugt eine neue Nachricht	nlmsg_data(nlh)	Adresse des Datenteils
nlmsg_put()	Fügt Netlink-Nachricht an einen »skb« an	nlmsg_len(nlh)	Länge des Datenteils
nlmsg_put_answer()	Callback-Variante von »nmlsg_put()«	nlmsg_attrdata(nlh, hdrlen)	Adresse des Attribut-Starts
nlmsg_end()	Abschließen der Nachricht	nlmsg_attrlen(nlh, hdrlen)	Länge der Attributdaten
nlmsg_get_pos()	Rückgabe der gegenwärtigen Position	**Message Parsing**	
nlmsg_trim()	Nachricht kürzen	nlmsg_ok(nlh, remaining)	Passt der Header in die restlichen Bytes?
nlmsg_cancel()	Abbruch	nlmsg_next(nlh, remaining)	Gibt die nächste Netlink-Nachricht zurück
nlmsg_free()	Freigabe der Netlink-Nachricht	nlmsg_parse()	Parsen der Nachrichtenattribute
Versenden von Nachrichten		nlmsg_find_attr()	Attributsuche
nlmsg_multicast()	Multicast	nlmsg_for_each_msg()	Schleife über alle Nachrichten
nlmsg_unicast()	Unicast	nlmsg_validate()	Validieren einer Nachricht
nlmsg_notify()	Senden einer Notification-Nachricht	nlmsg_for_each_attr()	Schleife über alle Attribute
Berechnung der Nachrichtenlänge		**Verschiedenes**	
nlmsg_msg_size(payload)	Nachrichtenlänge ohne Auffüll-Bytes (Pad)	nlmsg_hdr(skb)	Liefert die Startadresse des Netlink-Headers zurück
nlmsg_total_size(payload)	Gesamtlänge (mit Padding)		
nlmsg_padlen(payload)	Anzahl der Auffüll-Bytes	nlmsg_report()	Echo

Es regelt das Verhalten der Funktion für den Fall, dass nicht ausreichend Speicher zur Verfügung steht: »GFP_KERNEL« bedeutet, dass die Funktion schlafen darf, »GFP_ATOMIC« verbietet dies.

Um im neu erworbenen Socket-Buffer einen Netlink-Header anzulegen, ruft der Entwickler »nlmsg_put()« auf (Zeile 43). An dieser Stelle übergibt er auch die bereits erwähnte Sequenznummer und die eigene Absenderadresse. Außerdem übergibt er den Nachrichtentyp, den die Dokumentation oft auch als Kommando bezeichnet. Die ersten 16 Kommandos sind für Kontrollaufgaben reserviert, wobei der Kernel zurzeit nur die aus Abbildung 4 verwendet. Flags spezifizieren den Bearbeitungsmodus detaillierter.

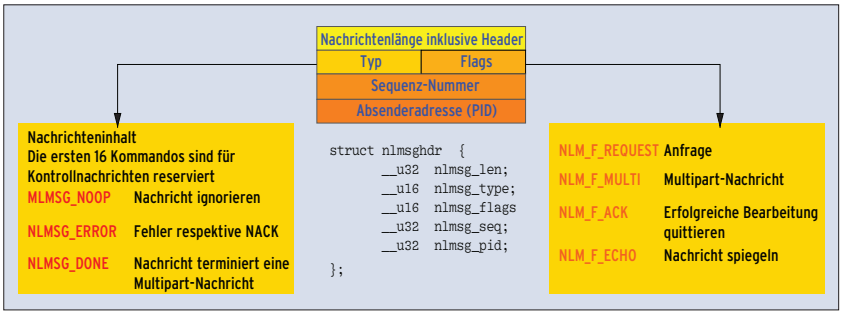


Abbildung 4: Der Netlink-Nachrichtenheader enthält neben der Absenderadresse auch das protokollspezifische Kommando. Die bitweise verknüpften Flags im Attribut »nlmsg_flags« steuern die Abläufe genauer.

Übergabe

Ist die Antwort vollständig aufgebaut, ruft der Programmierer »nlmsg_end()« auf und versendet sie mit »nlmsg_unicast()« (ab Zeilen 54). Diese Funktion erhält unter anderem die Adresse des Empfängers, also die zuvor im Empfangspaket gefundene PID. Der Funktionsname deutet bereits an, dass das Netlink-Interface auch den Versand von Multicast-Nachrichten zulässt. Das ist mit »nlmsg_multicast()« in der Tat möglich. Das Element »groups« in der Netlink-Adressenstruktur »sockaddr_nl« bestimmt, ob Linux die Nachricht an mehrere Empfänger auf einmal zustellt.

Auf der Client-Seite sind vier unterschiedliche Datenstrukturen zu initialisieren, falls der Code das bekannte Socket-Interface verwendet (siehe Abbildung 5). Zentral für die verbindungslose Socketkommunikation ist die Datenstruktur »struct msghdr« (Zeilen 58 bis 63). Damit lassen sich mehrere Datenpakete an einen Adressaten verschicken. Jedes Datenpaket spezifiziert dazu über die Datenstruktur »struct iovec« die Adresse und Anzahl der Pakete. Den eigentlichen Header mit Sequenznummer und Typ legt Netlink ebenso wie die Nutzdaten im Datenpaket ab.

Der Programmierer verpackt die Daten mit Hilfe der gleichen Makros zu Attributen, die im Kernel vorliegen. Es fehlen noch notwendige Adressangaben in »struct sockaddr_nl«. Die eigene Adresse bindet sich per »bind()« an den durch »socket()« allozierten Socket (Listing 2, Zeile 31). In der Struktur »struct msghdr« gibt es einen Verweis auf den Empfänger. Per »sendmsg()« gelangen die Daten in den Kernel (Zeile 63).

Zum Empfang benötigt die Applikation ausreichend Datenpuffer. Am einfachs-

Listing 1: Netlink-Server »nltest.c«

```
01 #include <linux/socket.h>
02 #include <linux/kernel.h>
03 #include <linux/module.h>
04 #include <linux/netlink.h>
05 #include <net/sock.h>
06 #include <net/tcp_states.h>
07
08 /* Sollte langfristig in
09    .../include/linux/netlink.h stehen */
10 #define NETLINK_TEST 25
11
12 #define MY_NL_COMMAND 0x11
13 #define MY_MESSAGE "Answer from Kernel"
14
15 static struct sock *nl_sk = NULL;
16 static DEFINE_MUTEX(nltest_mutex);
17
18 static void
19 nltest_netlink_server(struct sk_buff *skb)
20 {
21   struct nlmsghdr *nlh, *hdr;
22   u32 pid;
23   struct sk_buff *ret_skb;
24   char *data_ptr;
25
26   printk("nltest_netlink_server(%p)\n", skb);
27   BUG_ON(!skb);
28   nlh = nlmsg_hdr(skb);
29   pid = nlh->nlmsg_pid;
30   printk("pid: %d, seq: %d, data: \"%s\"\n",
31          pid, nlh->nlmsg_seq,
32          (char *)nlmsg_data(nlh));
33
34   /* Antwort verschicken  */
35   ret_skb = nlmsg_new(NLMSG_GOODSIZE,
26                       GFP_KERNEL);
37   if (ret_skb == NULL) {
38     printk("%s: no mem\n", __FUNCTION__);
39     return;
40   }
41
42   /* Header vorbereiten */
43   hdr = nlmsg_put(ret_skb, 0,
44                   nlh->nlmsg_seq, NLMSG_DONE,
45                   strlen(MY_MESSAGE), 0);
46   if (IS_ERR(hdr)) {
47     printk("nlmsg_put failed\n");
48     nlmsg_free(ret_skb);
49     return;
50   }
51
52   data_ptr = nlmsg_data(hdr);
53   strcpy(data_ptr, MY_MESSAGE);
54   nlmsg_end(ret_skb, nlh);
55   netlink_unicast(nl_sk, ret_skb, pid,
56                   MSG_DONTWAIT);
57 }
58
59 static int
60 __init nltest_module_init(void)
61 {
62   printk("nltest_module_init\n");
63   nl_sk = netlink_kernel_create(&init_net,
64                 NETLINK_TEST, 0,
65                 nltest_netlink_server,
66                 &nltest_mutex, THIS_MODULE);
67   return 0;
68 }
69
70 static void
71 __exit nltest_module_exit(void)
72 {
73   printk("nltest_module_exit\n");
74   if (nl_sk)
75     sock_release(nl_sk->sk_socket);
76 }
77
78 module_init(nltest_module_init);
79 module_exit(nltest_module_exit);
80 MODULE_LICENSE("GPL");
```

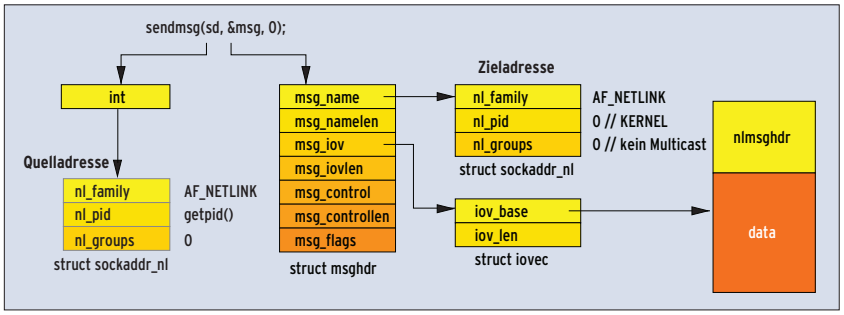


Abbildung 5: Für den Aufruf von »sendmsg()« benötigt des Netlink-Subsystem diverse Datenstrukturen. Die Makros aus dem Kernel darf der Programmierer auch im Userspace zum Aufbau der Nachrichten verwenden.

ten lassen sich bereits verwendete Puffer recyceln. Dazu reicht sie der Code an die Funktion »recvmsg()« weiter. Sobald die Antwort da ist, meldet sich die Funktion zurück. Ist der Transfer komplett, gibt die Applikation den Socket durch die Funktion »close()« wieder frei.

Nützliche Bibliothek

Während sich das Socket-Interface für einfache Applikationen eignet, greifen Entwickler bei umfangreicheren Aufgaben auf die Netlink-Library »libnl« zurück [2]. Sie stellt ein Interface ähnlich dem des Kernels zur Verfügung. Anregungen zur Programmierung der Userspace-Seite finden sich in »accounting/getdelays.c« des Dokumentationsverzeichnisses des Kernels und in einem Beitrag von Asanga Udugama [3]. Die in Listing 1 und Listing 2 beschriebenen Programme für erste Schritte mit Netlink setzen einen Kernel 2.6.24 oder neuer voraus. Die Beispiele erklären allerdings nicht die Attributverarbeitung.

Ein Makefile hilft den Code zu kompilieren. Im Linux-Quelltext finden sich Beispiele, einzig die Variable »obj-m« muss der Entwickler auf »nltest.o« setzen. Experimentierwillige erzeugen mit »make« und »make nltestappl« Modul und Anwendung. Das entstehende Modul laden sie mit »insmod nltest.ko« in den Kernel und starten schließlich mit »./nltestappl« die zugehörige Anwendung. Sie schickt dem Kernel eine Nachricht, der seinerseits darauf antwortet. Das Syslog visualisiert die Vorgänge im Kernel. Es lässt sich mit »tail -n3 /var/log/kern.log« anzeigen. Haben Userspace und Kernel erfolgreich Daten ausgetauscht, lässt sich das Beispiel einfach erweitern, um etwa die Daten auch für eigene Entwicklungen zu nutzen. *(mg)* ■

Infos

[1] RFC 3549, „Linux Netlink as an IP Services Protocol“: [http://www.ietf.org/rfc/rfc3549.txt]
[2] Netlink Library »libnl«: [http://people.suug.ch/~tgr/libnl/]
[3] Asanga Udugama, „Manipulating the Networking Environment Using RTNETLINK“: [http://linuxjournal.com/article/8498]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 2: Netlink-Client »nltestappl.c«

```
01 #include <stdio.h>
02 #include <memory.h>
03 #include <malloc.h>
04 #include <unistd.h>
05 #include <bits/sockaddr.h>
06 #include <linux/netlink.h>
07 #include <sys/socket.h>
08
09 /* Sollte in langfristig in
10    .../include/linux/netlink.h stehen */
11 #define NETLINK_TEST 25
12
13 #define MY_NL_COMMAND 0x11
14 #define MAX_PAYLOAD 1024
15
16 int main(int argc, char **argv)
17 {
18   struct sockaddr_nl src_addr, dst_addr;
19   struct nlmsghdr *nlh = NULL;
20   struct msghdr msg;
21   struct iovec iov;
22   int sock_fd;
23
24   /* Eigenen Endpoint anlegen */
25   sock_fd = socket(PF_NETLINK,
26             SOCK_DGRAM, NETLINK_TEST);
27   memset(&src_addr, 0, sizeof(src_addr));
28   src_addr.nl_family = AF_NETLINK;
29   src_addr.nl_pid = getpid();
30   src_addr.nl_groups = 0; /* Unicast */
31   bind(sock_fd, (struct sockaddr*)&src_addr,
32        sizeof(src_addr));
33
34   /* Zieladresse vorbereiten */
35   memset(&dst_addr, 0, sizeof(dst_addr));
36   dst_addr.nl_family = AF_NETLINK;
37   dst_addr.nl_pid = 0; /* 0: Kernel */
38   dst_addr.nl_groups = 0; /* Unicast */
39
40   /* Netlink-Header erzeugen */
41   nlh = (struct nlmsghdr *)calloc(1,
42                 NLMSG_SPACE(MAX_PAYLOAD));
43   memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
44   nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
45   nlh->nlmsg_pid = getpid();
46   printf("Absender: %d\n", nlh->nlmsg_pid);
47   nlh->nlmsg_flags = NLM_F_REQUEST|
48                      NLM_F_ECHO;
49   nlh->nlmsg_type = MY_NL_COMMAND;
50
51
52   /* Daten einpacken */
53   strcpy(NLMSG_DATA(nlh), "Hello World");
54
55   iov.iov_base = (void *)nlh;
56   iov.iov_len = nlh->nlmsg_len;
57
58   memset(&msg,0,sizeof(msg));
59   msg.msg_name = (void *)&dst_addr;
60   msg.msg_namelen = sizeof(dst_addr);
61   msg.msg_iov = &iov;
62   msg.msg_iovlen = 1; /* Anzahl */
63   sendmsg(sock_fd, &msg, 0);
64
65   /* Netlink wartet auf eine eingehende
66     Nachricht vom Kernel */
67   printf("Warte auf Antwort...\n");
68   recvmsg(sock_fd, &msg, 0);
69   printf("Antwort:\n\tflags %d\n"
70          "\ttype %d\n\t%s\n",
71          nlh->nlmsg_flags,
72          nlh->nlmsg_type,
73          (char *)NLMSG_DATA(nlh));
74   close(sock_fd);
75   return 0;
76 }
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 41

Kern-Technik

Der Kernel bietet auf Multicore-Maschinen eine ganze Reihe von Methoden an, um Prozesse auf CPUs zu verteilen. Für alle braucht er die Mithilfe des Admin. So gewappnet erfüllt Linux auch harte Echtzeitanforderungen – wenn die betroffenen Prozesse ausreichend CPU-Zeit bekommen. Eva-Katharina Kunst, Jürgen Quade



Multitasking im Linux-Kernel verhält sich ähnlich wie das Fotografieren: Im Automatik-Modus bringt jeder brauchbare Bilder zustande – ausreichende Lichtverhältnisse vorausgesetzt. Professionelle Fotografen dagegen, die mehr als Schnappschüsse erwarten, stellen selbst Blende, Verschlusszeit, Entfernung und Lichtempfindlichkeit ein und schaffen damit unter schwierigen Bedingungen stilvolle Bilder.

Mit der Hand gefertigt

Auch der Linux-Kernel kommt mit den meisten Lastsituationen prima zurecht, bietet aber zusätzlich Einstellschrauben, um auch schwierige Anforderungen zu erfüllen. Wer beeinflussen will, wie Linux Aufgaben auf die Rechnerkerne einer Multicore-Maschine verteilt, darf zwischen konkurrierenden Methoden wählen: CPU-Affinity, CPU-Hotplugging, CPU-Isolation und CPU-Sets. Das ist verwirrend und mancher Admin fragt sich zu Recht, welche Methoden seine Anforderungen am besten erfüllen.

Der Hintergrund: Mehrkern-Maschinen steigern einerseits die Verarbeitungsleistung und reduzieren andererseits die Latenzzeiten. Das erreicht der Linux-Scheduler allerdings nur, wenn er die anstehenden Aufgaben auch geeignet auf die mehrfach vorhandenen Verarbeitungseinheiten verteilt. Manchmal ist es hilfreich, ihm dabei unter die Arme zu greifen.

Das ist oft gar nicht so einfach, weil nicht nur Prozesse, sondern genau genommen Kernelthreads, Workqueues, Applikationsthreads ebenso wie Interrupts und Soft-IRQs inklusive Tasklets und Timer darauf warten, dass Linux ihnen eine CPU zuweist.

Zuneigung zeigen

Der Kernel verteilt die Threads auf Grundlage des Attributs »cpus_allowed«. Es ist Teil des Prozess-Control-Blocks (PCB) und für jeden Rechenprozess einmal vorhanden. Das Attribut »cpus_allowed« ist ein Bitfeld vom Typ »cpumask_t«, in dem je ein Bit eine CPU repräsentiert (siehe Abbildung 1). Ein gesetztes Bit teilt dem Scheduler mit, dass er die entsprechende CPU damit beauftragen darf, den Prozess zu bearbeiten. Eine Null unterbindet jegliche Verarbeitung.

Neue per »fork()« respektive »clone()« erzeugte Jobs erben das Attribut von ihren Eltern. Vom Userland aus erhalten Anwender durch Aufruf von »sched_getaffinity()« Auskunft über das Attribut, über die Makros »CPU_ZERO()«, »CPU_CLR()« und »CPU_SET()« modifizieren sie es oder setzten den Wert direkt mit dem Systemaufruf »sched_setaffinity()« [1].

Gruppenbildung

Etwas eleganter lassen sich Threads zu einer CPU durch CPU-Sets zuordnen: Sie sind zwar seit Version 2.6.12 im Standardkernel enthalten, basieren aber seit Kernel 2.6.24 auf dem allgemein einsetzbaren Framework Control Groups, das ehemals Container hieß. Control Groups (Cgroups) bilden einen Rahmen, in dem Anwender Rechenprozesse und ihre Nachkommen, die spezifische Eigenschaften besitzen, hierarchisch organisieren. Mit diesem Mittel gruppiert der Admin über ein virtuelles Filesystem vom Typ »cpuset« sowohl Prozessoren als auch Speicherressourcen (Memory Nodes) und verteilt die Rechenprozesse auf die eingerichteten Gruppen.

Abbildung 2 zeigt, dass diese Verteilung hierarchisch aufgebaut ist. Die oberste Gruppe enthält sämtliche Prozessoren und Memory Nodes. In einer Ebene darunter lassen sich die Ressourcen auf mehrere Untergruppen verteilen. Per Flag »cpu-exclusive« und »memory-exclusive« lässt sich festlegen, ob mehrere Gruppen eine Ressource (überlappend) verwenden dürfen oder ob diese exklusiv einer Gruppe zur Verfügung steht.

Verlagert der Kernel einen Rechenprozess in eine Gruppe, überschreibt er das Attribut »cpus_allowed« des verschobenen Prozesses. Er erbt auf diesem Wege die Einstellungen der Gruppe, da der Prozess die Affinity-Maske der Gruppe erhält. Das ist praktisch: Auf diese Weise lassen sich

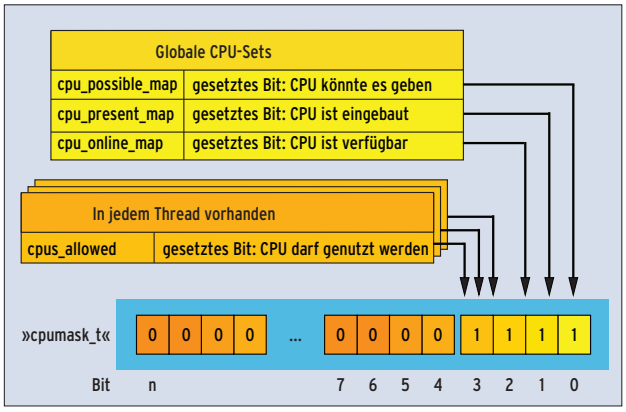


Abbildung 1: Der universelle Datentyp »cpumask_t« verwaltet die Rechnerkerne mit einer Bitmaske. Sie legt als Attribut »cpus_allowed« des PCB für jeden Thread fest, welche Kerne online sind und dem Thread zur Verfügung stehen.

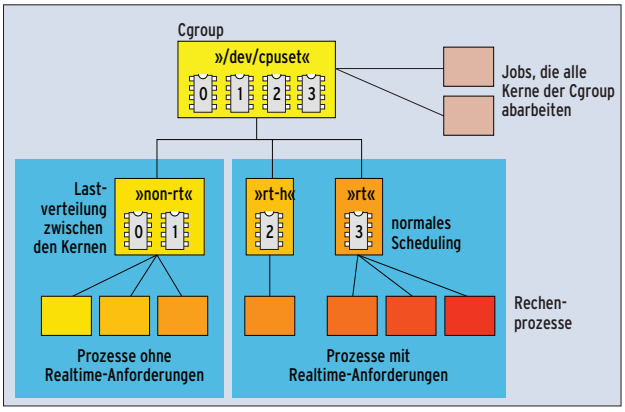


Abbildung 2: CPU- und Memory-Ressourcen lassen sich über Cgroups hierarchisch verwalten. Solange es kein gesetztes »exclusive«-Flag besitzt, sind Jobs einer übergeordneten Cgroup allen Ressourcen der Gruppe zugeordnet.

einfach und dauerhaft die zeitkritischen Prozesse von den weniger anspruchsvollen Prozessen trennen.

Ein Beispiel zeigt Listing 1: Im einfachsten Fall legt der Admin etwa die zwei Gruppen »rt« und »non-rt« (Zeilen 3 und 4) an, weist ihnen die Speicherressourcen und die Prozessorkerne zu (Zeilen 5 bis 10) und verschiebt sämtliche Prozesse in die »non-rt«-Gruppe (»for«-Schleife ab Zeile 11). Anschließend pickt er sich die zeitkritischen Rechenprozesse heraus und migriert diese in die »rt«-Gruppe (Zeilen 14 bis 16).

Wer die Prozesse einmal verteilt hat, sollte davon absehen, nachträglich die Affinity-Maske einer Gruppe zu ändern. Das wirkt sich nämlich auf den innerhalb der Gruppe befindlichen Prozess nicht mehr aus. Um die Parametrierung wirksam zu verändern, müsste der Admin den Rechenprozess der entsprechenden Gruppe neu zuordnen. Ein Prozess kann aber über die Funktion »sched_setaffinity()« aus dem Container-Gefängnis Cgroup weder ausbrechen noch darin eindringen. Nur wenn ein Prozessorkern zu der Gruppe gehört, darf der Prozess dorthin migrieren.

Schiebung

Verschiebt ein Anwender Prozesse in die Gruppe »non-rt«, schaufelt er den nicht benutzen Kern für Realzeitaufgaben frei. Das lässt sich einfacher per CPU-Hotplugging bewerkstelligen. Eigentlich heißt Hotplugging, dass Anwender während des Betriebs Hardware hinzufügen oder entfernen; in diesem Fall ist einer von mehreren Prozessoren gemeint. Das sollten allerdings glückliche Besitzer von Mehrprozessormaschinen nicht wörtlich nehmen und während des Betriebs den Rechner aufschrauben. Echtes Hardware-CPU-Hotplugging unterstützen nämlich nur wenige Servertypen.

Unabhängig davon können Admins aber Software-technisch (logisch) einzelne Prozessoren (und damit auch Prozessorkerne) deaktivieren und wieder aktivieren. Schalten sie eine CPU ab, migriert Linux dessen Prozesse auf andere Kerne. Dieses Feature setzten Admins dann sinnvoll ein, wenn sie den Verdacht haben, dass ein einzelner CPU-Kern defekt ist. Die CPU 0 spielt als Bootprozessor eine Sonderrolle und lässt sich auf vielen Systemen nicht deaktivieren. Das hängt damit zusammen, dass bei diesen Architekturen einige Interrupts fest an den ersten Prozessorkern gekoppelt sind.

Kerne klarmachen

Abbildung 3 zeigt, wie sich durch Zugriffe auf das Sys-Filesystem eine CPU aktivieren und deaktivieren lässt. Sichtbar am Inhalt der Datei »/proc/interrupts« sind zunächst vier Kerne online. An der mit »ps axwu« erzeugten Prozesstabelle lassen sich die auf dieser CPU lokalisierten Kernelthreads entdecken. Sie tragen die CPU-Nummer im Namen, abgetrennt durch einen Schrägstrich. Deaktiviert der Anwender die CPU jetzt durch

```
echo "0" > online
```

dann taucht die CPU in »/proc/interrupts« nicht mehr auf. Auch die prozessorspezifischen Kernelthreads sind verschwunden. Fehlt bei einer CPU der Dateieintrag »online« im virtuellen Filesystem, lässt sie sich auch nicht deaktivieren.

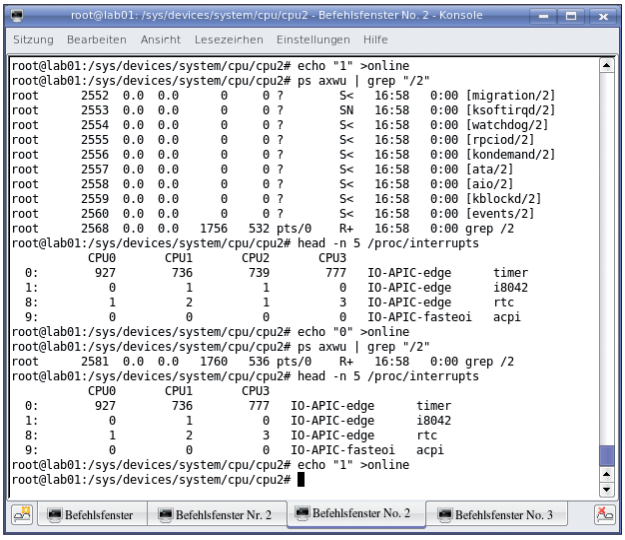
Listing 2 zeigt anhand einer Callback-Funktion, wie sich Kernelcode darüber informiert, ob eine CPU wegfällt oder eine neue den Dienst aufgenommen hat. Über die Kernelfunktion »register_hotcpu_notifier()« meldet das ladbare Kernelmodul in Zeile 29 die ab Zeile 5 implementierte Funktion »my_cpu_callback()« an. Sobald sich etwas am CPU-Status ändert, informiert die Funktion darüber. Die Ausgabe lässt sich mit »dmesg | tail« beobachten.

Weggegangen, Platz vergangen

Wer übrigens eine CPU logisch entfernt, die zu einem CPU-Set gehört, streicht diese auch aus dem CPU-Set. Daher kehrt

Listing 1: Realtime-Affinität zuweisen

```
01 mkdir /dev/cpuset
02 mount -t cpuset -ocpuset cpuset /dev/cpuset
03 mkdir /dev/cpuset/rt
04 mkdir /dev/cpuset/non-rt
05 echo 0 > /dev/cpuset/rt/mems
06 echo 0 > /dev/cpuset/non-rt/mems
07 echo 2 >/dev/cpuset/rt/cpus
08 echo 1 >/dev/cpuset/rt/cpu_exclusive
09 echo 0-1 >/dev/cpuset/non-rt/cpus
10 echo 1 >/dev/cpuset/rt/cpu_exclusive
11 for pid in $(cat /dev/cpuset/tasks)
12 do
13     echo $pid > /dev/cpuset/non-rt/tasks
14 done
15 cd /dev/cpuset/rt
16 echo $$ > tasks
17 starte_rt_task
```



◀ Abbildung 3: Durch einfache Schreibzugriffe auf das Sys-Filesystem lassen sich per Kommandozeile einzelne Rechnerkerne aktivieren und deaktivieren. Die virtuelle Datei »online« steuert dies. Hier schickt der Admin eine Null an die CPU 2, die daraufhin ihre Kernelprozesse beendet und zusätzlich Interrupts auf die noch aktiven Kerne umleitet.

sie nicht automatisch wieder in das CPU-Set zurück, wenn der Anwender sie wieder aktiviert; er muss sie aktiv mit Hilfe des Sys-Filsystems wieder aufnehmen. Ebenfalls spannend ist die Frage, wie der Kernel mit einem Rechenprozess verfährt, der fest an eine offline zu nehmende CPU gebunden ist: Der Kernel geht bei Wegfall der den Prozess bearbeitenden CPU recht pragmatisch vor und ändert die CPU-Maske so, dass der Prozess fortan auf den übrigen Prozessoren arbeitet.

Isolationshaft

Von den CPU-Sets grenzt sich die CPU-Isolation dadurch ab, dass sie bereits beim Booten einsetzt. Per Bootparameter »isolcpus« lassen sich einzelne Rechnerkerne mit Ausnahme der CPU 0 direkt deaktivieren und vom Load-Balancing, dem Verteilen der Last zwischen den vorhandenen Prozessoren, ausnehmen. Die Methode ist sehr rudimentär, bringt aber kürzere Bootzeiten mit sich. Ansonsten spricht wohl noch die einfache Handhabung für den Bootparameter.

Sollen auf einer Mehrkernmaschine mehrere Prozessoren zeitkritische Rechenprozesse abarbeiten, verteilt der Kernel – anders als bei den Cgroups – bei vorhandenem Bootparameter »isolcpus« keine Lasten innerhalb der isolierten Kerne. Trotzdem arbeitet Linux auch auf den auf diese Weise isolierten Rechnerkernen ISRs, Tasklets und Timer ab. Das lässt sich allenfalls durch den Bootparameter »max_cpus« verhindern. Insofern ist verständlich, dass bei den wenigen Vorteilen die Entwickler diskutieren, dieses Feature aus dem Kernel zu entfernen.

Unterbrechungen verhindern

Ähnlich wie Rechenprozesse besitzen auch Interrupts im Kernel ein Attribut namens »mask«. Es legt pro Prozess mit Hilfe einer Bitmaske fest, welcher Prozessor eine Interrupt-Serviceroutine ausführen darf. Bei den Hardware-Interrupts legt der Kernelprogrammierer beim Einhängen der Interrupt-Serviceroutine fest, ob die ISR „balanced“ sein soll oder nicht. Wählt er diese Eigenschaft ab, führt der Kernel die Routinen nur auf einer CPU aus.

Aus dem Userland manipuliert der Anwender das Attribut »mask« wieder einmal über das Proc-Filesystem:

```
cat /proc/irq/1/smp_affinity
0f
echo "0x0e" > /proc/irq/1/smp_affinity
cat /proc/irq/1/smp_affinity
0e
```

Dazu ist etwas Hex-Artithmetik nötig: In dem Beispiel nimmt der Anwender auf einer Vier-Kerne-Maschine – »smp_affinity« hat den Wert 0x0f, also sind vier

Listing 2: »hotplug.c«

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <linux/cpu.h>
04
05 static int __cpuinit my_cpu_callback(struct notifier_block *nfb,
06     unsigned long action, void *hcpu)
07 {
08     unsigned int cpu = (unsigned long)hcpu;
09
10     switch (action) {
11     case CPU_ONLINE:
12     case CPU_ONLINE_FROZEN:
13         printk("cpu 0x%x aktiviert\n", cpu);
14         break;
15     case CPU_DEAD:
16     case CPU_DEAD_FROZEN:
17         printk("cpu 0x%x deaktiviert\n", cpu);
18         break;
19     }
20     return NOTIFY_OK;
21 }
22
23 static struct notifier_block my_cpu_notifier = {
24     .notifier_call = my_cpu_callback;
25 };
26
27 static int __init mod_initialize(void)
28 {
29     register_hotcpu_notifier(&my_cpu_notifier);
30     printk("cpu_notifier angemeldet\n");
31     return 0;
32 }
33
34 static void __exit mod_release(void)
35 {
36     unregister_hotcpu_notifier(&my_cpu_notifier);
37     printk("cpu_notifier abgemeldet\n");
38 }
39
40 module_init( mod_initialize );
41 module_exit( mod_release );
42 MODULE_LICENSE("GPL");
```

Bits gesetzt – die CPU 0 von der Verarbeitung des Interrupts 1 aus, indem er das niederwertigste Bit löscht und »smp_affinity« nun den Wert 0x0e zuweist.

Weiche Interrupts

Wie die vorhandenen Prozessoren die Threads und Interrupts abarbeiten, lässt sich mit den beschriebenen Methoden gut kontrollieren. Das sind aber nicht die einzigen Arbeiter im Linux-Kernel: Soft-IRQs, Tasklets und Timer, aber auch Workqueues sind anders gelagert. Typischerweise arbeitet jene CPU Soft-IRQs und Tasklets ab, auf der auch die Interrupt-Serviceroutine lief.

Daher muss der Systemarchitekt dafür sorgen, dass er die IRQ-Affinität für Prozesse mit harten Echtzeitanforderungen auf dafür reservierte Kerne lenkt. Da heutzutage mehrere Hardwarekomponenten oft einen Interrupt gemeinsam nutzen, darf nur ein Kern die zugehörige Interruptleitung nutzen. Das ist bei vielen Architekturen nur durch einen Eingriff in das PCI-Handling möglich und erschwert die Portabilität.

Tuning schafft Performance

Das ist auch die Quintessenz, die aus einer guten und einer schlechten Nachricht besteht. Die gute: Dank der manuellen Kontrolle über das Load-Balancing lässt sich ein Standard-Linux-Kernel mit handelsüblichen Mehrkernprozessoren für Anwendungen mit harten Echtzeitanforderungen einsetzen. Entwickler berichten auf der Linux-Kernel-Mailingliste von Worst-Case-Latenzzeiten im einstelligen Mikrosekundenbereich [2].

Die schlechte Nachricht: Die guten Latenzzeiten erreicht Linux nicht von selbst, vielmehr muss der Systemarchitekt seine Maschine gut kennen und geeignet konfigurieren. Dazu bietet der Kernel gleich mehrere, konkurrierende Methoden. Trotz etwas manuellem Mehraufwand in der Konfiguration kristallisiert sich dabei ein Favorit heraus: Je mehr Kerne eine Maschine besitzt, desto weniger kommt der verantwortliche Admin an den CPU-Sets vorbei. *(mg)* ■

Infos

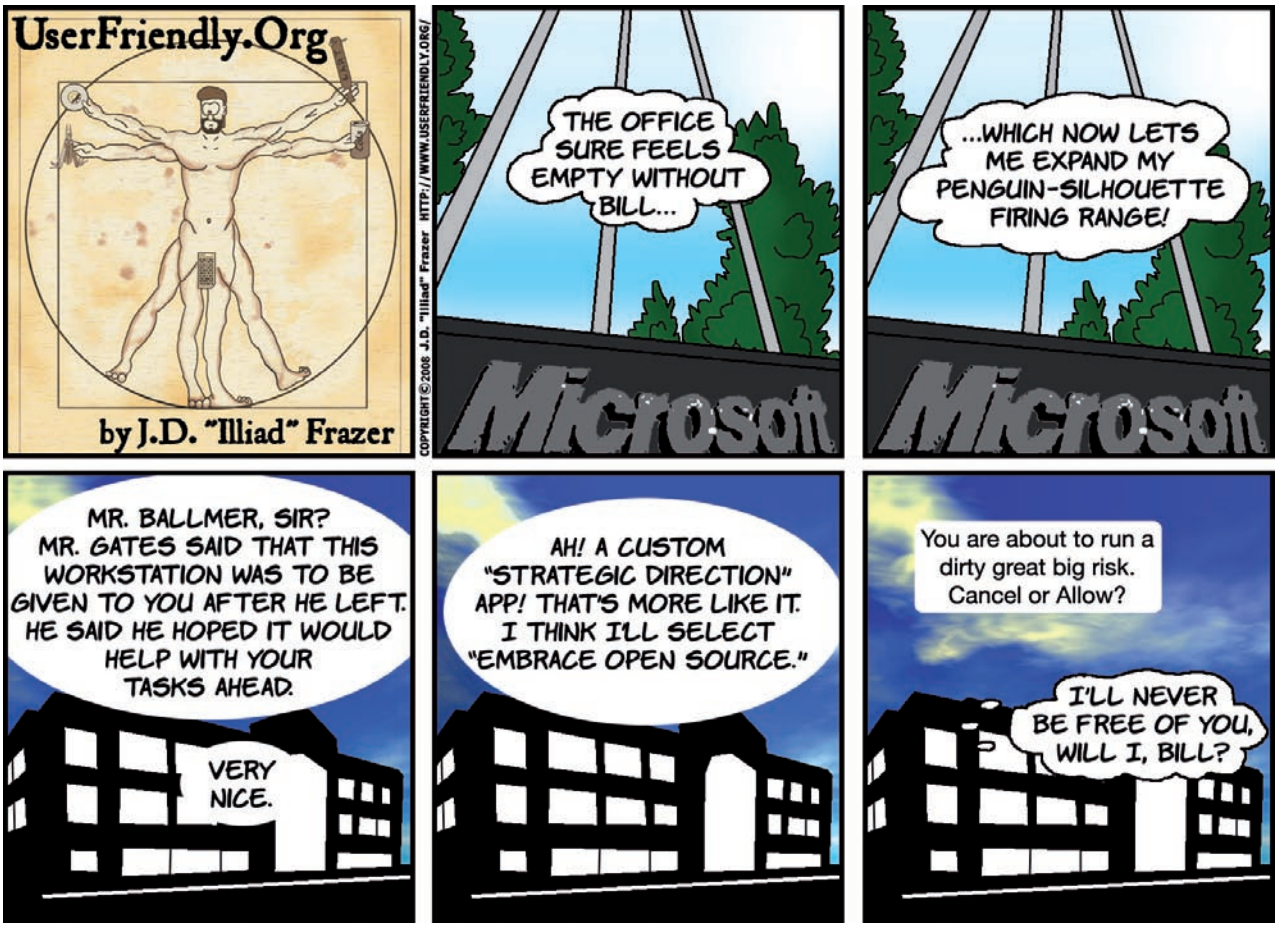
[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik: Multiprocessing und -threading", Folge 33: Linux-Magazin 05/07, S. 52

[2] Realtime-Linux-Wiki: CPU shielding using »/proc« and »/dev/cpuset«: [http://rt.wiki.kernel.org/index.php/CPU_shielding_using_/proc_and_/dev/cpuset]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

User Friendly – der monatliche Comic im Linux-Magazin



Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 42

Kern-Technik

Kennen und unterstützen alle Treiberprogrammierer das Interface fürs Powermanagement, steht einem funktionierenden Suspend-to-Ram und Suspend-to-Disk nichts mehr im Wege. Eva-Katharina Kunst, Jürgen Quade



Die Zukunft ist grün! Auch die von Linux. Mit jeder neuen Version wird sie noch grüner, schließlich tunen die Entwickler den Kernel mit immer ausgereifteren Stromspartechniken (siehe Kasten „Green Computing“). Ganz aktuell überarbeiten die Kernelhacker das Linux-Powermanagement (PM). Sie hoffen damit Treiberentwickler dafür zu gewinnen, noch mehr PM-Funktionen zu unterstützen. Das Powermanagement kümmert sich um Suspend-to-Ram (STR) und um Suspend-to-Disk (STD). An anderer Stelle nennen Applikationen das auch Schlafenlegen (Sleep) oder Einfrieren des Systems (Hibernate).

Suspend-to-Ram und Suspend-to-Disk sind nur zwei der drei möglichen Schlafzustände, die die Industrie eingeführt hat, um Strom zu sparen (siehe Abbildung 1 und Tabelle 1). Den dritten Zustand – Standby – kennt Linux auch. Aber weil er kaum Vorteile bietet, setzt Linux ihn typischerweise nicht ein: Das Betriebssystem versetzt dabei seine Komponenten nur in einen leichten Schlummer und benötigt daher etwas mehr Strom als STR. Das wiegt die nur unwesentlich kürzere Zeit, um wieder aufzuwachen, nicht auf, findet Kernelentwickler Pavel Machek und kommentiert dies mit „easy, but useless“.

Damit Suspend-to-Ram und Suspend-to-Disk funktionieren, müssen nicht nur die Hardware, sondern vor allem die Treiber die Methoden unterstützen. Daran hapert es aber vielfach. Oft deshalb, weil die Programmierer das zugehörige Interface einerseits nicht kennen und es sich darüber hinaus in der Vergangenheit gelegentlich änderte. Zum Glück erfordern die letzten Änderungen des Powermanagement-Code im Kernel nicht gleich eine Neu-Implementierung aller Treiber; vielmehr passen Entwickler ihre Treiber mit wenigen Zeilen Code an.

Altes und neues API

Um den Ruhezustand (Suspend) und den Tiefschlaf (Hibernate) zu unterstützen, implementieren Treiberprogrammierer bis einschließlich Kernel 2.6.26 ein einfaches API. Es besteht aus einer Funktion zum Einschlafen (Suspend) und einer zum Aufwachen (Resume). Es findet noch weithin Anwendung, auch wenn es die Linux-Entwickler in Version 2.6.27 aktualisiert haben.

Im alten Modell ruft Linux in jedem Treiber die zwei genannten Methoden auf. Beide erhalten neben einem Zeiger auf das schlafen zu legende oder aufzuweckende Gerät noch einen Parameter vom Typ »pm_message_t«. Er gibt über den anvisierten Zustand Auskunft, zur Auswahl stehen Suspend-to-Ram oder Suspend-to-Disk. Innerhalb der Suspend-Funktion implementieren Entwickler die notwendigen Schritte, um das entsprechende Gerät ordnungsgemäß schlafen zu legen. Typischerweise schalten sie dazu alle DMA-Transfers und I/O-Operationen ab, retten den Gerätezustand und deaktivieren dann das Gerät. Innerhalb der Resume-Funktion aktivieren sie das Gerät wieder, restaurieren den geretteten Zustand und setzen eventuell unterbrochene Zugriffe fort.

Starke Bindungen

Das Powermanagement folgt programmtechnisch dem Gerätemodell. Es bietet drei Möglichkeiten, eigene Funktionen einzubinden: Entweder bindet der Entwickler die Suspend- und Resume-Methoden an das Klassen-Geräteobjekt (»struct class_device«) oder an das Geräteobjekt (»struct device«). Das zugehörige Bussystem, zum Beispiel PCI oder USB, verwaltet es. Wer also ein neues Bussystem in den Kernel integrieren möchte – etwa für Controller Area Networks – bindet die beiden Funktionen direkt an das Busobjekt vom Typ »struct bus_type«.

An welches Objekt der Entwickler die Funktionen bindet, hängt vom Zeitpunkt ab, zu dem Linux die Funktionen aufrufen soll. Tatsächlich durchläuft der Kernel mehrere Zustände, bis er zum Stillstand kommt (siehe Abbildung 2). Sobald der Anwender das System schlafen legt, schreibt Linux zunächst alle im Page-Cache befindlichen, ungesicherten Daten auf den Hintergrundspeicher. Dazu verwendet es den Systemaufruf »sys_sync()«, den auch das Kommando

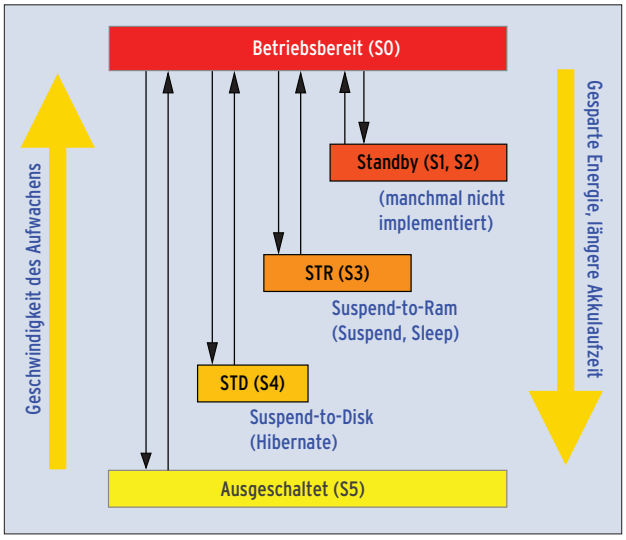


Tabelle 1: S-States

Zustand	Beschreibung
S0	Betriebsbereit (Working)
S1	Schlafend, der CPU-Kontext bleibt erhalten
S2	Schlafend, der CPU-Kontext geht verloren
S3	Suspend-to-Ram (STR), diverse Hardware ist abgeschaltet
S4	Suspend-to-Disk (STD), gesichert auf Hintergrundspeicher
S5	Ausgeschaltet

◀ **Abbildung 1: Linux stellt einen von drei Schlafzuständen zur Auswahl, um verschieden tief zu schlafen und so Strom zu sparen.**

»sync« verwendet. Registrierte Komponenten wie Threads informiert das Betriebssystem per Suspend-Notifier über die bevorstehende Zustandsänderung. Hat es seine Callback-Funktionen abgearbeitet, friert es die Rechenprozesse ein. Aus Diskussionen der Linux-Entwickler lässt sich ablesen, dass das Einfrieren der Rechenprozesse wohl in künftigen Kernelversionen unterbleibt.

Anschließend versetzt Linux die Geräte in genau umgekehrter Reihenfolge ihrer Registrierung in den Schlafzustand. Dazu ruft es zuerst die Suspend-Methoden auf, die der Treiberentwickler über die Geräteklassen-Objekte dem Kernel bekannt gegeben hat (siehe Listing 1).

Im zweiten Schritt bearbeitet Linux die Funktionen der Geräteobjekte (siehe Listing 2) und drittens die des Bussystems. Das Powermanagement schaltet nun alle Prozessoren einer Mehrkernmaschine außer dem ersten ab, sperrt Interrupts und ruft die an die Geräteobjekte gekoppelten »suspend_late()«-Methoden auf. Nachdem die Systemgeräte abgeschaltet sind, schläft Linux. Erwartungsgemäß weckt Linux das System in umgekehrter Reihenfolge wieder auf.

Listing 1 zeigt ein vollständiges Beispiel, das Powermanagement-Funktionen über das Klassenobjekt im Kernel 2.6.26 einbindet. Bevor Experimentierfreudige den Code allerdings übersetzen und den Treiber per »insmod suspendtest.ko« laden, sollten sie sicherstellen, dass ihr System den Ruhezustand unterstützt. Bei den meisten Systemen drücken sie dafür kurz den Ein-Aus-Schalter und schicken es so in den Ruhezustand STR. Danach wecken sie es wieder auf.

Müde bin ich, geh' zur Ruh

Das Powermanagement hat in der Zwischenzeit die Funktionen »my_sus-

Listing 1: »suspendtest.c« testet den Ruhezustand

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <linux/device.h>
04 #include <linux/pm.h>
05 #include <asm/uaccess.h>
06
07 static int major = 0;
08 static int suspend_count = 0;
09
10 int my_suspend(struct device *dev,
11                pm_message_t state) {
12   switch (state.event) {
13   case PM_EVENT_ON:
14     printk("on ...\n"); break;
15   case PM_EVENT_FREEZE:
16     printk("freeze ...\n"); break;
17   case PM_EVENT_SUSPEND:
18     printk("suspend ...\n"); break;
19   case PM_EVENT_HIBERNATE:
20     printk("hibernate ...\n"); break;
21   default:
22     printk("pm_event: 0x%x\n",
23            state.event); break;
24   }
25   printk("my_suspend(%p, %d)\n",
26          dev, ++suspend_count);
27   return 0;
28 }
29
30 int my_resume(struct device *dev) {
31   printk("my_resume(%p, %d)\n", dev,
32          ++suspend_count );
33   return 0;
34 }
35
36 static struct file_operations fops = {
37   .owner = THIS_MODULE,
38 };
39
40 static struct class *test_class;
41
42 static int __init my_module_init(void) {
43   printk("my_module_init\n");
44   if ((major = register_chrdev(0,
45                 "testdev", &fops))) {
46     test_class = class_create(THIS_MODULE,
46                   "testclass");
47     if (IS_ERR(test_class)) {
48       printk("no udev support\n");
49       unregister_chrdev(major, "testdev");
50       return -EIO;
51     }
52     test_class->suspend = my_suspend;
53     test_class->resume  = my_resume;
54     device_create(test_class, NULL,
55                   MKDEV(major, 0), "lulu");
56     return 0;
57   }
58
59   printk("loading failed\n");
60   return -EIO;
61 }
62
63 static void __exit my_module_exit(void) {
64   unregister_chrdev(major, "testdev");
65   if (!IS_ERR(test_class)) {
66     device_destroy(test_class,
67                    MKDEV(major, 0));
68     class_destroy(test_class);
69   }
70   printk("my_module_exit; "
71          "%d times suspended\n",
72          suspend_count);
73 }
74
75 module_init(my_module_init);
76 module_exit(my_module_exit);
77 MODULE_LICENSE("GPL");
```

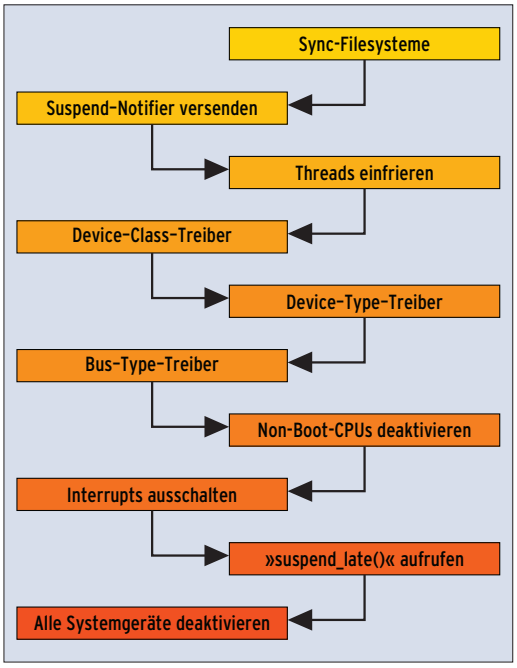


Abbildung 2: Bei Suspend oder Hibernate versucht Linux zunächst ganze Geräteklassen, danach die an den Bussen befindlichen Geräte zum Nachtschlaf zu bewegen.

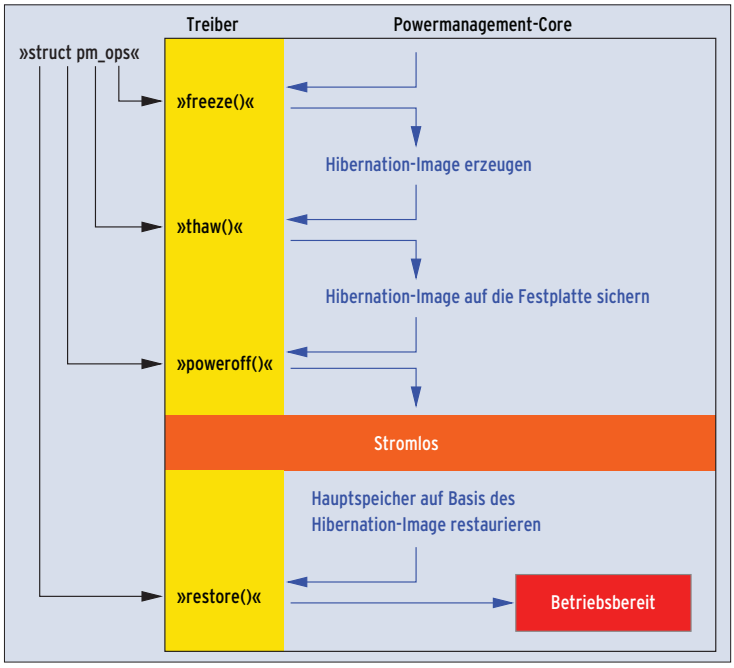


Abbildung 3: Für die Implementierung von STD erwartet Kernel 2.6.27 neuerdings vier Methoden vom Treiber statt nur zwei. Gerade beim Suspend-to-Disk haben Entwickler feinere Möglichkeiten, etwa ein Image zu erzeugen und zu reagieren, falls dies fehlschlägt.

pend()« und »my_resume()« aufgerufen. Die Funktion »my_suspend()« hat die Nachricht vom Typ »pm_message_t« in den Zeilen 12 bis 24 dekodiert und die globale Variable »suspend_count« hochgezählt. Der Sparringspartner »my_resume()« inkrementiert die Variable »suspend_count« ebenfalls (Zeile 32). Wer das Modul mittels »rmmod suspendtest« entlädt, sorgt dafür, dass es den Wert von »suspend_count« über die Kernelmeldungen ausgibt. Typischerweise findet sich der Wert also in der Datei »/var/log/messages«. Alternativ lässt sich der Ringbuffer mittels »dmesg« ausgeben.

Schlafen in Bussen

Die Funktionen »my_suspend()« und »my_resume« lassen sich nicht nur über Geräteklassen einbinden, denn alternativ erledigt das auch das jeweilige Bus-Subsystem. Die Prototypen der Funktionen unterscheiden sich dazu nur leicht. Listing 2 zeigt ein zugehöriges Codefragment für den PCI-Bus. Die Zeilen 5 bis 18 fassen die Methoden in einem »pci_drv«-Objekt zusammen, Zeile 40 registriert sie. Testen lässt sich der Code aber nur, wenn auch wirklich ein PCI-Gerät angeschlossen ist. Nur in diesem Fall ruft das PCI-Subsystem die Funktionen auf. Eine Alternative zum Testen des Powerma-

Listing 2: »pcitest.c« testet Suspend und Resume mit PCI-Bus

```
01 ...
02 static int suspend_count = 0;
03 static int major = 0;
04 ...
05 static struct pci_device_id pci_drv_tbl[] = {
06   {MY_VENDOR_ID, MY_DEVICE_ID,
07      PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
08   {0, }
09 };
10
11 static struct pci_driver pci_drv = {
12     name:     "pci_drv",
13     id_table: pci_drv_tbl,
14     probe:    pci_init_one,
15     remove:   pci_remove_one,
16     suspend:  my_suspend,
17     resume:   my_resume,
18 };
19
20 static int my_suspend(struct pci_dev *dev,
21                       pm_message_t state) {
22   printk("my_suspend(%p, %d)\n", dev,
23          ++suspend_count);
24   return 0;
25 }
26
27 static int my_resume(struct pci_dev *dev) {
28   printk("my_resume(%p, %d)\n", dev,
29          ++suspend_count);
30   return 0;
31 }
32 ...
33 static int __init pci_drv_init(void) {
34   int ret;
35
36   if ((major = register_chrdev(0,
37         "driver-name", &driver_fops)) != 0)
38   {
39     if ((ret =
40        pci_register_driver(&pci_drv)) < 0 ) {
41       printk(KERN_ERR "can't register "
42              "PCI driver (%d)\n", ret);
43       unregister_chrdev(major,"driver-name");
44       return -EIO;
45     }
46     return 0;
47   }
48   return -EIO;
49 }
50
51 static void __exit pci_drv_exit(void) {
52     pci_unregister_driver(&pci_drv);
53     unregister_chrdev(240, "driver-name");
54     printk("pci_drv_exit(%d)\n",
55            suspend_count);
56 }
57 ...
```

Green Computing

Seit Energiepreise und Verkaufszahlen von Netbooks gleichermaßen in die Höhe schießen, nimmt die Öffentlichkeit den Stromverbrauch von IT-Hardware wahr. Statt Prozessorleistung vergleichen Anwender jetzt die Prozessorleistung pro Watt. Statt Tipps zum Übertakten fragen Anwender in den einschlägigen Foren zunehmend danach, wie sie ihren Rechner untertakten können, und erhoffen davon ein probates Mittel zum Stromsparen und damit längere Akkulaufzeit im laufenden Betrieb.

Zwei Ansätze zum Energiesparen

Green Computing verspricht – neben der umweltfreundlichen Produktion und Wiederverwertung der Geräte – den Stromverbrauch von Servern, Desktops, Notebooks und sonstigen prozessorgesteuerten Geräten auf ein Minimum zu senken. Seine Realisierungsvarianten fußen alle auf zwei Grundprinzipien: Erstens stellen die Geräte nur so viel Leistung zur Verfügung, wie tatsächlich benötigt wird. Andernfalls drehen sie den Strom ab. Zweitens verschieben Systeme Aufgaben möglichst auf einen späteren Zeitpunkt, wenn auch andere Komponenten eine Leistung anfordern. Das nennen die Experten Leistungsbündelung.

Sanftes Herunterregeln

Neben dem im Beitrag vorgestellten Powermanagement bietet Linux weitere Stromsparteчniken. Es unterstützt etwa Prozessoreigenschaften wie Speedstep oder den konfigurierbaren Loadbalancer des Schedulers. Letzterer versucht alle Threads auf einer Mehrprozessormaschine auf eine möglichst kleine Anzahl von CPU-Kernen zu gruppieren. Ist das System nicht vollständig ausgelastet, lassen sich so einige Prozessorkerne in einen Schlafzustand versetzen. Der Anwender aktiviert diese Funktion durch »echo 1 > /sys/devices/system/cpu/sched_mc_power_savings«.

Das seit gut einem Jahr im Kernel aufgenommene Tickless-Patch [4] leistet auch seinen Beitrag zum Energiesparen: Linux löst nur dann Interrupts aus, wenn es etwas abzuarbeiten gibt. Für den energiebewussten Kernelhacker bringt das Tickless-Patch die Funktionen »round_jiffies()« und »round_jiffies_relative()« mit. Sie verschieben zeitgesteuerte Aktionen auf die nächste volle Sekunde und verhindern häufiges Aufwecken für nur kurze Zeiträume. Der Erfolg dieser und anderer Maßnahmen ist durchaus messbar: Ein Vergleichstest bescheinigt einem Red-Hat-Linux-Server einen um bis zu zwölf Prozent günstigeren Stromverbrauch als einem Server unter Windows 2008 [5].

nagements zeigt Rafael J. Wysocki mit einem Patch für den Kernel, der im Sys-FS ein Gerät anlegt, das den Übergang in die jeweiligen Powerstates simuliert [2].

Die gezeigten Suspend- und Resume-Funktionen reizen die Möglichkeiten, Powermanagement im Treiber zu realisieren, bei Weitem noch nicht aus. Die Methoden »suspend_late(struct device *, struct pm_message_t *)« und »resume_early(struct device *)« ruft Linux – wie die Namen andeuten – beim Schlafenlegen zu einem späteren und beim Aufwecken zu einem früheren Zeitpunkt auf. Sie unterscheiden sich von den vorgenannten Funktionen darin, dass die Interrupts bereits beziehungsweise noch gesperrt sind (siehe Abbildung 2).

Einfrieren und auftauen

Wie bereits angedeutet, ändert Kernel 2.6.27 die Schnittstellen der Energieverwaltung. Leonard Brown und Rafael Wysocki haben sie auf dem diesjährigen Linux-Symposium in Ottawa vorgestellt [3], Torvalds hat die zugehörigen Patches im Juli in seinen Quellcode übernommen. Eines der Hauptprobleme beim bisherigen Ansatz bestand darin, dass der Funktion »resume()« der Parameter vom Typ »pm_message_t« fehlt. Daher ist nicht sichtbar, ob Linux den Treiber aus dem Ruhezustand oder aus dem Tiefschlaf aufweckt.

Einige Treiber wollen jedoch feiner steuern, wie sie ihr betreutes Gerät in den Schlaf wiegen oder behutsam wieder aufwecken. Das trifft insbesondere auf

Anzeige

Advanced Configuration and Power Interface (ACPI)

Typischerweise ist auf den PC-Plattformen ACPI für das Schlafenlegen von Hardware zuständig. Der von HP, Intel, Microsoft, Phoenix und Toshiba im Jahr 1996 definierte Standard legt verschiedene Zustände fest, unter anderem die Sleep-Zustände (S-States, [6]). Die S-States wiederum korrespondieren mit den Device-Power-States (D-States). Sie definieren unabhängige Stromsparzustände, jedoch keine vollständigen Leistungszustände wie die S-States. Ein Gerät kann sich etwa in einem Sparmodus befinden, obwohl sich das Gesamtsystem im Betriebsmodus S0 befindet. Ein Rechner hingegen ist nicht betriebsbereit, wenn er sich in einem der Zustände S1 bis S5 befindet (siehe Tabelle 1).

ACPI ist im Bios in Form von Tabellen und Methoden abgelegt. Die Tabellen beschreiben die Hardware, während die Methoden Code darstellen, den der ACPI-Code des Betriebssystems zu bestimmten Zeitpunkten im Kernelkontext aufruft. Den Code geben die Hardwarehersteller vor, die Kernelentwickler haben kaum eine Möglichkeit, um zu prüfen, was er genau tut.

Viele Linux-Entwickler sind nicht besonders glücklich über ACPI. Die Spezifikation gilt als komplex und mit 600 Seiten als umfänglich. Daher sind viele Umsetzungen im Bios entweder fehlerhaft, nicht offengelegt oder schlecht dokumentiert. Linux führt den ACPI-Code mit privilegierten Rechten aus, was Experten für sicherheitskritisch halten. Der Code unterscheidet das aufrufende Betriebssystem. Aus diesem Grund meldet sich Linux neuerdings beim ACPI als Windows an. Nur so erreichen die Entwickler ein konsistentes Verhalten. Und nicht zuletzt hat Microsoft in der Vergangenheit versucht, über Patente Linux von der Verwendung von ACPI auszuschließen [7].

die Low-Level-Treiber von Bussystemen zu. Dem Wunsch kommen die Kernelentwickler nach: Für beide Übergänge in den Ruhezustand STR und den Schlafzustand STD bieten sie Callbackfunktionen an. Geht alles glatt, liefern sie »NULL«, sonst einen Fehlercode zurück.

Für den Ruhezustand stehen die vorgestellten Methoden »suspend()« und »resume()« bereit. Suspend-to-Disk erwartet jetzt die vier neuen Methoden »freeze()«, »thaw()«, »poweroff()« und »restore()« (siehe Abbildung 3): Das Subsystem ruft »freeze()« auf, bevor es das Image erzeugt und dieses später auf den Hintergrundspeicher schreibt (siehe Listing 3). Die »thaw()«-Methoden arbeitet es ab, bevor es das erzeugte Image speichert. Diese ruft das System auch auf, wenn es kein Image erzeugen konnte. Dann stellt es den ursprünglichen Betriebszustand wieder her.

Nach dem Speichern des Image arbeitet der Kernel sämtliche »poweroff()«-Callbacks der Geräte ab. Erst damit zieht er sinnbildlich den Stecker der einzelnen Geräte, die dann stromlos sind. Den in Listing 3 ebenfalls aufgeführten Callback »prepare()« ruft das Powermanagement beim Suspend oder beim Hibernate auf. Anders als bei dem später folgenden Aufruf der Methode »suspend()« darf der Programmierer davon ausgehen, dass zum Zeitpunkt des Aufrufs noch alle Geräte aktiviert und nutzbar sind. Innerhalb dieser Funktion lässt sich auf normalem Wege Speicher reservieren.

Gnadenfrist für alten Code

Das Gegenstück »complete()« ruft der PM-Core auf, nachdem er bei allen Geräten die Callbacks »resume()«, »thaw()« oder »restore()« abgearbeitet hat. Eine genaue Beschreibung all dieser Methoden enthalten die Kernelquellen in der Headerdatei »include/linux/pm.h«.

Treiberprogrammierer, die alte und neue Kernel unterstützen möchten, sind zurzeit noch fein raus: Das bisherige Interface steht bis auf Weiteres zusätzlich in den aktuellen Kerneln zur Verfügung. Bleibt abzuwarten, wann dieser Code auf einen biologisch abbaubaren grünen Code-Komposthaufen wandert. *(mg)* ■

Listing 3: Datenstrukturen zum Powermanagement

```
01 ...
02 static struct pm_ops my_pm = {
03     prepare:  my_prepare,
04     complete: my_complete,
05     suspend:  my_suspend,
06     resume:   my_resume,
07     freeze:   my_freeze,
08     thaw:     my_thaw,
09     poweroff: my_poweroff,
10     restore:  my_restore,
11 };
12 ...
13 static int __init my_module_init(void) {
14   printk("my_module_init\n");
15   if ((major = register_chrdev(0, "testdev",
16                                &fops))) {
17     test_class = class_create(THIS_MODULE,
18                               "testclass");
19     if (IS_ERR(test_class)) {
20       printk("no udev support\n");
21       unregister_chrdev(major, "testdev");
22       return -EIO;
23     }
24     test_class->pm = &my_pm;
25     device_create(test_class, NULL,
26                   MKDEV(major, 0),
27                   NULL, "lulu");
28     return 0;
29   }
30   printk("loading failed\n");
31   return -EIO;
32 }
```

Infos

[1] Timo Hönig, „ACPI-Implementierung in Linux 2.6“: Linux-Magazin 02/04, S. 32

[2] „Kernelpatches testen Ruhezustände im laufenden Betrieb“: Linux-Magazin Online [http://www.linux-magazin.de/news/kernelpatches_testen_ruhezustaende_im_laufenden_betrieb]

[3] A. Leonard Brown und Rafael J. Wysocki, „Suspend-to-RAM in Linux“: [http://ols.fedoraproject.org/OLS/Reprints-2008/brown-reprint.pdf]

[4] Eva-Katharina Kunst und Jürgen Quade, „Gerade echtzeitig“: Linux-Magazin 06/08, S. 106

[5] Tom Henderson und Rand Dvorak, „Linux captures the 'green' flag, beats Windows 2008 power-saving measures“: [http://www.networkworld.com/research/2008/060908-green-windows-linux.html]

[6] Hewlett-Packard et al., „Advanced Configuration and Power Interface Specification“, Revision 3.0b, 2006: [http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf]

[7] Julius Stiebert, „Microsoft: ACPI sollte nur unter Windows funktionieren“: [http://golem.de/0704/51686.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 43

Kern-Technik

Ein kleiner, aber wichtiger Teil des Linux-Kernels ist in Assembler geschrieben. Der Inline-Assembler integriert die Maschinenbefehle dabei harmonisch in C-Funktionen. Bleibt, den Code zu verstehen. Eva-Katharina Kunst, Jürgen Quade



Kein Zweifel, die Programmiersprache C dominiert die IT-Landschaft und erst recht den Linux-Kernel. Effizienter Code, Hardwarenähe und Freiheit für den Programmierer zählen unter C-Fans zu den Gründen für den Einsatz.

Doch der Hardwarenähe und Effizienz zum Trotz geht es prinzipbedingt auch im Jahre 2008 noch immer nicht ohne den guten alten Assembler: Die Programmiersprache C kennt – klugerweise – weder direkten Zugriff auf die Ein- und Ausgabegeräte der Hardware noch in der Sprache verankerte Elemente zur Synchronisation des Prozessors. Darüber hinaus ist ein vom Profi handoptimierter Assembler-Code – sparsam und strategisch eingesetzt – in seiner Performance durch automatisch generierten Code eines Compilers selten zu schlagen.

Die Frage ist also weniger, ob Entwickler Assembler-Code benötigen, sondern vielmehr, wie sie ihn effizient mit C-Code kombinieren, um die jeweiligen Vorteile zu nutzen. Eigenständige Assembler-Funktionen zu schreiben und sie mittels Linker mit dem C-Code zu kombinieren ist wenig spektakulär. Die Entwickler wünschen sich stattdessen Teile einer C-Funktion in Assembler auszuführen, den Assembler-Code also in den C-Code einzubetten. Genau diese Funktion bietet der GNU-Compiler GCC mit seinem von den Linux-Entwicklern reichlich genutzten Inline-Assembler.

Wie der Code-Ausschnitt aus dem Linux-Kernel in Listing 1 zeigt, hat dieser Inline-Assembler allerdings etwas von einer Geheimsprache, die nur eingeweihte Entwickler kennen. Zeit, um etwas Licht ins Dunkel zu bringen und zu erläutern, wie man Assembler im Kernel liest und schreibt.

Abstimmung erforderlich

Um Assembler-Code einzubetten, stellt der GCC das Makro »asm« bereit. Denkbare Konflikte mit einem Bezeichner dieses Namens vermeiden Kernelentwickler, indem sie stattdessen auf die gleichwertige Variante »__asm__« zurückgreifen. Das Makro nimmt nicht nur die einzelnen Assembler-Zeilen entgegen, sondern regelt zugleich dessen Interaktion mit dem C-Code. Denn will der Low-Level-Programmierer auf die dort definierten Variablen zugreifen, muss er sich mit dem Compiler abstimmen, wie sie untereinander Daten austauschen.

Das geht entweder über die Register der CPU oder über Speicherzellen. Dies gilt für beide Richtungen, vom C-Programm zum Assembler-Code und umgekehrt. Das Asm-Makro koordiniert den Austausch und hat vier Felder, die jeweils ein Doppelpunkt (»:«) voneinander trennt (siehe Abbildung 1).

Im ersten Feld findet sich der eigentliche Assembler-Code (siehe Kasten „Registermodell und Befehlssatz einer x86-CPU“ und Tabelle 1). Ihm folgen – im zweiten und dritten Feld – erst eine Zuordnungsliste von Ausgaberegistern zu C-Variablen und dann eine Abbildung von C-Variablen zu Eingaberegistern, jeweils aus Sicht des übergebenen Assembler-Code. Im letzten Feld schließlich findet sich optional eine Liste von Registern, die der Assembler-Code modifiziert.

Den eigentlichen Programmtext definiert der Entwickler in einem einzelnen String.

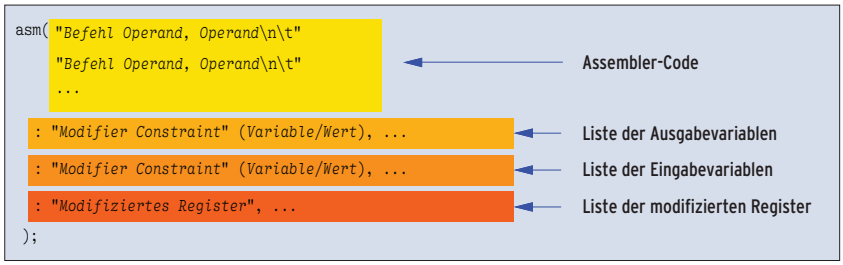


Abbildung 1: Das Asm-Makro baut eine Brücke zwischen Quelltext in C und eingebettetem Assembler-Code. Es hat vier Felder, Doppelpunkte trennen sie ab. Neben dem eigentlichen Code (gelbes Feld) gibt der Programmierer für ihn Ein- (hellorgange) und Ausgabevariablen (dunkelorange). Ändert der Code weitere Register, benennt der Entwickler diese im vierten, optionalen Feld (rot) als „clobbered Register“.

Jede einzelne Codezeile schließt dabei entweder ein Semikolon oder noch besser »\n\t« ab. Das ist darum vorteilhaft, weil der resultierende Assembler-Code so übersichtlicher wird.

Die nächsten zwei Felder der Anweisung sind komplizierter. Sie legen die Beziehung zwischen den C-Variablen und den CPU-Registern oder Speicherzellen fest. Dazu fasst der Programmierer die verwendeten C-Variablen in normale Klammern ein und trennt sie durch Kommas. Zusätzlich spezifiziert er für jede Variable, wie der C-Compiler sie handhaben soll: Entweder überlässt er es dem Compiler, ein aus dessen Sicht geeignetes Prozessorregister für den Variableninhalt auszuwählen, oder er gibt selbst eines vor.

Geben und nehmen

Diese Entscheidung legt er über die so genannten Constraints fest (siehe Tabelle 2). So überlassen die Constraints »r« für normale 32-Bit-Register und »q« für die Register der 64-Bit-Erweiterung dem Compiler die freie Registerwahl für Variablen (die generische Variante ohne x86-Bezug für »q« ist »g«). Ein Constraint »a« bis »d« zwingt dagegen den Compiler im Fall einer 32-Bit-Variablen dazu, eine Variable über die Register EAX bis EDX zugänglich zu machen (siehe Listing 2). Benötigt die zugehörige Variable weniger Platz, verwendet der Inline-Assembler automatisch die passenden Register, etwa AX oder AL und so weiter.

Es gibt noch viele weitere Constraints. Sie erlauben es dem Entwickler, dediziert auf alle Register inklusive der 64-Bit-Erweiterungen zuzugreifen. Eine ausführliche Liste der Constraints für verschiedene Prozessorfamilien steht in der Dokumentation zum GNU-C-Compiler [1].

Falls ein Programmierer eine Zuordnung zwischen C-Variable und CPU-Register über ein Constraint wie »a« fixiert hat, verwendet er im Assembler-Code auch die bekannten Registernamen (»eax«).

Tabelle 1: Wichtige Assembler-Instruktionen der x86-Architektur

Befehl	Beispiel	Bedeutung
Verschiebebefehle		
mov*x s, d*	movl %eax, %ebx	Kopiere Operand *s* auf Operand *d*
lea p	lea 0(%eax, %ecx, 4), %ecx	Lade effektive Adresse
Arithmetische Befehle		
add*x r2, r1*	addb %ebl, %eal	Addiere *r1 = r2 + r1*
sub*x r2, r1*	addq %rcx, %rax	Subtrahiere *r1 = r1 - r2*
inc*x r*	incw %ax	Inkrementiere *r*
dec*x r*	decl %eax	Dekrementiere *r*
mul*x w*	mull $5	Multipliziere *a * w*, Ergebnis in *a* und *d*
idiv*x w*	idiv $5	Dividiere *a = a / w*, Rest in *d*
Logische Befehle		
and*x r2, r1*	andl %ebx, %eax	Verknüpfe bitweise UND *r1 &= r2*
or*x r2, r1*	orl %ebx, %eax	Verknüpfe bitweise OR *r1 |= r2*
xor*x r2, r1*	xorl %ebx, %eax	Verknüpfe bitweise XOR *r1 ^= r2*
not*x r1*	notl %ecx	Verknüpfe bitweise NICHT *r1 = ~r1*
Schiebebefehle		
shl*x w, r1*	shll $4, %eax	Schiebe bitweise nach links
shr*x w, r1*	shrl $4, %eax	Schiebe bitweise nach rechts
sal*x w, r1*	sall $4, %eax	Schiebe arithmetisch nach links
rol*x w, r1*	rolb $2, %al	Rotiere nach links
rcl*x w, r1*	rclq $2, %rax	Rotiere nach rechts mit Übertrag
Sprungbefehle		
call *label*	call fctn	Rufe Unterprogramm auf
jmp *label*	jmp L1	Springe zu einem Label
cmp *b, a*	cmp $5, %eax	Vergleiche *a* mit *b*, Ergebnis im Flagregister
je *label*	je L1	Springe, wenn *a == b*
jl *label*	jl L1	Springe, wenn *a < b*
jg *label*	jg L1	Springe, wenn *a > b*
ja *label*	ja L1	Springe, wenn *a > b* (vorzeichenlos)
jbe *label*	jbe L1	Springe, wenn *a <= b* (vorzeichenlos)
Stackbefehle		
push*x w*	pushq %rdx	Schiebe *w* auf den Stack
pop*x w*	popq %rsi	Restauriere *w* vom Stack
Sonstiges		
cld	cld	Lösche das Richtungs-Flag (DF)
lock	lock	Schütze die Speicheradresse des nachfolgenden Befehls
scas*x*	repnz; scasb	Suche wiederholt in String ab Adresse EDI
stos*x*	rep; stosl	Lege wiederholt EAX in String ab EDI ab

Listing 1: Inline-Assembler aus »irq_32.c«

```
01 static int check_stack_overflow(void) {
02   long sp;
03
04   __asm__ __volatile__("andl %%esp, %0"
05                        : "=r" (sp)
06                        : "0" (THREAD_SIZE - 1));
07
08   return sp <
09           (sizeof(struct thread_info) + STACK_WARN);
10 }
```

Listing 2: »strlen()« per Inline-Assembler

```
01 #include <stdio.h>
02
03 int main(int argc, char **argv) {
04   char *from = "Hello World\n";
05   int count;
06
07   asm(
08     "movl $-1, %%ecx \n\t"// Charzähler
09     "movb $0, %%al \n\t"  // Suchzeichen '\0'
10     "cld \n\t"            // Suchrichtung einstellen
11     "repne; scasb \n\t"   // Suche durchführen
12     "not %%ecx \n\t"      // Charzähler korrigieren
13     "dec %%ecx \n\t"
14   : "=c" (count)      // Ergebnis in "count" legen
15   : "D" (from)        // Input für "scas" in RDI
16   : "al", "cc"        // Modifizierte Register
17   );
18   return printf("strlen = %d\n", count);
19 }
```

Registermodell und Befehlssatz einer x86-CPU

Bei der häufig eingesetzten x86-Architektur ist am Registermodell der Ballast einer ursprünglich auf 16 Bit ausgelegten Architektur sichtbar. Den 8086 stellte Intel immerhin bereits 1978 vor. Dessen Code läuft im Prinzip heute noch auf den modernsten Multicore-CPUs. Die Abbildung 3 illustriert, dass aus den ehemals sechs 16-Bit-Registern zwischenzeitlich 32 64-Bit-Register wurden. Jede Erweiterung der Registerbreite bringt ihr eigenes Präfix vor dem Registernamen mit, nutzt aber praktisch das gleiche Register. So lässt sich ablesen, ob ein Operand mit 8 Bit (AL), mit 16 Bit (AX), mit 32 Bit (EAX) oder schließlich mit 64 Bit (RAX) gemeint ist.

Die meisten Assembler-Befehle im GCC verwenden die so genannte AT&T-Syntax und haben den Aufbau »*Befehl mit Ergänzung Quelle, Ziel;*«. Ein Beispiel, das einen 32-Bit-Wert (long, »l«) von Register EAX nach EBX kopiert, lautet:

```
movl %eax, %ebx;
```

Die Befehlsergänzung – möglich sind hier die Buchstaben »b«, »s«, »l« und »q« – legt die Größe der Operanden fest. Die Buchstaben am Ende der Opcodes geben an, dass die Argumente jeweils eine Breite von 8, 16, 32 oder 64 Bit haben, und korrespondieren mit den jeweiligen Registernamen gleicher Länge.

Tabelle 1 listet die wichtigsten Befehle auf, sortiert nach Befehlsgruppen. Eine Einführung in den x86-Prozessor, sein Registermodell, die Adressierungsarten und seine Programmierung findet sich unter [5] und [6]. CPU-Hersteller Intel selbst beschreibt in [7] ausführlich die einzelnen Befehle.

Will er dem Compiler die Möglichkeit zur Optimierung einräumen, überlässt er diesem die Wahl des Registers. Dann verwendet er im Code statt der ausgeschriebenen Registernamen eine numerische Referenz aus den Zuordnungslisten. Dabei zählt der Assembler die Variablen in den Listen bei Null beginnend durch und referenziert sie dann im Code über die Substitutionszeichen »%*n*«. Den Platzhalter »*n*« ersetzt der Entwickler mit dem entsprechenden Zählwert.

Diffiziler Austausch

Gibt es beispielsweise drei Ausgabevariablen im C-Code, in die der Assembler-Code Daten schreibt, listet der Programmierer sie in der ersten Zuordnungsliste auf. Er greift auf die erste Variable über »%0«, auf die zweite über »%1« und auf die dritte schließlich über »%2« zu.

Die Positionsnummer darf er übrigens auch im Constraint selbst verwenden.

```
int main(int argc, char **argv)
{
    int i = 10, j = 0, k;

    asm("movl  %2  %%eax;    \n\t"
        "movl  %%eax,  %0;   \n\t"
        "movl  %%eax,  %1;   \n\t"
      : "=r" (j), "=r" (k)
      : "r" (i)
      : "eax"
    );
    return 0;
}
```

Abbildung 2: Die Zuordnung von Variablen zu Registern bereitet oft Schwierigkeiten: Der C-Code übergibt »i« an den Assembler und heißt dort »%2« (rot). Die in der Zuordnungsliste als erste und zweite genannten Variablen »j« und »k« heißen »%0« (gelb) und »%1« (organge).

Das dient beispielsweise zur Vorinitialisierung eines Registers, wie Zeile 6 in Listing 1 zeigt. Der Compiler legt den Wert von »(THREADSIZE - 1)« in der Variablen mit der Nummer »0« ab. Das ist in Zeile 5 die Variable »sp«.

Abbildung 2 enthält ein weiteres Codebeispiel, das demonstriert, wie der Assembler auf Variablen aus einem C-Programm zugegreift. Dort sind die drei Variablen »i«, »j« und »k« definiert. Gemäß der Reihenfolge ihres Auftretens in den beiden Zuordnungslisten für Ein- und Ausgaberegister referenziert sie der Inline-Assembler: Die Variable »j« taucht dort als Erste auf. Sie erhält damit innerhalb des Assembler-Code den Namen »%0«. Es folgen »k« als zweites Argument mit dem Namen »%1« und schließlich schreibt der Assembler-Programmierer »%2«, wenn er »i« ansprechen möchte. Um die Operanden (im Beispiel »%0«, »%1« und »%2«) von den Registern besser zu unterscheiden, bekommen die Register übrigens neben dem ohnehin obligatorischen »%«-Zeichen ein identisches zweites Zeichen zugestellt. Das Register EAX schreibt sich also im Inline-Assembler-Code »%%eax«.

Wer die Nummerierung der automatisch zugeordneten Register nicht mag, darf für die Variablen einen symbolischen Namen wählen. Dieser steht dann in eckigen Klammern vor dem zur Variablen gehörenden Constraint. Das sieht folgendermaßen aus:

```
asm("movl %[variable_i], %%eax;\n\t"
    "movl %%eax, %[variable_j];\n"
:[variable_j] "=r"(j)        /* output */
:[variable_i] "r"(i)          /* input */
:"%eax"          /* clobbered register */
);
```

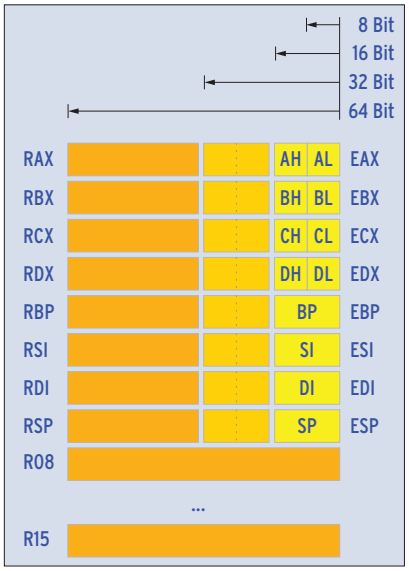
Jeder Constraint lässt sich um einen Modifier ergänzen, also um die Beschreibung seiner Eigenschaften (siehe Tabelle 3). Das im Beispiel vorangestellte »=« bedeutet etwa, dass sich das Register nur vom Assembler-Code beschreiben lässt. Häufig ist auch der Early-Clobber-Modifier »&« notwendig: So gekennzeichnete

Tabelle 2: x86-Constraints (Ausw.)

Modifier	Bedeutung
Vom Assembler gewählte Register	
r	Register für jeden Einsatzzweck
R	Register aller x86-CPUs: *a, b, c, d,* SI, DI, BP, SP
q	Daten-Register (*a, b, c* oder *d* bei 32-Bit-CPUs, bei 64 Bit jedes Integer-Register)
Q	Register mit adressierbarem High-Byte (*a, b, c, d*)
f	Fließkomma-Register des 80387-Coprozessors
y	MMX-Register
x	SSE-Register
Vom Programmierer gewählte Register	
a	RAX, EAX, AX, AL
b	RBX, EBX, BX, BL
c	RCX, ECX, CX, CL
d	RDX, EDX, DX, DL
S	RSI, ESI, SI
D	RDI, EDI, DI
A	*a* und *d* (für spezielle Befehle, etwa die Mutliplikation)
K	8-Bit-Konstante mit Vorzeichen
N	8-Bit-Konstante ohne Vorzeichen

Tabelle 3: Constraint-Modifier

Zeichen	Bedeutung
=	Nur zum Schreiben
+	Lesen und Schreiben
&	Early Clobber
%	Tausch mit Nachfolger erlaubt
#	Ignore (bis zum Komma)
*	Ignore (nächstes Zeichen)



◀ **Abbildung 3: Gab es beim 8086 nur sechs 16-Bit-Register, stehen dem Programmierer in der neusten x86-Generation 16 Datenspeicher mit je 64 Bit zur Verfügung. Das Register A etwa hat 64 Bit, RAX bezeichnet alle von ihnen. EAX Hat nur 32 Bit. 16 Bit umfasst AX und mit AH und AL erhält der Entwickler die beiden niederwertigsten Bytes.**

Register werden bereits vor dem Zugriff im Assembler-Code verändert. Dafür könnte beispielsweise der Compiler verantwortlich sein, sollte er es mit einem Initialwert belegen. Weitere Ergänzungen für die x86-Architektur enthalten die Tabelle 3 und zusätzlich für andere CPU-Architekturen die GCC-Dokumentation.

Unvermutete Änderung

Nach den beiden Zuordnungslisten folgt noch das letzte Feld für die Clobbered Register. Es listet die CPU-Register auf, die der Assembler-Code verändern wird. Die muss der Compiler kennen, um zu planen, welche Register er vor dem Starten des Inline-Code sichern muss.

Doch aufpassen! Der Inline-Assembler ist empfindlich und erlaubt nur solche Register anzugeben, die nicht bereits in der Liste für die Ausgabe- oder Eingabevariablen stehen. Ansonsten reklamiert der Assembler einen Fehler. Programmierer müssen hier auch das Condition Code Register mit dem Namen »cc« angeben, wenn es der Assembler-Code verändert, was bei mehr als einzeiligem Code praktisch immer der Fall ist. Eine Liste der enthaltenen Flags zeigt Tabelle 4.

Gerade für Neulinge ist die Zuordnung zwischen CPU-Register und C-Variable unübersichtlich. Wer aber den Compiler mit der Option »-S« startet, kann den erzeugten Assembler-Code kontrollieren. Den selbst geschriebenen Assembler-Code schließt der Aufruf

```
GCC -S -o ij.s ij.c
```

dabei in die Kommentare »#APP« und »#NO_APP« ein (siehe Zeilen 16 bis 20 in Listing 3).

Zugang zum Herzen

Zum Abschluss ein Hinweis zu der in Listing 1 verwendeten Kombination von »asm« und »volatile«: Falls der Compiler per Option optimiert, passiert es schon mal, dass er den Assembler-Code aus einer Schleife hinausschiebt oder gar ganz wegoptimiert. Das Schlüsselwort »volatile« verhindert das.

Mit diesen Informationen sind angehende Kernelentwickler in der Lage, Inline-Assembler im Kernel zu lesen und auch selbst zu schreiben. Sie sollten sich dabei nicht durch den mitunter ruppigen Compiler entmutigen lassen, denn er achtet penibel darauf, dass die Entwickler die Notation genau einhalten. Eine Reihe weiterführender und hilfreicher Hinweise enthalten [2], [3] und [4]. Dem direkten Weg zum Herzen des Prozessors steht damit nichts mehr im Wege. *(mg)* ■

Tabelle 4: Wichtige Prozessorflags (Condition Code Register)

Abkürzung	Name	Bedeutung
AF	adjust flag	Überlauf in der Bitposition 3 (BCD)
CF	carry flag	Überlauf einer vorzeichenlosen Operation
DF	direction flag	Richtungsflag für Stringbefehle
OF	overflow flag	Überlauf einer vorzeichenbehafteten Operation
PF	parity flag	Gesetzt, wenn das niederwertigste Byte gerade Anzahl Bits enthält
SF	sign flag	Entspricht dem höchstwertigen Bit eines Ergebnisses
ZF	zero flag	Ergebnis ist null

Infos

[1] Constraints und Modifier: [http://GCC.GNU.org/onlinedocs/GCC-4.3.2/GCC/Constraints.html#Constraints]
[2] Brennen Underwood, „Brennan's Guide to Inline Assembly": [http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html]
[3] S. Sandeep, „GCC-Inline-Assembly-Howto": [http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html]
[4] Bharata Rao: „Inline assembly for x86 in Linux": [http://www.ibm.com/developerworks/linux/library/l-ia.html]
[5] Fabian Schmied, „AMD64-Assembler-Handbuch": [http://www.complang.tuwien.ac.at/ubvl/amd64/amd64.ps]
[6] Basic of x86-Architecture: [http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1a.html]
[7] Intel, „Architecture Software Developer's Manual": [http://developer.intel.com/design/pentium/manuals/24319101.pdf]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 3: Assembler markiert handgeschriebenen Code

```
01     .file   "ij.c"
02     .text
03 .globl main
04     .type   main, @function
05 main:
06 .LFB2:
07     pushq   %rbp
08 .LCFI0:
09     movq    %rsp, %rbp
10 .LCFI1:
11     movl    %edi, -20(%rbp)
12     movq    %rsi, -32(%rbp)
13     movl    $10, -8(%rbp)
14     movl    $0, -4(%rbp)
15     movl    -8(%rbp), %edx
16 #APP
17     movl %edx, %eax;
18     movl %eax, %edx;
19
20 #NO_APP
21     movl    %edx, -4(%rbp)
22     movl    $0, %eax
23     leave
24     ret
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 44

Kern-Technik

Manche Applikationen müssen verhindern, dass die CPU bestimmte Codesequenzen mehrfach betritt. Dazu aktivieren sie Realtime-Mutexe im Kernel. Die sind nicht nur besonders effizient, sondern beherrschen auch die für Echtzeitanwendungen wichtige Prioritätsinversion. Eva-Katharina Kunst, Jürgen Quade



Der Schutz kritischer Abschnitte gehört zu den diffizilsten und unangenehmsten Aufgaben eines Programmierers. Er muss diese Codeteile identifizieren (siehe Kasten „Kritischer Abschnitt“) und dazu aus einer Unzahl von Schutzmöglichkeiten die effizienteste Variante auswählen [1]. Sehr verbreitet sind das bekannte Semaphor und das Mutex. Seit Kernel 2.6 hat Linus Torvalds zusätzlich das so genannte Futex in seinen Kernel aufgenommen [2]. „Ein Futex“, so verspricht die Werbebroschüre, „ist ein Fast-User-Mutex, das deshalb so schnell ist, weil es ohne den Kernel auskommt.“

Kernel kontrolliert Eintritt

Das entspricht allerdings nur der halben Wahrheit. Richtig ist, dass ein Futex schnell ist. Unwahr hingegen, dass es ohne den Kernel auskommt. Auch das Futex benötigt Kernelunterstützung: Befindet sich bereits ein Thread im kritischen Abschnitt, legt Linux nachfolgende Threads schlafen beziehungsweise weckt sie wieder auf. Die Kernelentwickler sprechen in diesem Fall von einem Slow-Path, dem langsamen Weg. Im Fast-Path, dem schnellen Weg also, ist der Kernel tatsächlich nicht beteiligt.

Da im Userspace keine Codesequenz wirklich atomar abläuft, beruhen Futexe ausschließlich auf einzelnen Maschinenbefehlen. Einer davon ist »cmpxchg«, der in einem Schritt eine Speicherzelle verändert und gleichzeitig deren Wert überprüft. Die das Futex repräsentierende Speicherstelle »F« kann dabei einen Wert von 0, 1 oder 2 haben (Tabelle 1). Die Operation »futex_lock()« lässt sich damit auf Applikationsebene wie in Abbildung 1 realisieren [3]: Betritt der Kontrollfluss den kritischen Abschnitt, setzt er Futex »F« auf 1. So zeigt er an, dass der Abschnitt ab jetzt nicht mehr frei ist (Abbildung 1, Zeile 2). Je nach dem bisherigen Wert von »F« treten jetzt drei Fälle auf.

War der bisherige Wert 0, ist der kritische Abschnitt bislang frei. Das ist der Fast-Path und bedeutet, dass die Applikation ohne anzuhalten den kritischen Abschnitt betreten darf. Das Futex benötigt dazu keine Hilfe vom Kernel.

Im zweiten Fall ist der ursprüngliche Wert von »F« 1. Ein anderer Prozess belegt also bereits den kritischen Abschnitt, doch warten keine weiteren Rechenprozesse. Daher setzt der Thread das Futex »F« nun auf den Wert 2 (Zeile 4). Das ist das Zeichen dafür, dass ein Rechenprozess wartet. Das Betriebssystem weckt diesen Schläfer wieder auf, sobald der Abschnitt frei ist.

Prozesse treten einzeln ein

Während andere Prozesse die Zeilen 2 bis 4 abarbeiten, kann der Thread, der sich bis jetzt im kritischen Abschnitt befindet, die Funktion »futex_unlock(F)« aufrufen. Damit gibt er das Lock wieder frei. Auf einer SMP-Maschine tritt so ein Fall häufig auf. Er lässt sich am Rückgabewert von 0 in Zeile 4 leicht erkennen. Der Rechenprozess wartet dann doch nicht (er durchläuft dann die While-Schleife ab Zeile 5 nicht), sondern beendet die Funktion »futex_lock()«.

Damit betritt er direkt den kritischen Abschnitt. Allerdings hat »F« jetzt den Wert 2, obwohl in Wirklichkeit doch gar kein Prozess wartet. Die Konsequenz dieser Konstellation: Signalisiert der Prozess durch Aufruf der Funktion »futex_unlock(F)«, dass er den kritischen Abschnitt verlassen hat, durchläuft er unnötigerweise den Slow-Path und setzt einen Systemaufruf zum Aufwecken eines Rechenprozesses ab.

Tabelle 1: Werte einer Futex-Speicherzelle

Wert	Bedeutung
F = 0	Der kritische Abschnitt ist frei
F = 1	Der kritische Abschnitt ist besetzt und es gibt keinen Rechenprozess, der wartet
F = 2	Der kritische Abschnitt ist besetzt, es gibt mindestens einen wartenden Rechenprozess

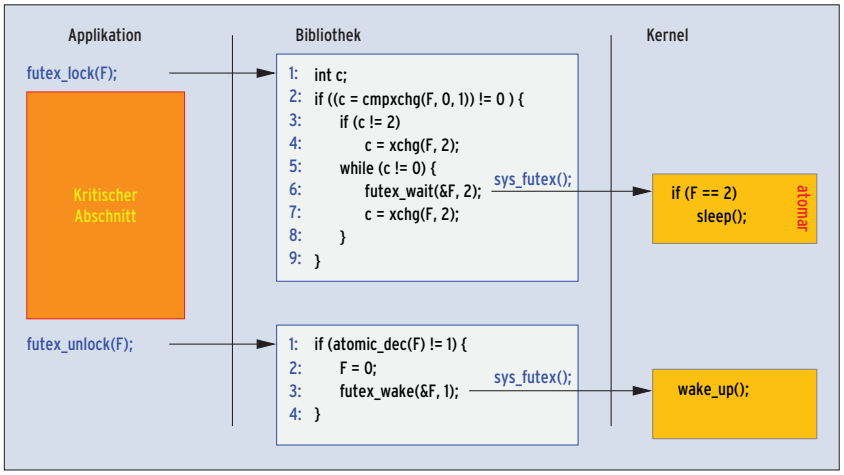


Abbildung 1: Zur Implementierung eines Locks benötigen Entwickler im Userland nur wenige Zeilen Code. Bibliotheken setzen die vom Kernel bereitgestellten Mechanismen um, hier ein Mutex.

Die dritte Alternative bekommt Bedeutung, wenn der ursprüngliche Wert 2 war. Dann war der kritische Abschnitt belegt und es warten bereits weitere Rechenprozesse darauf, ihn betreten zu dürfen. In diesem Fall legt Linux den Prozess direkt mit Hilfe des Systemcalls »sys_futex()« schlafen.

Linux singt das Schlaflied

Passiert dies, kommt der Kernel ins Spiel. Über den Systemcall »sys_futex()« erhält er neben der Adresse von »F« auch noch einen Sollwert: Nur wenn »F« diesen Sollwert besitzt, bettet der Kernel den aufrufenden Prozess wirklich zur Ruhe. Schließlich könnte ja wiederum – beispielsweise auf einem anderen CPU-Kern – der kritische Abschnitt soeben verlassen worden sein. Der Rückgabewert 0 der Funktion »xchg()« in Zeile 4 signalisiert dies, sodass der Thread die While-Schleife ab Zeile 5 nicht durchläuft.

Die Funktion zum Verlassen des kritischen Abschnitts ist glücklicherweise einfacher gestrickt. Sie liest den Wert von »F« aus und dekrementiert ihn gleichzeitig. Dabei sind nur zwei Fälle zu betrachten: War der Wert von »F« vor dem Dekrementieren 2, gibt es Rechenprozesse, die darauf warten, den kritischen Abschnitt betreten zu dürfen. Einen von ihnen weckt Linux mit Hilfe des Systemcalls »sys_futex()« wieder auf. Andernfalls beträgt der Wert vor dem Dekrementieren 1. Da niemand am Eingang des Abschnitts wartet, ist ein Aufwecken also unnötig. Der Fall, dass »F« den Wert 0 hat, tritt bei der Unlock-Operation nicht auf.

Der Kernel hat sich zum Futex »F« eine Liste angelegt, in der er die schlafenden Rechenprozesse verwaltet. Zum Aufwecken wird der erste aus der Liste entfernt und wachgerüttelt. Aber es gibt noch ein Problem: Was passiert, wenn eine Applikation in einem kritischen Abschnitt abstürzt? Anders als bei Filedeskriptoren, die der Kernel verwaltet und im Fall eines Programmabsturzes freigibt, ist das bei den Futexen zunächst nicht möglich. Aus Sicht des Kernels ist ein Futex nichts weiter als eine Speicherstelle.

Um bei Abstürzen Deadlocks zu verhindern, haben Torvalds und seine Mannen das „robuste Futex“ entwickelt [5]. Hinter ihm verbirgt sich der Systemaufruf »sys_set_robust_list()«. Er macht das Futex – genauer eine Reihe von Futexen – vor der Benutzung dem Kernel als solches bekannt. Stürzt ein Programm jetzt ab, prüft der Kernel dessen Liste der verwendeten Futexe. Findet er aktive, informiert er die wartenden Prozesse über den Absturz.

Parameter verwirren Programmierer

Entwickler verwenden also den Systemcall »sys_futex()« dazu, einen Prozess schlafen zu legen oder aufzuwecken. Seine Parameter lassen sich sehr universell einsetzen und sind damit komplex und in der Praxis fehlerträchtig. Kein Wunder also, dass die Standardbibliothek keine Funktion bereithält, die die Systemcalls »sys_futex()« und »sys_set_robust_list()« explizit unterstützt. Allerdings verwenden die unter Linux üblicherweise eingesetzten Synchronisationsfunktionen »pthread_mutex_lock()« und »pthread_mutex_unlock()« intern den Syscall »sys_futex()«. Auf diese Weise setzten Programmierer unbemerkt bereits reichlich Fast-User-Mutexe ein. Die Pthread-Funktionen unterstützen übrigens insbesondere das robuste Futex.

Für Echtzeitprogrammierer wird es allerdings erst richtig spannend, wenn sie

Kritischer Abschnitt

Entwickler bezeichnen Codesequenzen, in denen sie auf gemeinsam genutze Betriebsmittel zugreifen, als kritischen Abschnitt. Das gemeinsam genutzte Betriebsmittel ist zumeist eine Datenstruktur, etwa im Kernel die Liste aller Rechenprozesse. Damit es beim Zugriff auf das Betriebsmittel zu keinen Inkonsistenzen kommt, dürfen keine parallelen Zugriffe stattfinden. Die notwendige Sequenzialisierung realisieren die Programmierer über ein Lock (Semaphor, Mutex, Spinlock). Dazu rufen sie zu Beginn der Codesequenz eine Funktion »lock()« oder »P()« auf und zeigen das Ende des Abschnitts mittels »unlock()« oder »V()« an (Abbildung 2).

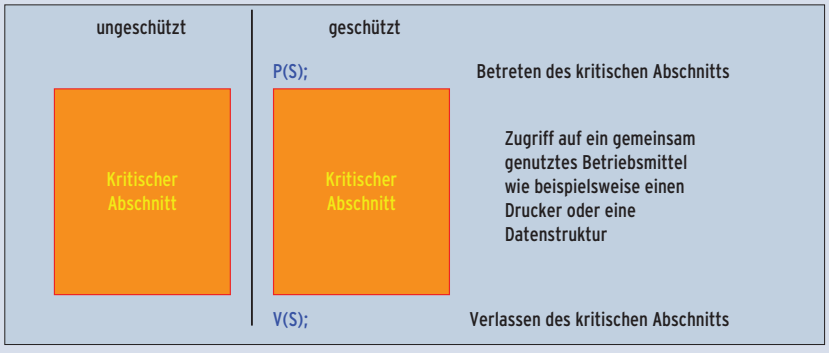
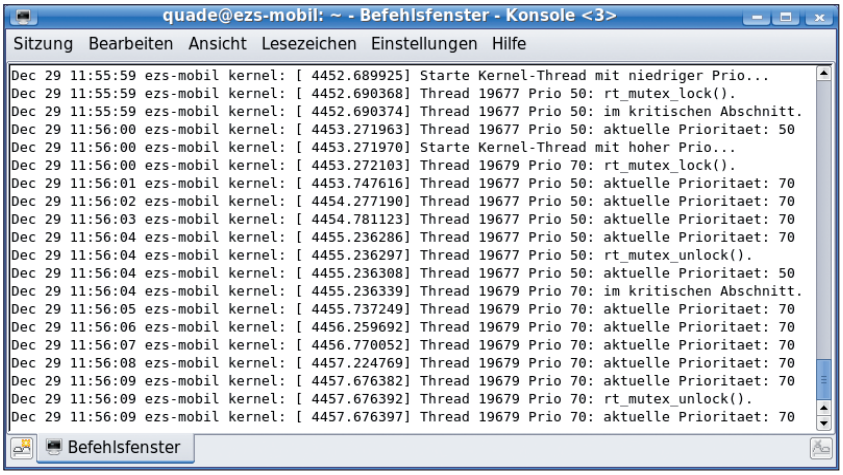


Abbildung 2: Anwender schützen kritische Abschnitte in ihren Programmen, damit diese sich nicht inkonsistent verhalten. Dazu umgürten sie betroffene Codeteile mit den Funktionsaufrufen »P()« und »V()«.



◀ **Abbildung 3: Der hochpriorisierte Thread 19679 vererbt dem niedrig priorisierten Thread 19677 seine Vorrangstellung, weil dieser ein gemeinsam genutztes Lock hält. Nun geht es für beide schneller voran.**

sich im Kernel die für ein Futex verwendete Technik ansehen: Futexe bauen auf den Realtime-Mutexen »rt_mutex« auf [4]. Darüber hinaus kommen sie auch mit den in Echtzeitanwendungen wichtigen Problemen zurecht, die sich durch unterschiedliche Prioritäten ergeben (siehe Kasten „Prioritätsinversion und Prioritätsvererbung“). Zusätzlich interessant: Realtime-Mutexe stehen sämtlichen Kernelfunktionen und Treibern zur Verfügung. Damit komplettieren sie das umfangreiche Arsenal von Schutzmechanismen, das bei Spinlocks beginnt und über Sequence-Locks zu Memory-Barrieren führt [1].

Pünktlich und priorisiert

Die Anwendung der Echtzeit-Mutexe ist einfach: Im Kernel gibt es die Datenstruktur »struct rt_mutex« und die darauf aufbauenden Funktionen »rt_mutex_lock()«, »rt_mutex_lock_interruptible()«, »rt_mutex_timed_lock()« und »rt_mutex_unlock()«. Wie in Listing 1 zu sehen, ruft der Entwickler die Funktion »rt_mutex_lock()« (im Code in der Variante »rt_mutex_lock_interrupti-

Listing 1: Zwei Threads in »prioinv.c« vererben ihre Priorität

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/sched.h>
05 #include <linux/completion.h>
06 #include <linux/kthread.h>
07
08 static wait_queue_head_t wq;
09 static DECLARE_COMPLETION(on_exit);
10 static struct rt_mutex tmut;
11 static struct task_struct *plow, *phigh;
12
13 static int process(void *data) {
15   unsigned long to, i;
16   struct sched_param schedpar;
18
19   schedpar.sched_priority = (long)data;
20   sched_setscheduler(current, SCHED_FIFO, &schedpar);
22   printk("Thread %d Prio %d: rt_mutex_lock().\n",
24          current->pid, current->rt_priority);
26
27   // Begin des kritischen Abschnitts
28   if (rt_mutex_lock_interruptible(&tmut, 0) < 0) {
30     printk("Interrupt waehrend rt_mutex_lock_interruptible\n");
31   } else {
32     printk("Thread %d, Prio %d: im kritischen Abschnitt.\n",
34            current->pid, current->rt_priority);
35     for (i = 0; i < 5; i++) {
36       to = HZ; // eine Sekunde warten
37       to = wait_event_interruptible_timeout(wq, (to == 0), to);
39       printk("Thread %d Prio %d: aktuelle Prioritaet: %d\n",
40              current->pid, current->rt_priority, 99 - current->prio);
43     }
44   }
45   printk("Thread %d Prio %d: rt_mutex_unlock().\n",
47          current->pid, current->rt_priority);
49   rt_mutex_unlock(&tmut);
50   // Ende des kritischen Abschnitts
51
52   printk("Thread %d Prio %d: aktuelle Prioritaet: %d\n",
54          current->pid, current->rt_priority, 99 - current->prio);
56   complete_and_exit(&on_exit, 0);
57 }
58
59 static int __init kthread_init(void) {
61   unsigned long to; // Timeout
62
63   init_waitqueue_head(&wq);
64   rt_mutex_init(&tmut);
65
66   printk("Thread mit niedriger Prio...\n");
67   plow = kthread_create(process, (void *)50, "prio-low");
69   if (IS_ERR(plow)) return -EFAULT;
70   wake_up_process(plow);
71
72   to = HZ; // eine Sekunde warten
73   to = wait_event_interruptible_timeout(wq, (to == 0), to);
74
75   printk("Thread mit hoher Prio ...\n");
76   phigh = kthread_create(process, (void *)70, "prio-high");
78   if (IS_ERR(phigh)) return -EFAULT;
79   wake_up_process(phigh);
81   return 0;
82 }
83
84 static void __exit kthread_exit(void) {
86   printk("kthread_exit()\n");
87   wait_for_completion(&on_exit); wait_for_completion(&on_exit);
89 }
90
91 module_init(kthread_init); module_exit(kthread_exit);
94 MODULE_LICENSE("GPL");
```

ble()«) auf, bevor sein Code den kritischen Abschnitt betritt. Gleiches gilt für »rt_mutex_unlock()«, um das Ende des kritischen Abschnitts zu signalisieren.
Die Funktionen »rt_mutex_lock()« und »rt_mutex_lock_interruptible()« unterscheiden sich nur dadurch, dass sich letztere durch ein Signal unterbrechen lässt, erstere nicht. Die Funktion »rt_mutex_timed_lock()« sorgt dafür, dass ihr Thread nicht unendlich lange auf die Freigabe des kritischen Abschnitts wartet.

Prozess rechts überholt

Wer Listing 1 kompiliert, erhält Einblick in die Prioritätsvererbung. Lädt er das Modul, erzeugt es zwei Threads. Einer erhält eine niedrige Priorität von 50, der andere eine hohe von 70. Das Beispiel räumt dem Thread mit niedriger Priorität genug Zeit ein, um den über das Realtime-Mutex geschützten kritischen Abschnitt zu betreten. Danach aktiviert es den zweiten Thread, der ebenfalls »rt_mutex_lock()« aufruft. Der bevorzugte Prozess wartet auf das RT-Mutex des niedriger eingestuften. Daher vererbt der Kernel die hohe Priorität auf den nachrangigen Prozess (Abbildung 3).
Die aktuelle Priorität des Thread speichert Linux in dem Element »prio« des Prozess-Kontrollblocks »struct task_struct«. Da für den Kernel – entgegengesetzt der Definition am Programmier-Interface – jedoch 0 eine hohe und 99 eine niedrige Echtzeitpriorität bedeutet, rechnen die Zeilen 35 und 43 die Werte wieder in Posix-Darstellung zurück.
Die Kernelentwickler haben Prioritätsvererbung durchaus kontrovers diskutiert: So schrieb Linus Torvalds noch im Dezember 2005: „Friends don't let friends use priority inheritance." Dass er die Prioritätsvererbung letztlich doch in den Kernel eingebaut hat, zeigt, dass er durchaus in der Lage ist, seine Meinung zu revidieren. *(mg)*

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Linux-Treiber entwickeln": Dpunkt-Verlag, 2. Auflage, Juni 2006
[2] Willi Nüßer, „Kampf um die Ressourcen": Linux-Magazin 10/05, S. 84; [http://www.linux-magazin.de/heft_abo/ausgaben/2005/10/kampf_um_die_ressourcen]
[3] Ulrich Drepper, „Tricky Futexes": [http://people.redhat.com/drepper/futex.pdf]
[4] E.-K. Kunst, J. Quade: „Realtime-Preemption", Linux-Magazin 07/07, S. 102
[5] Ingo Molnar, „Robust Futexes", Kerneldokumentation: [http://lxr.linux.no/linux/Documentation/robust-futexes.txt]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Prioritätsinversion und Prioritätsvererbung

Wenn ein Job mit mittlerer Priorität einen Job höherer Priorität aufhält, nennt man das „Prioritätsinversion" (Abbildung 4). Typischerweise sind daran mindestens drei Jobs beteiligt: Einer mit hoher (A), einer mit mittlerer (B) und eben einer mit niedriger Priorität (C). Außerdem teilen sich Job A und C ein Betriebsmittel. Sie haben also einen kritischen Abschnitt gemeinsam, den ein Lock (Semaphor, Mutex, Spinlock oder Futex) schützt.
Zu einer Prioritätsinversion kommt es, wenn C den kritischen Abschnitt betreten hat, in den A eintreten will. In diesem Fall blockiert A so lange, bis C die Unlock-Funktion aufruft. Wird in dieser Situation Job B lauffähig, verzögert er die Weiterverarbeitung von C, denn der hat ja niedrigere Priorität. Indirekt bremst er so aber auch A, obwohl die Verarbeitung von A aufgrund der Priorität dringlicher wäre.
Zur Entschärfung der Prioritätsinversion setzen Entwickler auf die so genannte Prioritätsvererbung. Sobald der hochpriorisierte Job A die Lock-Funktion beim Betreten des kritischen Abschnitts aufruft und das Semaphor oder das Mutex durch den niedrig priorisierten Rechenprozess C belegt ist, erbt C die hohe Priorität von A (Abbildung 5).
So ausgestattet lässt C sich nicht mehr durch den mittelpriorisierte Job B verdrängen. Gibt er den Lock frei, nimmt er wieder seine ursprüngliche Priorität an und der hochpriorisierte Job betritt – mit signifikant kürzerer Latenzzeit – den kritischen Abschnitt.

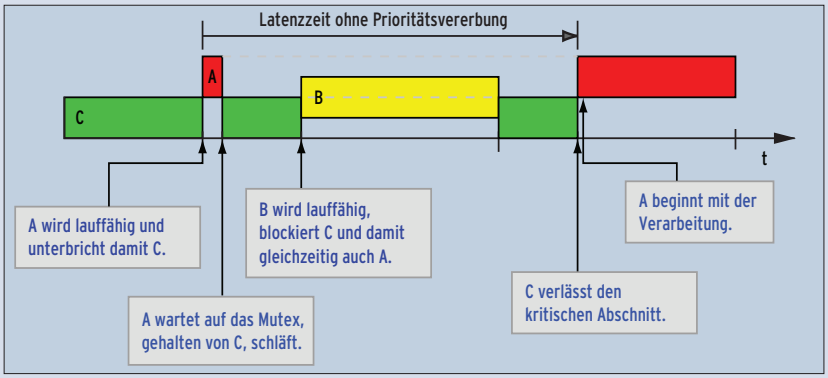


Abbildung 4: Eine Prioritätsinversion mit drei Jobs – A (rot, hohe Priorität), B (gelb, mittlere Priorität) und C (grün, niedrige Priorität) – entsteht, wenn der hochrangige Job A auf zwei nachrangige Jobs wartet.

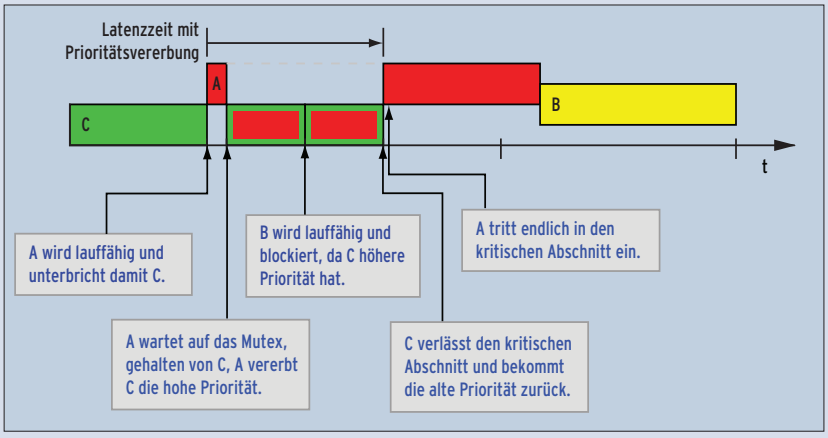


Abbildung 5: Prioritätsvererbung löst die Situation: Der langsame Job C erbt die hohe Priorität von A, solange dieser auf Einlass wartet. Job B wartet. Die Latenz von A ist insgesamt deutlich geringer.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 45

Kern-Technik

Wenn sonst nichts mehr hilft, sorgen automatisierte Neustarts für kurze Downtimes. Um Watchdogs optimal zu nutzen, brauchen Entwickler Kenntnisse über Aufbau und Anwendung der Treiber. Eva-Katharina Kunst, Jürgen Quade



Sehr naive Anwender vertrauen dem Reset-Button, denn sie wissen: Ein Reboot macht alles wieder gut. So bringen sie ihre Systeme wieder in einen definierten Zustand. Auch wenn neugierige Linux-Admins Ursachenforschung dieser brachialen Methode vorziehen, hat sie Vorzüge für alle, die auf hohe Verfügbarkeit angewiesen sind: Hängt eine Anwendung, so zieht mancher Admin die kurze Downtime des Neustarts einer stockenden Anwendung vor. Besonders bei Embedded Systems ist das Chaos eines Kaltstarts schnell behoben.

Sofern ein Reboot also Heilwirkung hat, liegt es nahe, ihn auf kritischen Systemen automatisch auszulösen. Linux identifiziert mit Hilfe einer Watchdog-Applikation den undefinierten Zustand und startet neu. Wie ein Lokführer regelmäßig den Totmannschalter betätigt, meldet sich auch die Watchdog-Applikation regelmäßig beim Kernel. Bleibt die Meldung eine Zeit lang aus, startet Linux neu.

Programmierer setzen Watchdogs in der Informatik reichlich ein. Alles, was sie dazu benötigen, ist ein Rückwärtszähler. Wenn dieser bei null ankommt, leitet der Wachhund einen Neustart ein. Im günstigsten Fall stellt die Hardware so einen Timer zur Verfügung und verbindet ihn mit dem Reset-Schalter. Solch ein Gerät zählt selbst dann weiter, wenn der gesamte Kern – aus welchen Gründen auch immer – zum Stillstand gekommen ist. Läuft alles wie gewünscht, setzt eine Anwendung den Zähler rechtzeitig wieder zurück. Linux unterstützt Watchdogs seit ewigen Kernelgenerationen. Das Verzeichnis »drivers/watchdog« in den Kernelquellen enthält Code sowohl für Hard- als auch Software-Implementationen.

Tabelle 1: IO-Controls in »linux/watchdog.h«

Name	Datentyp	Bedeutung
WDIOC_GETSUPPORT	struct watchdog_info	Bitmaske der unterstützten IO-Controls
WDIOC_GETSTATUS	int	Aktueller Status (siehe Abbildung 3)
WDIOC_GETBOOTSTATUS	int	Ursache des letzten Reboot (siehe Abbildung 3)
WDIOC_GETTEMP	int	Temperatur
WDIOC_SETOPTIONS	int	Bitmap: Enable- oder/und Disable-Card
WDIOC_KEEPALIVE	int	Setzt den Timeout auf den Defaultwert (Trigger)
WDIOC_SETTIMEOUT	int	Einstellen des Timeout (Heartbeat, in Sekunden)
WDIOC_GETTIMEOUT	int	Rückgabe des eingestellten Timeout
WDIOC_SETPRETIMEOUT	int	Setzen des Pretimeout (in Sekunden)
WDIOC_GETPRETIMEOUT	int	Auslesen des Pretimeout (in Sekunden)
WDIOC_GETTIMELEFT	int	Auslesen des Timeout

Harter Hund zählt zurück

Passende Hardware vorausgesetzt, laden die meisten Distributionen automatisch einen passenden Watchdog-Treiber. Bei halbwegs moderner Hardware, zum Beispiel Intels Centrino-Plattform, stehen die Chancen dafür gut. Falls die Hardware die Technik nicht unterstützt, springt der Software-Watchdog ein. Den dazu notwendigen Treiber lädt der Anwender mit »insmod softdog«. Den Zähler implementiert er als Timer im Kernel. Läuft er ab, ruft er eine Callback-Funktion auf, die den Rechner neu startet. Ein solcher Watchdog bringt allerdings nur Anwendungen zur Räson, die massiv andere Prozesse behindern – insbesondere die Watchdog-Applikation. Hängt der Kernel selbst, hilft dieser Ansatz nicht.

Sobald der Kernel das Character-Device »/dev/watchdog« anbietet, lassen sich seine Funktionen nutzen. Aufgepasst! Wer unbedacht mit Systemkommandos wie »cat« auf die Gerätedatei zugreift, aktiviert unversehens den Watchdog. Nach kurzer Wartezeit überrascht ein schwarzer Bildschirm alle, die ihn nicht rechtzeitig wieder deaktiviert haben.

Ansonsten ist der Einsatz von Watchdogs – egal für welchen Treiber – recht einfach. Abbildung 1 zeigt das Prinzip der Userland-Anwendung, Listing 1 setzt es um: Sobald das Programm »/dev/watchdog« sich per »open()« öffnet, ist der Wach-

hund scharf und lässt sich auch durch ein einfaches »close()« nicht ausschalten. Das Schreiben eines beliebigen Zeichens mittels »write()« auf die Gerätedatei setzt den zugehörigen Zähler zurück. Kommt dieser Aufruf nicht rechtzeitig, startet das System neu. Der Timeout beträgt in der Voreinstellung je nach Treiber zwischen 30 und 60 Sekunden.

Einige Treiber implementieren die Option »MAGICCLOSE«, um den Watchdog wieder zu deaktivieren. Ob das der Fall ist, findet der Anwender per IO-Control heraus. Er darf dann den Zeichencode für den Buchstaben »V« in die Gerätedatei schreiben. Die Wahl des Buchstabens ist rein willkürlich und hat keine weitere Bedeutung. Sobald die Applikation die Gerätedatei mittels »close()« schließt, deaktiviert Linux den Watchdog. Wer dieses Feature nicht wünscht, lädt den Treiber mit der Option »nowayout«. Dann bleibt der Watchdog bis zum nächsten Reboot unweigerlich aktiv.

IO-Controls justieren fein

Neben diesem intuitiven Interface dürfen sich Programmierer auch bestimmter IO-Controls bedienen, um Parameter wie den Timeout in Sekunden zu justieren. Ein modernes Proc- oder Sys-FS-Interface gibt es dazu leider nicht. Tabelle 1 listet die zur Verfügung stehenden IO-Controls auf. Die Kommandos »WDIOC_GETSUPPORT«, »WDIOC_SETTIMEOUT« und »WDIOC_GETTIMEOUT« kennen alle Treiber, die übrigen sind optional.

Listing 2 zeigt, wie Entwickler mit Hilfe von »WDIOC_GETSUPPORT« in Erfahrung bringen, welche Funktionalität zur Verfügung steht. Der Software-Watchdog-Treiber antwortet beispielsweise mit dem Hexcode 0x8180, aus dem sich wie in Abbildung 2 »WDIOF_SETTIMEOUT | WDIOF_KEEPALIVEPING | WDIOF_MAGICCLOSE« dekodieren lässt. Achtung: Wer vor dem Aufruf der IO-Controls den Treiber öffnet, aktiviert damit auch den Watchdog selbst! Es gilt also, danach rechtzeitig den Zähler zurückzusetzen oder aber den Watchdog wieder zu deaktivieren, wie in den Zeilen 22 und 23 ausgeführt.

Die in Hardware ausgeführten Karten überwachen einerseits das System und starten es bei Bedarf neu, denn sie sind direkt mit der Resetleitung des Computers verbunden. Bei ihnen aktiviert der Anwender einfach per IO-Control die Funktion (Tabelle 2), danach hat der Kernel damit keine Arbeit mehr – außer die regelmäßigen Trigger an die Hardware weiterzuleiten. So können diese Geräte auch einen Neustart einleiten, wenn der Kernel selbst oder einige seiner Subsysteme nicht mehr funktionieren.

Abbildung 2 zeigt außerdem, dass manche Watchdog-Karten mehr als nur einen Countdown bieten. Einige überwachen die Betriebsspannung (»WDIOF_POWERUNDER«, »WDIOF_POWEROVER«), die Temperatur (»WDIOF_OVERHEAT«) oder den Lüfter (»WDIOF_FANFAULT«). Manche Geräte verfügen sogar über separate Event-Eingänge. Liegen auf diesen Ein-

Tabelle 2: Watchdog-Settings

Name	Bedeutung
WDIOS_DISABLECARD	Deaktivieren des Watchdog-Timers
WDIOS_ENABLECARD	Aktiveren des Watchdog-Timers
WDIOS_TEMPPANIC	Kernel-Panic bei Temperatur-Problemen

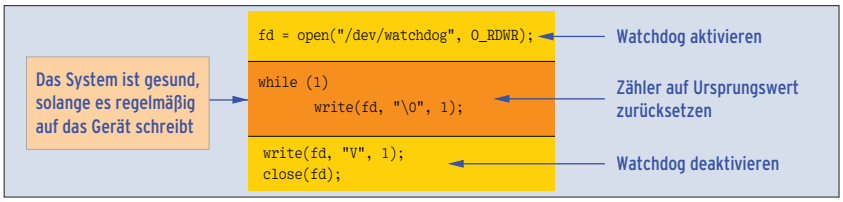


Abbildung 1: Solange eine Anwendung noch in der Lage ist, regelmäßig ein vereinbartes Zeichen in die Watchdog-Gerätedatei zu schreiben, läuft Linux weiter. Bleibt das Signal aus, rebootet der Treiber.

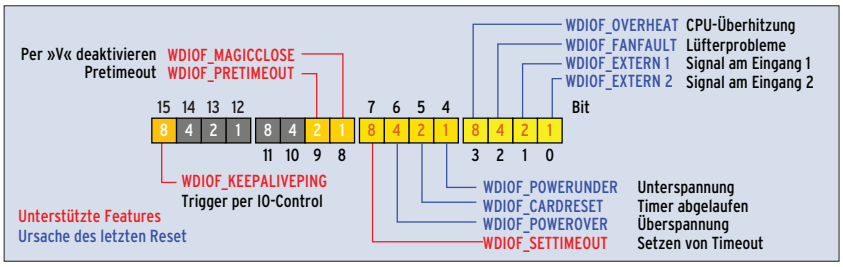


Abbildung 2: Die Bits der Struktur »watchdog_info«, die der IO-Control »WDIOC_GETSUPPORT« liefert, kodieren die Ursache für einen Neustart und die durch den Watchdog angebotenen Features.

Listing 1: Applikation zum Triggern des Watchdog

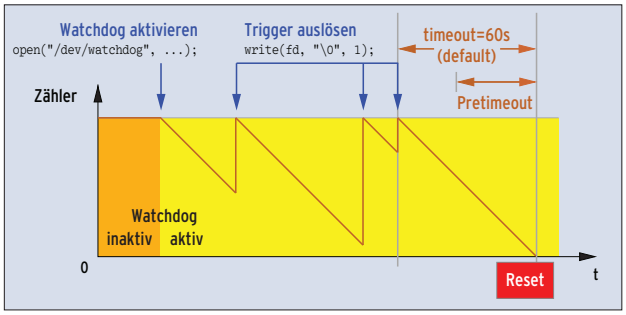
```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <fcntl.h>
05
06 int main(void) {
07     int fd = open("/dev/watchdog", O_WRONLY);
08     if (fd < 0) {
09         perror("watchdog");
10         return -1;
11     }
12     while (write(fd, "\0", 1) != 1)
13         sleep(10);
14     close(fd); /* Should never reach here */
15     return -1;
16 }
```

Listing 2: Unterstützte Treiberfunktionen

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <unistd.h>
04 #include <fcntl.h>
05 #include <sys/ioctl.h>
06 #include <linux/watchdog.h>
07
08 int main(void) {
09     struct watchdog_info wdinfo;
10     int fd = open("/dev/watchdog", O_WRONLY);
11
12     if (fd < 0) {
13         perror("wdinfo");
14         return -1;
15     }
16     if (ioctl(fd, WDIOC_GETSUPPORT, &wdinfo) < 0)
17         return -1;
18     printf("options=0x%x version=0x%x "
19         "identity=%s\n", wdinfo.options,
20         wdinfo.firmware_version,
21         wdinfo.identity);
22     write(fd, "V", 1); // Watchdog deaktivieren
23     return close(fd);
24 }
```

Listing 3: »watchdog_template.c«

```
001 #include <linux/module.h>
002 #include <linux/miscdevice.h>
003 #include <linux/watchdog.h>
004 #include <linux/fs.h>
005 #include <linux/reboot.h>
006 #include <asm/uaccess.h>
007
008 static int timeout = 60; /* in seconds */
009 static int nowayout = 0;
010 static char expect_close;
011 static unsigned long open_once=0;
012
013 /* Module parameter registration */
014 module_param(timeout, int, 0);
015 module_param(nowayout, int, 0);
016
017 /* Watchdog functions */
018 static void
019 watchdog_fire(unsigned long data) {
020     printk("reboot...\n");
021     emergency_restart();
022 }
023
024 static int reset_watchdogtimer(void) {
025     printk("reset_watchdogtimer()\n");
026     return 0;
027 }
028
029 static int
030 start_watchdogtimer(void) {
031     printk("start_watchdogtimer()\n");
032     return 0;
033 }
034
035 static int
036 stop_watchdogtimer(void) {
037     printk("stop_watchdogtimer()\n");
038     return 0;
039 }
040
041 /* File operations */
042 static int
043 driver_open(struct inode *inode,
044             struct file *file) {
045     if (test_and_set_bit(0, &open_once))
046         return -EBUSY;
047     start_watchdogtimer();
048     return nonseekable_open(inode, file);
049 }
050
051 static int
052 driver_release(struct inode *inode,
053                struct file *file) {
054     if (expect_close == 42) {
055         stop_watchdogtimer();
056     }
057     clear_bit(0, &open_once);
058     expect_close = 0;
059     return 0;
060 }
061 static ssize_t
062 driver_write(struct file *file,
063              const char __user *data,
064              size_t len, loff_t *ppos) {
065     if (len) {
066         if (!nowayout) {
067             size_t i;
068             expect_close = 0;
069
070             for (i = 0; i < len; i++) {
071                 char c;
072                 if (get_user(c, data + i))
073                     return -EFAULT;
074                 if (c == 'V')
075                     expect_close = 42;
076             }
077         }
078         reset_watchdogtimer();
089     }
080     return len;
081 }
082
083 static long
084 driver_ioctl(struct file *file,
085              unsigned int cmd,
086              unsigned long arg) {
087     void __user *argp = (void __user *)arg;
088     int __user *p = argp;
089     static const struct
090       watchdog_info ident = {
091       .options = WDIOF_SETTIMEOUT |
092                  WDIOF_KEEPALIVEPING |
093                  WDIOF_MAGICCLOSE,
094       .firmware_version = 0,
095       .identity ="Watchdog-Template",
096     };
097     switch (cmd) {
098     case WDIOC_GETSUPPORT:
099         return copy_to_user(argp, &ident,
100                sizeof(ident)) ? -EFAULT : 0;
101     case WDIOC_GETSTATUS:
102     case WDIOC_GETBOOTSTATUS:
103         return put_user(0, p);
104     case WDIOC_KEEPALIVE:
105         reset_watchdogtimer();
106         return 0;
107     case WDIOC_SETTIMEOUT:
108         if (get_user(timeout, p))
109             return -EFAULT;
110         reset_watchdogtimer();
111         /* Fall */
112     case WDIOC_GETTIMEOUT:
113         return put_user(timeout, p);
114     default:
115         return -ENOTTY;
116     }
117 }
118
119 static int
120 reboot_callback(struct notifier_block *this,
121                 unsigned long code,
122                 void *unused) {
123     if (code == SYS_DOWN ||
124         code == SYS_HALT)
125         stop_watchdogtimer();
126     return NOTIFY_DONE;
127 }
128
129 static const struct file_operations
130   driver_fops = {
131     .owner          = THIS_MODULE,
132     .write          = driver_write,
133     .unlocked_ioctl = driver_ioctl,
134     .open           = driver_open,
135     .release        = driver_release,
136 };
137
138 static struct miscdevice
139   watchdog_miscdev = {
140     .minor      = WATCHDOG_MINOR,
141     .name       = "watchdog",
142     .fops       = &driver_fops,
143 };
144
145 static struct notifier_block
146   reboot_notifier = {
147     .notifier_call  = reboot_callback,
148 };
149
150 static int __init watchdog_init(void) {
151     int ret;
152
153     ret = misc_register(&watchdog_miscdev);
154     if (ret) {
155         printk("misc_register() "
156                "failed with %d\n", ret);
157         return ret;
158     }
159     // init_watchdogtimer(watchdog_fire); ...
160     ret = register_reboot_notifier(
161                           &reboot_notifier);
162     if (ret) {
163         misc_deregister(&watchdog_miscdev);
164         printk("register_reboot_notifier()"
165                " failed\n");
166         return ret;
167     }
168     return 0;
169 }
170
171 static void __exit
172 watchdog_exit(void) {
173     unregister_reboot_notifier(
174                       &reboot_notifier);
175     misc_deregister(&watchdog_miscdev);
176 }
177
178 module_init(watchdog_init);
179 module_exit(watchdog_exit);
180 MODULE_LICENSE("GPL");
```



◀ Abbildung 3: Ist der Watchdog aktiviert, tickt die Uhr. Setzt der Anwender den Timer nicht rechtzeitig mit einem Trigger zurück, startet Linux neu. Vorher darf es noch letzte Maßnahmen treffen (Pretimeout).

gängen definierte elektrische Signale an, führt das ebenfalls zu einem Reset. Solche leistungsfähige Hardware ist typischerweise in der Lage, die Ursache für einen erfolgten Neustart über den Reboot hinaus abzuspeichern. Per »WDIOC_GETBOOTSTATUS« erfährt der Programmierer den Grund, sobald das System wieder läuft.

Ein Blick in die Quellen des Treibers schadet aber nicht, um die Verlässlichkeit dieser Informationen abzuschätzen: Ähnlich wie bei »WDIOC_GETSTATUS« implementieren zwar fast alle Treiber den IO-Control, geben aber nur 0 an die Applikation zurück. Übrigens: Wenn der Treiber mit »WDIOC_CARDRESET« antwortet, war der abgelaufene Watchdog-Timer selbst die Ursache des letzten Reboot. In der Theorie bietet der Linux-Watchdog-Treiber weitere Funktionalität [1]: Sofern die Hardware dies unterstützt, darf der Anwender einen Pretimeout einstellen. Gerät das System in einen ungesunden Zustand, löst es vor dem Reset einen Interrupt aus – zumeist einen nicht maskierbaren. Das tut er genau in dem zeitlichen Abstand des Pretimeout (Abbildung 3).

Die Zeit bis zum tatsächlichen Neustart sollte das Betriebssystem nutzen, um wichtige Daten zu sichern und Applikationen gezielt herunterzufahren. Auch wenn das API dies vorsieht, implementiert gegenwärtig kein einziger Treiber im Standardkernel diese Funktion.

Aus diesem Grund hinterlässt das Subsystem ein zweigeteiltes Bild: Einerseits hat es eine sinnvolle, aber nur in Spezialfällen nutzbare Funktionalität. Andererseits fehlen eindeutig implementierte Rückgabewerte, gute Dokumentation und ein Proc-Interface. Immerhin ist das Datei-Interface so intuitiv und universell, dass sich jeder Programmierer schnell eine für ihn angepasste Watchdog-Applikation schreiben kann. *(mg)* ■

Infos

[1] Christer Weingel, „Linux Watchdog Driver API“: Linux-Kernel-Dokumentation, [http://lxr.linux.no/linux/Documentation/watchdog/watchdog-api.txt]

[2] Oliver Frommel, Jens-Christoph Brendel, „Dynamisches Gerätemanagement mit Udev“: Linux-Magazin 09/06, S. 32

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Eigene Treiber schreiben

Ein Watchdog-Treiber im Kernel umfasst nur wenige Hundert Zeilen Code, ein Grundgerüst gibt Listing 3 vor. Für eine einfache Version sind nur die Funktionen »start_watchdogtimer()«, »stop_watchdogtimer()« und »reset_watchdogtimer()« zu implementieren (ab Zeile 24). Für einen Software-Watchdog implementiert der Kernelentwickler noch die Reboot-Funktion »watchdog_fire()«. Aufwändigere Treiber passen auch die File Operations an.

Das Modul meldet der Programmierer beim Laden in »watchdog_init()« bei Misc-Subsystem an (ab Zeile 150). So benötigt es keine eigene Major-Nummer. Der Entwickler übergibt dem Subsystem ein »struct miscdevice« und initialisiert es mit dem Namen »watchdog«. Es erzeugt die Gerätedatei im Verzeichnis »/dev« mit Hilfe von Udev [2] ohne weiteres Zutun.

Wettrennen um den Shutdown

Um eine Race Condition beim normalen Reboot und aktiviertem Watchdog zu verhindern, deaktiviert sich der Treiber in diesem Fall automatisch, damit er nicht einen von ihm selbst ausgelösten Shutdown abbricht. Ohne rechtzeitigen Trigger, den er in »driver_write()« bearbeitet (ab Zeile 61), löst er aber einen Reset aus (Zeile 21). Dazu registriert er einen Reboot Notifier mit Hilfe der Kernelfunktion »register_reboot_notifier()« in Zeile 160. Die zugehörige Callback-Funktion deaktiviert den Watchdog ab Zeile 119. Um das Modul zu entladen, ruft der Kernel »watchdog_exit()« auf und bestellt alle Registrationen wieder ab (Zeile 171).

Bitte nur einzeln eintreten

Öffnet die Watchdog-Applikation das zugehörige Device, ruft der Kernel »driver_open()« auf (ab Zeile 43). Die Funktion stellt mit Hilfe des Flag »open_once« sicher, dass nur eine Applikation den Watchdog bedient. Außerdem initialisiert und startet »driver_open()« den Rückwärtszähler. Da ein sauber programmierter Watchdog-Treiber einer Applikation nicht die Verwendung der Funktion »lseek()« erlaubt, konfiguriert der Programmierer durch »nonseekable_open()« die notwendigen Datei-Flags in Zeile 48.

Unterstützt der Treiber »WDIOC_MAGICCLOSE«, dann überprüft »driver_release()«, ob die Watchdog-Applikation vorher das magische Zeichen »V« geschrieben hat. Dies ist der Fall, wenn die Modul-globale Variable »expect_close« den Wert 42 hat. Dann stoppt der Rückwärtszähler. Ansonsten läuft er weiter und setzt den Kernel zurück, falls nicht zeitnah eine andere Watchdog-Applikation das Triggern übernimmt. Außerdem macht der Treiber in Zeile 57 den Weg frei für die nächste Applikation (»open_once«).

Der Treiber sollte minimal die IO-Controls »WDIOC_GETSUPPORT«, »WDIOC_SETTIMEOUT« und »WDIOC_GETTIMEOUT« unterstützen. Das implementiert der Modulhacker durch die Funktion »driver_ioctl()« ab Zeile 83. Die meisten Programmierer implementieren auch »WDIOC_GETSTATUS«, wobei das Modul bei Aufruf grundsätzlich 0 zurückgibt. Ein Hardware-Watchdog löst typischerweise einen Reset aus. Ist die Watchdog-Funktionalität in Software implementiert, leitet »watchdog_fire()« mit Hilfe der globalen Funktion »emergency_restart()« den Neustart ein.

Vier globale Variablen speichern den Status

Der Treiber signalisiert durch »expect_close«, dass er den Watchdog deaktivieren will. Mit »open_once« verhindert er, dass zwei Applikationen parallel den Watchdog-Treiber triggern. Die maximale Zeit bis zum nächsten Trigger legt »timeout« fest, »nowayout« verfügt, dass niemand einen aktivierten Watchdog wieder deaktivieren darf. Das Makro »modul_param« markiert die letzten beiden Optionen als Übergabeparameter an das Modul.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 46

Kern-Technik

Das Notifier-Subsystem ist die Nachrichtenzentrale des Linux-Kernels. Nur wenig Code ist nötig, um Entwickler mit den neusten Informationen zu versorgen. Eva-Katharina Kunst, Jürgen Quade



Linus Torvalds steht fest zu seinem monolithischen Betriebssystemkern. Auch wenn es sich per definitionem um einen großen Code-Klumpen handelt, ist das System in Wahrheit modular und aus vielen Quellcode-Häppchen aufgebaut. Um diese Modularität zu verwirklichen, durchwirken die Entwickler den Kern mit vielen Schnittstellen, etwa für die Integration von neuem Code, den Datenaustausch mit dem Userspace, das Ankoppeln von Hardware und Vereinbarungen darüber, wie sie Zeit verwalten. Eine der unberechtigterweise bisher wenig beachteten Schnittstellen, das Notifier-Subsystem, regelt das zentrale Informations- und Ereignismanagement.

Ähnlich einem RSS-Feed versorgt der Kernel seine Abonnenten, also Subsysteme des Kernels oder Gerätetreiber, mit Neuigkeiten, etwa über einen bevorstehenden Shutdown, das Ändern des Prozessortakts oder darüber, dass er ein Kommunikations-Interface hoch- oder runterfährt. Der Empfänger sichert daraufhin Daten, passt die Berechnung von Zeitdifferenzen an oder sucht alternative Wege zur Datenübertragung.

Bitte um Rückruf

Das Notifier-Subsystem stellt den generischen Mechanismus zur Verfügung, über den Provider Nachrichten, Informationen respektive Ereignisse publizieren und Konsumenten diese Informationen ereignisorientiert empfangen. Sobald der Provider eine Nachricht über das Notifier-Subsystem publiziert, ruft dieses beim Empfänger eine Callback-Funktion auf.

Technisch betrachtet haben die Entwickler das Notifier-Subsystem einfach realisiert (siehe Abbildung 1): Der Kernel verwaltet für jede Informationsquelle eine Liste, eine Notifier-Chain. In sie hängt er die Callback-Funktionen über definierte Routinen ein und aus.

Stellt der Provider einen zu publizierenden neuen Systemzustand fest, beginnt das Notifier-Subsystem damit, die Callback-Funktionen der Reihe nach aufzurufen. Dabei hat der Provider die Möglichkeit, dem Callback einen Nachrichtentyp (Kommando, Ereignis, Datentyp »int«) und eine Adresse (Datentyp »void *«) zu übergeben. Als Ergebnis erhält er ein Bitfeld, mit dem er sich über den erfolgreichen Ablauf oder eventuelle Fehler informiert. Der Empfänger darf auch per Returnwert darum bitten, noch in der Chain befindliche und bisher nicht aufgerufene Callbacks zu canceln.

Quartett der Ketten

Die Situation verkompliziert, dass Kernelcode in verschiedenen Kontexten abläuft. Code auf Kernelebene darf dynamisch Speicher anfordern und sich beispielsweise schlafen legen. Auf der Ebene von Soft-IRQs und in Interrupt-Handlern ist das nicht erlaubt (siehe Abbildung 2). Die Konsequenz: Code, der dort abläuft, darf kritische Abschnitte keinesfalls über

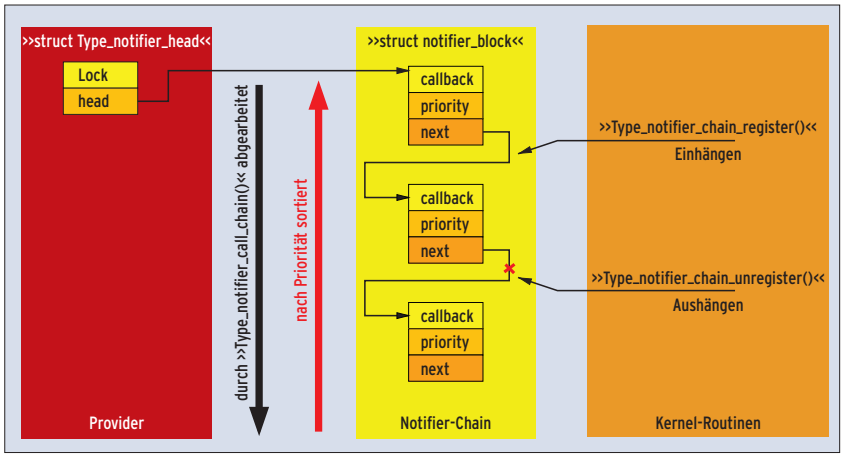


Abbildung 1: Im Zentrum des Subsystems steht die Notifier-Chain, in der der Kernel Callback-Funktionen nach Priorität sortiert ablegt. Jeder Rückruf hat einen der vier Typen »raw«, »atomic«, »blocking« oder »srcu«.

Tabelle 1: Funktionen des Notifier-Subsystems

Funktionsname	Beschreibung
int *Type*_notifier_chain_register(struct *Type*_notifier_head *nh, struct notifier_block *nb);	Einhängen einer Callback-Funktion eines bestimmten Typs. Kann für »raw«, »atomic«, »blocking« oder »srcu« stehen.
int blocking_notifier_cond_register(struct blocking_notifier_head *nh, struct notifier_block *nb);	Einhängen einer Callback-Funktion vom Blocking-Typ, falls nicht bereits geschehen.
int *Type*_notifier_chain_unregister(struct atomic_notifier_head *nh, struct notifier_block *nb);	Aushängen einer Callback-Funktion eines bestimmten Typs. Kann für »raw«, »atomic«, »blocking« oder »srcu« stehen.
int *Type*_notifier_call_chain(struct *Type*_notifier_head *nh, unsigned long val, void *v);	Aufruf der Callbacks. Als Typ sind »raw«, »atomic«, »blocking« oder »srcu« zulässig. »v« enthält eine Payload.
int __*Type*_notifier_call_chain(struct *Type*_notifier_head *nh, unsigned long val, void *v, int nr_to_call, int *nr_calls);	Aufruf mit maximal »nr_to_call« Funktionsaufrufen. Die Anzahl der tatsächlichen Callbacks befindet sich nach dem Aufruf in »nr_calls«.

Semaphore schützen, da sie wartende Instanzen schlafen legen. Folglich könnten unterschiedliche Kontexte die Funktion zum Ein- und auch zum Aushängen einer Callback-Funktion aufrufen.

Linux stellt dazu passend gleich vier Notifier-Chain-Typen zur Verfügung und überlässt es dem Programmierer, den für seine Aufgabe optimalen Typ auszuwählen. Neben der Raw-Variante, die einfach nur den Mechanismus zum Einhängen, zum Aushängen und zum Aufruf der Callback-Funktion zur Verfügung stellt, gibt es noch die Atomic-, die Blocking- und die SRCU-Notifier-Chain. Letzteres Synchronisationselement steht für den etwas sperrigen Namen Sleepable Read Copy Update.

Die letzten drei Verfahren bringen neben der Grundfunktionalität noch Schutzmechanismen für die kritischen Abschnitte mit. Kritisch ist dabei das Ein- und Aushängen einer neuen Callback-Funktion in die Notifier-Chain und schließlich das Abarbeiten der Rückrufe selbst. Technisch unterscheiden sich die Chain-Typen also dadurch, wie sie kritische Abschnitte schützen. Die Atomic-Variante verwendet als Schutzelement ein Spinlock, das parallele Abläufe mit Hilfe von Interruptsperren und aktiven Warteschleifen, dem so genannten Spinning, verhindert. Diese Variante hat nur wenig Overhead.

Jede Kernelfunktion darf die Funktionen »atomic_notifier_chain_register()« zum Ein- und »atomic_notifier_chain_unregister()« zum Aushängen aus jedem Kontext heraus aufrufen. Insbesondere ist das auch aus einer Interrupt-Service-Routine (ISR) oder einem Tasklet heraus erlaubt. Tabelle 1 beschreibt die erlaubten Aufrufe des Notifier-Subsystems.

Code muss wach bleiben

Der Kontext der Atomic-Variante bringt zwei Konsequenzen mit sich: Erstens dürfen Entwickler im Callback-Code keine Funktionen verwenden, die dazu führen, dass sich der Thread schlafen legt, und zweitens muss seine Laufzeit möglichst kurz sein.

Die Blocking-Variante verwendet zum Schutz der kritischen Abschnitte Semaphore. Die setzen ihrerseits einen Prozesskontext voraus, in dem der Scheduler den aktuellen Prozess schlafen legen darf. Hier implementieren Entwickler das Ein- und das Aushängen mit »blocking_notifier_chain_register()« und »blocking_notifier_chain_unregister()« in einem Kernelthread oder in einer Treiberfunktion, zum Beispiel in »driver_open()«.

Ein Aufruf aus einem Interruptkontext, einer ISR typischerweise, ist allerdings tabu. Der Provider initiiert die Abarbeitung der Callbacks im Prozesskontext, Programmierer berücksichtigen also, dass der Thread mitunter passiv blockiert, weil er schläft.

Nachrichtenaustausch

Die letzte Variante, SRCU Notifier Chains, steht alternativ zu den Blocking Notifier Chains. Anstelle eines Semaphors benutzen Torvalds & Co. ein SRCU-Synchronisationselement zum Schutz der kritischen Abschnitte. Sie weisen im Vergleich zum Semaphor einen geringen Laufzeit-Overhead auf. Ihr Nachteil: Das Aushängen einer Callback-Funktion ist zeitlich betrachtet teuer und die Initialisierung leicht schwieriger. Soll der Code also im Prozesskontext arbeiten und wird eine einmal aktivierte Callback-Funktion nur selten wieder ausgehängt, bietet sich diese Variante an. Achtung: Rekursion verboten! Mit Ausnahme der Raw-No-

Listing 1: Notifier senden und empfangen (»procon.c«)

```
01 #include <linux/module.h>
02 #include <linux/notifier.h>
03
04 #define USER_DEFINED_NOTIFIER_COMMAND 0x0001
05
06 /* PROVIDER */
07 static
08 BLOCKING_NOTIFIER_HEAD(user_defined_chain);
09
10 /* CONSUMER */
11 int consumer_callback(struct notifier_block
12         *self, unsigned long val, void *data)
13 {
14     printk("consumer: val: %ld, data: %p\n",
15             val, data);
16     return NOTIFY_DONE;
17 }
18 static struct
19     notifier_block user_defined_notifier = {
20         .notifier_call = consumer_callback,
21 };
22
23 static int __init notifier_init(void)
24 {
25     blocking_notifier_chain_register(
26         &user_defined_chain,
27         &user_defined_notifier);
28     /* PROVIDER */
29     blocking_notifier_call_chain(
30         &user_defined_chain,
31         USER_DEFINED_NOTIFIER_COMMAND,
32         NULL);
33     return 0;
34 }
35
36 static void __exit notifier_exit(void)
37 {
38     blocking_notifier_chain_unregister(
39         &user_defined_chain,
40         &user_defined_notifier);
41 }
42
43 module_init(notifier_init);
44 module_exit(notifier_exit);
45 MODULE_LICENSE("GPL");
```

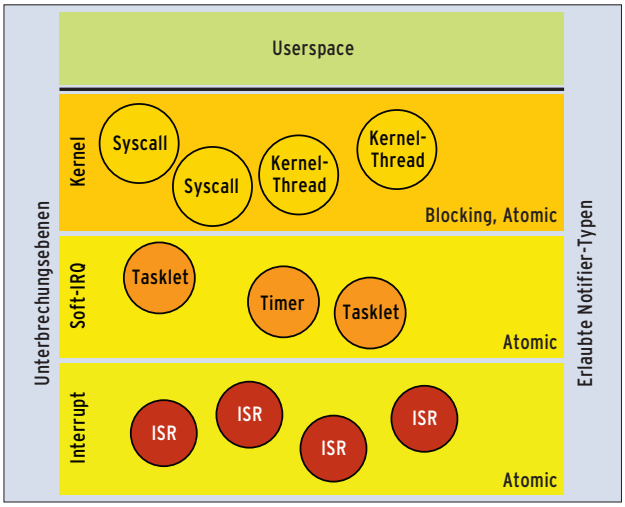


Abbildung 2: Blocking-Notifier sind auf Kernel- und Prozessebene verboten, Atomic-Notifier jedoch erlaubt.

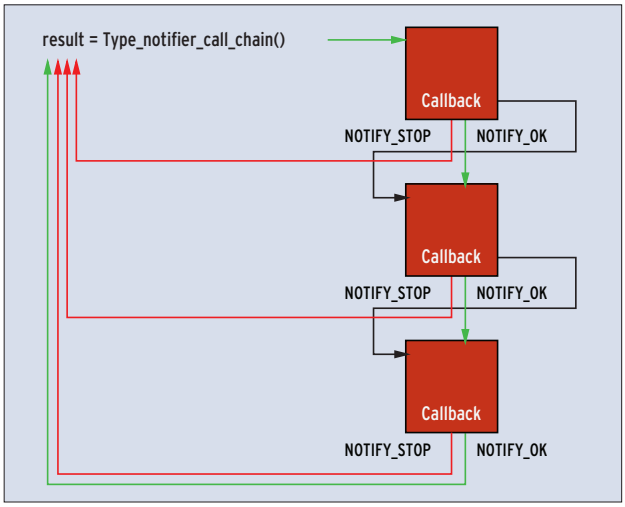


Abbildung 3: Sobald der erste Callback »NOTIFY_STOP« zurückliefert, bricht das Notifier-Subsystem die weitere Verarbeitung von Callbacks ab.

tifier-Chain dürfen Entwickler das Aushängen eines Callback nicht in einem solchen selbst implementieren.

Listing 1 zeigt beispielhaft, wie eigener Kernelcode Notifier-Chains sowohl als Provider als auch als Konsument nutzt. Für den Provider sind nur zwei Dinge zu implementieren: Erstens definiert und initialisiert er mit Hilfe der Makros »*Typ*_NOTIFIER_HEAD« das erste Element der Liste. Die Headerdatei »linux/notifier.h« deklariert sie und lässt als Typ »raw«, »atomic« und »blocking« zu.

Handelt es sich um eine SRCU-Chain ist nur die Initialisierung zur Laufzeit erlaubt: Dazu verwendet der Entwickler die Funktion »srcu_init_notifier_head()«. Zweitens beginnt er bei Eintritt des Ereignisses, die Callback-Liste abzuarbeiten. Dazu ruft er die Funktion »*Typ*_notifier_call_chain()« auf. Diesmal ist als Typ zusätzlich »srcu« erlaubt.

Um als Konsument eine Informationsquelle anzuzapfen, muss die Adresse der vom Provider definierten Notifier-Chain bekannt sein. Außerdem legt er eine Datenstruktur an, in der er die Adresse der Callback-Funktion einträgt (Listing 1, Zeile 18). Mit dieser Information implementiert er das Abonnieren in Zeile 23 und das Aushängen in Zeile 36.

Die vom Konsumenten ebenfalls zu implementierende Callback-Funktion gibt einen der Werte aus Tabelle 2 zurück. Der Code leitet den Rückgabewert der zuletzt aufgerufenen Callbacks schließlich dem Provider als Returncode der »*Typ*_notifier_call_chain()«-Routine weiter. Programmierer verwenden die Codes »NOTIFY_STOP« oder »NOTIFY_BAD«, um sicherzustellen, dass der Fehlercode ihrer Routine auch beim Provider ankommt. Sie bewirken nämlich, dass das Notify-Subsystem keine weiteren ausstehenden Callbacks mehr aufruft (siehe Abbildung 3). Dadurch ist ihre Routine automatisch die zuletzt aufgerufene. Sie können »NOTIFY_STOP« und »NOTIFY_BAD« auch noch mit einem eigenen Rückgabewert kombinieren. Die Makros »notifier_from_errno()« und »notifier_to_errno()« aus »linux/notifier.h« helfen die Werte zu wandeln.

Listing 2: Netdev liefert Interface-Informationen (»inet.c«)

```
01 #include <linux/module.h>
02 #include <linux/notifier.h>
03 #include <linux/inetdevice.h>
04
05 static int
06 inet_callback(struct notifier_block *self,
07               unsigned long val, void *data)
08 {
09     printk("inet_callback: Val=%ld, IP=%x\n",
10            val, ntohl(((struct in_ifaddr *)
11                        data)->ifa_address));
12     return NOTIFY_DONE;
13 }
14 static struct notifier_block inet_notifier = {
15    .notifier_call = inet_callback,
16 };
17
18 static int __init inet_init (void)
19 {
20     register_inetaddr_notifier(&inet_notifier);
21     return 0;
22 }
23
24 static void __exit inet_exit(void)
25 {
26     unregister_inetaddr_notifier(
27                &inet_notifier);
28 }
29
30 module_init(inet_init);
31 module_exit(inet_exit);
32 MODULE_LICENSE("GPL");
```

User Generated Content

In der Datenstruktur, die die Adresse des Callback aufnimmt, darf der Programmierer zusätzlich eine Priorität vergeben. Sie bewegt sich im Bereich von null bis »INT_MAX«, also 0x7fffffff. Ein hoher Wert spiegelt hohe Priorität wider. Nach ihm sucht sich die Funktion ihren Platz in der Liste. Entsprechend der Reihenfolge ruft Linux bei Eintritt des Ereignisses die Routinen auf. Durch eine Priorität von »INT_MAX« platzieren Programmierer einen Rückruf ganz vorne. Den Kernel- und Treiberprogrammierer interessiert nicht nur das generische Notifier-Framework, sondern auch die darüber gehandelten Informationen, denn viele Subsysteme nutzen den Mechanismus.

Tabelle 2: Callback-Rückgabewerte

Makro	Bedeutung
NOTIFY_DONE	Callback abgearbeitet
NOTIFY_OK	Callback abgearbeitet
NOTIFY_STOP_MASK	Letzter Callback
NOTIFY_BAD	(»NOTIFY_STOP_MASK | 0x0002«)

Leider dokumentiert der aktuelle Kernel bis dato nicht, welche Nachrichtenquellen und Nachrichtentypen er definiert. Tabelle 3 listet die wichtigsten Subsysteme inklusive der zugehörigen Funktionen auf. Im Zweifelsfall hilft ein Blick in den zugehörigen Code. Entwickler sollten beachten, dass eine Nachrichtenquelle oftmals verschiedene Nachrichtentypen umsetzt. Der Kern erlaubt es nicht, auf Basis der Nachrichtentypen zu filtern. Auch wer nur eine Art von Nachrichten benötigt, zum Beispiel den Typ „Neue IP-Adresse“, muss zunächst sämtliche von ihnen entgegennehmen.

Schwieriger Abo-Service

Listing 2 demonstriert, wie Schreiber eigener Routinen eine vorhandene Nachrichtenquelle abonnieren, in diesem Fall die des Netzgeräte-Subsystems Netdev. Der Code zeigt, dass Subsystem-Maintainer die Adresse der Datenstruktur nicht selbst exportieren, sondern oftmals nur gekapselte Registrierungsfunktionen. Leider geht aus dem Namen der Registrierungsfunktion nicht hervor, um welchen Notifier-Chain-Typ (Raw, Atomic, Blocking, SRCU) es sich handelt.

Wer den Code mit Hilfe eines geeigneten Makefile auf einem Kernel ab 2.6.29 übersetzt und mit dem Kommando »insmod inet.ko« lädt, erhält im Eventlog in »/var/log/messages« immer dann eine Nachricht, wenn Netdev eine IP-Adresse neu vergibt. Weitere Beispiele, die den Rückgriff auf vorhandene Informationen demonstrieren, enthalten frühere Ausgaben der Kerntechnik ([1], [2]). *(mg)* ■

Tabelle 3: Notifier im Kernel 2.6.29

Interface	Beschreibung
Kernel-Core, Typ Blocking für die Reboot-, Atomic für die Oops-Notifier	
register_reboot_notifier unregister_reboot_notifier	Shutdown, Halt, Power-off Erfordern Typ Bocking
register_die_notifier unregister_die_notifier	Kernel-Absturz (Oops) Erfordern Typ Atomic
Power-Management, Typ Blocking	
register_pm_notifier unregister_pm_notifier	Hibernation, Suspend, Restore
CPU, Typ Raw für die CPU-, Blocking oder SCRU für die Takt-Notifier	
register_cpu_notifier unregister_cpu_notifier	Up (Online), Down, Defekt Erfordern Typ Raw
cpufreq_register_notifier cpufreq_unregister_notifier	Wechsel von Taktfrequenz und Policy Wahlweise Typ Blocking oder SCRU
Modul-Handling, Typ Blocking	
register_module_notifier unregister_module_notifier	»MODULE_STATE_GOING«, »MODULE_STATE_COMING«, »MODULE_STATE_LIVE«
Crypto, Typ Blocking	
crypto_register_notifier crypto_unregister_notifier	»CRYPTO_MSG_ALG_REQUEST«
IP, Typ Blocking	
register_inetaddr_notifier unregister_inetaddr_notifier register_inet6addr_notifier unregister_inetaddr_notifier	»NETDEV_UP«, »NETDEV_DOWN« (siehe »notifier.h«)
Netdevice, Typ Raw	
register_netdevice_notifier unregister_netdevice_notifier	»NETDEV_CHANGE«, Adressenänderung, MTU, Name, Feature, Failover, Up, Down
Netevent, Typ Atomic	
register_netevent_notifier unregister_netevent_notifier call_netevent_notifiers	Update der Nachbarschaftsbeziehungen (»NETEVENT_NEIGH_UPDATE«)
Netlink, Typ Atomic	
netlink_register_notifier netlink_unregister_notifier	»NETLINK_URELEASE«
Firewall, Typ Atomic	
nf_conntrack_register_notifier nf_conntrack_unregister_notifier nf_ct_expect_register_notifier nf_ct_expect_unregister_notifier	Cached Events
ACPI, Typ Blocking	
register_acpi_notifier unregister_acpi_notifier register_acpi_bus_notifier unregister_acpi_bus_notifier	Power-Fault, Device-Check, Bus-Check, Wakeup, Eject
Memory, Typ Blocking	
register_memory_notifier unregister_memory_notifier	Online, Offline
Keyboard, Typ Atomic	
register_keyboard_notifier unregister_keyboard_notifier	Key-Codes
OOM, Typ Blocking	
register_oom_notifier unregister_oom_notifier	Pagefault, Out-of-Memory
Virtual Terminal, Typ Atomic	
register_vt_notifier unregister_vt_notifier	Schreiben, Update
Sonstiges, Typ Blocking	
regulator_register_notifier regulator_unregister_notifier	Unterspannung
bus_register_notifier bus_unregister_notifier	Gerät hinzufügen, Gerät entfernen

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, Kern-Technik, Folge 27, „Umgang mit Zeiten“: Linux-Magazin 04/06, S. 118

[2] Eva-Katharina Kunst und Jürgen Quade, Kern-Technik, Folge 45, „Treiber-Watchdogs“: Linux-Magazin 05/09, S. 96

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 47

Kern-Technik

Sicherheitsbewusste Kernelhacker beäugen skeptisch Netzwerkapplikationen, die in den Kernel wandern sollen. Manchmal gibt's dafür aber gute Gründe – Performance etwa. Eva-Katharina Kunst, Jürgen Quade



Wenn Anwender besondere Anforderungen an den Leistungsdurchsatz legen, kann die Verlagerung von Code aus dem Userland in den Kernel sinnvoll sein. Die Maßnahme spart Kontextwechsel-Zeiten ebenso wie aufwändige Kopieraktionen zwischen User- und Kernelspace. Allerdings hebt sie die sinnvolle Trennung von Applikation und Kernel auf. Die Applikation im Kernel hebelt die sonst vorhandenen Schutzmechanismen aus, öffnet im Fall von Programmierfehlern Crackern ein direktes Tor zum Kernel und gerät daher stets zur Speziallösung.

Dennoch lässt sich die Leistungssteigerung für eigene Netzwerkanwendungen im Kernel nutzen. Kleiner Nebeneffekt der Technik: Sämtliche Komponenten einer solchen Anwendung lassen sich in einem Stück Software, nämlich einem Kernelmodul, verpacken. Wo früher also ein oder mehrere Applikationsteile zu installieren waren, lädt der Anwender nur noch ein distributionsunabhängiges Kernelmodul nach.

Doch ist eine Applikation im Kern nicht so einfach zu programmieren wie eine Userspace-Anwendung, weil Linux von dort keinen Zugriff auf normale Bibliotheksfunktionen ermöglicht [1]. Spannenderweise haben die Kernelentwickler gerade für den Netzwerkzugriff aber einen ganzen Reigen von Funktionen geschrieben. Den dürfen sogar Programmierer nutzen, die ihre Software nicht unter die GPL stellen.

Die Funktionen kreieren und löschen Sockets, binden sie an Ports, warten auf eingehende Verbindungen oder bauen solche auf (siehe Tabelle 1). Aus der Bandbreite der Userspace-Funktionen steht also auch im Kernel das Wichtigste zur Verfügung (siehe Abbildung 1).

Ähnlich, aber nicht gleich

Entwicklern fallen vier Umstände auf: Erstens haben die Kernel-Pendants andere Namen im Vergleich zu ihren Userspace-Geschwistern, zweitens sind die Aufrufparameter unterschiedlich. Der dritte Unterschied betrifft die Funktionen »read()« und »write()« zum Datenaustausch: Diese Funktionen sucht der Entwickler im Kern vergeblich und muss sie selbst implementieren. Dazu bieten viertens die Funktionen »socket_sendmsg()« und »kernel_sendmsg()« ihre Dienste an – mit auf den ersten Blick ähnlicher Funktionalität.

Einfach anwenden lassen sich die Funktionen »kernel_sendmsg()« und »kernel_recvmsg()«. Beiden übergibt der Kernel-Netzwerker neben dem Socket noch eine Datenstruktur vom Typ »struct msghdr«, die bei verbindungslosen Sockets unter anderem auch die Zieladresse für das Paket enthält (siehe Abbildung 2). Die Daten müssen nicht zwangsläufig an einem zusammenhängenden Speicherort liegen. Vielmehr übergibt der Entwickler ein Feld vom Typ »struct kvec«, das Adressen und Längen der zugehörigen Speicherorte enthält. »kernel_sendmsg()« und »kernel_recvmsg()« kümmern sich darum, dieses Feld in die »struct msghdr« einzuhängen.

Variantenreich

Bei den Funktionen »sock_sendmsg()« und »sock_recvmsg()« hängt der Programmierer die Speicherort-Adressen – jetzt vom Typ »struct iovec« – vor dem Aufruf in den Messageheader ein. Das allein ist aber nicht der entscheidende Unterschied. Systemcalls wie die zwei »sock_«-Funktionen sind qua Implementierung darauf geeicht, Daten zwischen Kernel- und Userspace auszutauschen.

Der Transfer von Kernel zu Kernel ist hingegen nur in seltenen Fällen notwendig. Daher muss der Entwickler vor dem Aufruf der zum Systemcall gehörenden Kernelfunktion die Speicherverwaltung mit einem Trick überlisten. Es gilt, sie zu überreden auch aus dem Kernel selbst einen Zugriff zu gestatten.

Das zugrunde liegende Problem und die Lösung stellt der Kasten „Datentransfer innerhalb des Kernels“ vor. Falls sich die in »iov« spezifizierten Speicherorte im Kernelspace befinden und der Entwickler den Systemaufruf »sock_sendmsg()« verwendet, bettet er ihn in die in Listing 1 dargestellte Codesequenz ein.

Außerdem gilt für Netzwerkcode im Linux-Kern, dass viele Zugriffe auf das

Datentransfer innerhalb des Kernels

Systemcalls, die Dienste des Betriebssystemkerns, tauschen Daten zwischen Userspace und Kernelspace vorwiegend mit den Funktionen »copy_to_user()« und »copy_from_user()« aus [2]. Sie verarbeiten Adressen im Userspace, die von der Applikation stammen. Sie sind aus Sicht des Betriebssystemkerns alles andere als vertrauenswürdig. Daher überprüft die Speicherverwaltung vor einem Datentransfer, ob die Adressen gültig sind, ob sie nicht in der so genannten Zeropage liegen oder ob die Adressen nicht ungewollt (oder von einem Malevolenten gar manipuliert) auf eine Adresse im Kernelspace zeigen. Das wäre ein Wert oberhalb der Grenze zwischen User- und Kernelspace.

Damit ein Systemcall Daten aus dem Kern akzeptiert, verschiebt der Entwickler diese Grenze kurzfristig. Linux legte sie früher im FS-Register der x86-CPU ab. Wegen dieser Geschichte heißen bis heute die Funktion, die die Grenze auslesen und einstellen, »get_fs()« und »set_fs()«. Vor dem Systemcall aus dem Kernel rettet der Programmierer also den alten Wert und erlaubt temporär mit »KERNELDS« den internen Zugriff. Er darf natürlich nach dieser Operation nicht vergessen, die ursprünglichen und sicheren Werte wieder zu reaktivieren.

Netz-Interface nur im Prozesskontext, nicht aber innerhalb einer Interrup-Service-Routine (ISR), eines Tasklet oder einer Timerfunktion statthaft ist. Das Programm muss sich also entweder den Prozesskontext einer Applikation borgen (zum Beispiel den des Programms »insmod«) oder aber mit Hilfe der Funktion »kernel_thread()« eine unabhängige Task aufziehen.

Nachsager aus dem Kernel

Listing 3 implementiert einen TCP-Echo-Server innerhalb des Kernels. Der Code zieht dafür ab Zeile 109 beim Laden des Moduls (»server_init()«) einen eigenen Thread mit der dazu notwendigen Verwaltung auf. Anschließend legt er analog zu einer normalen Netzapplikation einen Socket an, bindet ihn an einen Port (hier 5555) und setzt die Länge der Warteschlange neuer Verbindungen mittels »listen()« fest. Analog lassen sich natürlich auch andere Transportprotokolle wie UDP oder ICMP nutzen. Im Code ist übrigens erkennbar, dass im Kernel auch die Hilfsfunktionen »htons()« und ihre Geschwister zur Verfügung stehen.

Entlädt ein Anwender das Modul, dann beendet der Code ab Zeile 125 den angelegten Thread mit dem notwendigen Entladeschutz über ein Completion-Objekt. Der Code des Kernelthread selbst findet sich in der Funktion »echo_server()« ab Zeile 83. Solange für den Thread kein Signal anliegt, wartet die Funktion »server_accept()« auf eine neue Verbindung. Kommt die zustande, legt der Kernel selbst den charakterisierenden Socket an. Ihn muss er später über die Funktion »sock_release()« wieder freigegeben.

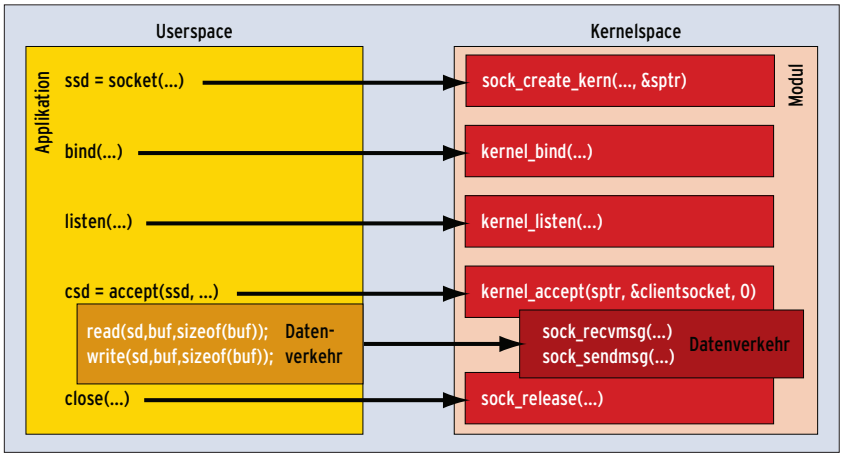


Abbildung 1: Für bekannte Userspace-Funktionen bietet der Kernel korrespondierende Routinen an.

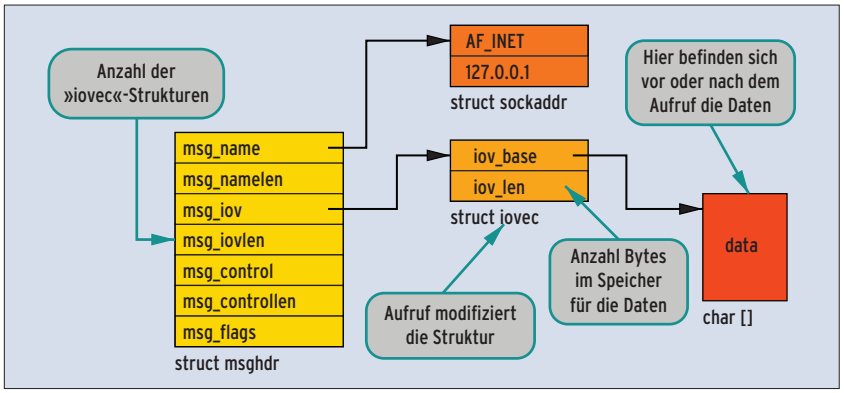


Abbildung 2: Die Datenstrukturen für den Datenaustausch erfordern genaues Hinsehen beim Programmieren.

Pakete schnüren

Solange eine neue Verbindung besteht, nimmt der Server ankommende Pakete mit der Funktion »server_receive()« entgegen und wirft die empfangenen Daten mittels »server_send()« als Echo wieder zurück. Die Implementation blockiert so lange, bis der Client die Verbindung beendet. Ernsthafte Serverapplikationen würden daher für jede Verbindung einen eigenen Thread erzeugen. Für den eigentlichen Datenverkehr baut der Program-

Listing 1: Zugriff auf Kerneldaten

```
01 msg.msg_iov    = &iov;
02 msg.msg_iovlen = 1;
03
04 oldfs = get_fs();  // Grenze sichern
05 set_fs(KERNEL_DS); // Aushebeln des Schutzmechanismus
06 len = sock_sendmsg(sock, &msg, len);
07 set_fs(oldfs); // Restauration des Ursprungszustands
```

Listing 2: Verbindungslogs

```
01 int len, error;
02
03 if ((error = kernel_getpeername(clientsocket,
04     (struct sockaddr *)&address, &len)) < 0) {
05     sock_release(clientsocket);
06     return NULL;
07 }
08 printk(KERN_INFO "%u.%u.%u.%u connected (%d)\n",
09     (unsigned char)address.sa_data[2],
10     (unsigned char)address.sa_data[3],
11     (unsigned char)address.sa_data[4],
12     (unsigned char)address.sa_data[5], error);
```

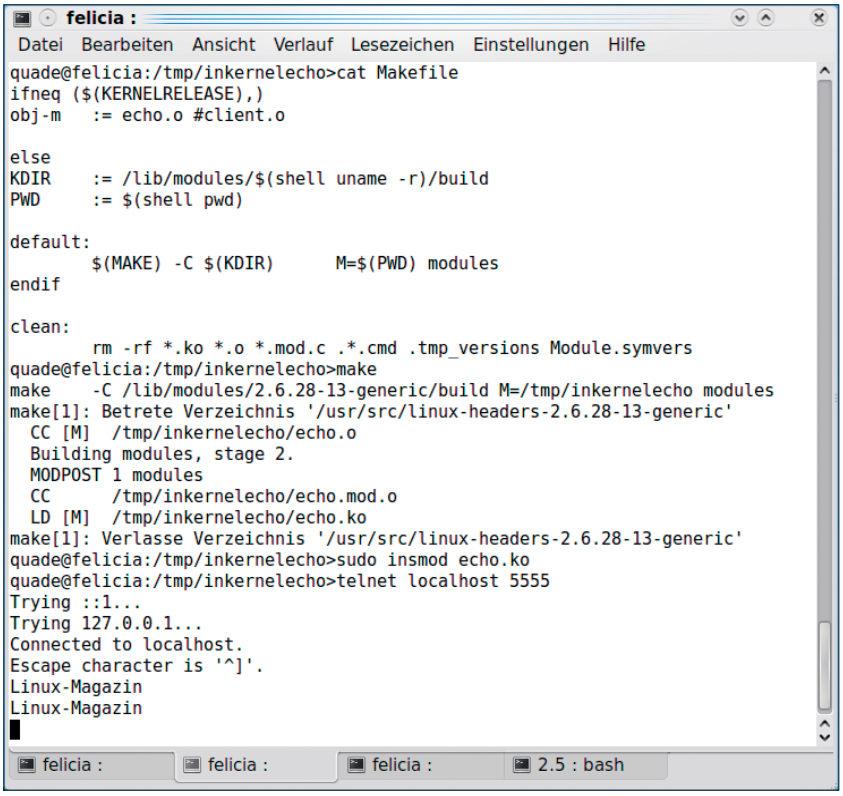


Abbildung 3: Nach dem Übersetzen und Laden lässt sich der Echo-Server mit »telnet« einfach testen.

mierer eine »struct msghdr« zusammen. Dafür benötigt er vier Teile: Erstens einen Speicherbereich »buffer«, in dem er die zu versendenden oder zu empfangenen Daten ablegt, dann Adresse und Größe dieses Speicherbereichs in einer Datenstruktur vom Typ »struct iovec« (respektive »struct kvec«). Nutzt er UDP, spezifiziert er als vierten Parameter die Zieladresse in der Struktur »struct sockkaddr«. Der Kleber dieser Komponenten ist die Struktur »struct msghdr« selbst.

Abbildung 2 illustriert die Zusammenhänge. Wer am Ende der Funktion »server_accept()« noch die Codezeilen aus Listing 2 einfügt, erfährt aus dem Syslog, wer Verbindung mit dem Server aufgenommen hat.

Ausreichend frankieren

Der Code aus Listing 3 lässt sich mit einem normalen Kernel-Makefile (siehe Abbildung 3) kompilieren. Sobald Linux das Modul geladen hat, ist der Echo-Server aktiv und lässt sich beispielsweise

Tabelle 1: Prototypen der Netzfunktionen im Kernel

Funktion	Bedeutung
int sock_create(int family, int type, int proto, struct socket res);	Legt einen Socket an
int sock_create_kern(int family, int type, int proto, struct socket res);	Legt einen Socket im Kernel-Namespace an
int sock_create_lite(int family, int type, int proto, struct socket res);	Reserviert Speicher für ein Socket-Objekt
void sock_release(struct socket *sock);	Gibt die Ressourcen des Socket wieder frei
int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t len);	Systemaufruf, der Daten verschickt, die im Userspace liegen
int sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t size, int flags);	Systemaufruf, der Daten empfängt und sie im Userspace ablegt
int kernel_sendmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num, size_t len);	Verschickt Datenfragmente, die im Kernelspace liegen; die Struktur »vec« enthält Zeiger auf die Blöcke und deren Längen
int kernel_recvmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num, size_t len, int flags);	Empfängt Daten und legt sie im Kernelspace ab
int kernel_bind(struct socket *sock, struct sockaddr *addr, int addrlen);	Stellt eine logische Verbindung zwischen einem bestehenden Socket und einer Adresse eines beliebigen Protokolls her
int kernel_listen(struct socket *sock, int backlog);	Markiert einen Socket als Empfänger und legt die Größe der zugehörigen Warteschlange eingehender Verbindungen fest
int kernel_accept(struct socket *sock, struct socket newsock, int flags);	Blockiert den Kontrollfluss, bis eine Verbindung auf dem zugehörigen Port eingeht; die neue Verbindung repräsentiert »newsock«
int kernel_connect(struct socket *sock, struct sockaddr *addr, int addrlen, int flags);	Nimmt in der Sender-Rolle aktiv Verbindung zum Verbindungspartner auf, der in »addr« festgelegt ist
int kernel_getsockname(struct socket *sock, struct sockaddr *addr, int *addrlen);	Liest Parameter des eigenen lokalen Socket aus
int kernel_getpeername(struct socket *sock, struct sockaddr *addr, int *addrlen);	Liest Parameter wie IP-Adresse eines entfernten Peer aus
int kernel_getsockopt(struct socket *sock, int level, int optname, char *optval, int *optlen);	Liest Optionen eines Socket aus. Dazu gehören etwa der Status, Buffergrößen oder der letzte detaillierte Fehlerstatus
int kernel_setsockopt(struct socket *sock, int level, int optname, char *optval, int optlen);	Setzt eine Socket-Option, etwa Logging, oder stellt ein, ob neue Prozesse einen Port wiederverwerten dürfen
int kernel_sendpage(struct socket *sock, struct page *page, int offset, size_t size, int flags);	Verschickt direkt eine Speicherseite
int kernel_sock_ioctl(struct socket *sock, int cmd, unsigned long arg);	Setzt netzwerkbezogene IO-Controls
int kernel_sock_shutdown(struct socket *sock, enum sock_shutdown_cmd how);	Beendet einseitig die Verbindung

mit Hilfe von »telnet« testen. Netzprofis achten darauf, dass sonst keine Anwendung Port 5555 verwendet. Ist die Ressource schon belegt, meldet der Code in Zeile 23 einen Fehler. Auch eine Fehlerquelle: eine blockierende Firewall.

In bewährtem Gewand

Trotz des funktionsorientierten Interface haben die Kernelentwickler den Netzwerkzugriff intern vollkommen objektorientiert implementiert. Die einzelnen Funktionen sind Wrapper-Routinen, die Methoden des jeweiligen Socket-Objekts aufrufen. Auch wenn das objektorientierte Paradigma seinen Charme hat, entspricht das mit Kernelversion 2.6.19 übernommene Interface einfach mehr dem API des Userspace, das Entwickler seit vielen Jahren kennen. Das macht das Portieren zumindest von überschaubaren Codestücken angenehm einfach. Komplexer Code hat im Kernel nach wie vor wenig zu suchen, zu weitreichend sind die Folgen möglicher Programmierfehler. Statusmeldungen eingebetteter Systeme sind hingegen einfach möglich. *(mg)* ■

Infos

[1] Richard Stevens, „Unix Network Programming", Volume 1: Prentice Hall India, 2003
[2] Quade, Kunst, „Linux-Treiber entwickeln": Dpunkt-Verlag, 2. Auflage, Juni 2006

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln" haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Listing 3: Inkernel-Echo-Server

```
001 #include <linux/module.h>
002 #include <linux/init.h>
003 #include <linux/in.h>
004 #include <net/sock.h>
005
006 static struct socket *serversock = NULL;
007 static DECLARE_COMPLETION(threadcomplete);
008 static int srvpid;
009
010 static int
011 create_socket(void)
012 {
013   struct sockaddr_in server;
014
015   if (sock_create_kern(PF_INET, SOCK_STREAM,
016         IPPROTO_TCP, &serversock) < 0) {
017     printk(KERN_ERR "server: No socket.\n");
018     return -EIO;
019   }
020   server.sin_family      = AF_INET;
021   server.sin_addr.s_addr = INADDR_ANY;
022   server.sin_port        = htons(5555);
023   if (kernel_bind(serversock,
024           (struct sockaddr *) &server,
025           sizeof(server)))
026     goto release;
027   if (kernel_listen(serversock, 3))
028     goto release;
029   return 0;
030
031 release:
032   sock_release(serversock);
033   printk(KERN_ERR "server: socket error\n");
034   return -EIO;
035 }
036
037 static struct socket *
038 socket_accept(struct socket *server)
039 {
040   struct sockaddr address;
041   struct socket *clientsock = NULL;
042
043   if (!server) return NULL;
044   if (kernel_accept(server,
045         &clientsock, 0) < 0) return NULL;
046   return clientsock;
047 }
048 static int
049 server_send(struct socket *sock,
050             unsigned char *buf, int len)
051 {
052   struct msghdr msg;
053   struct kvec iov;
054
055   if (!sock->sk) return 0;
056   iov.iov_base       = buf;
057   iov.iov_len        = len;
058   msg.msg_control    = NULL;
059   msg.msg_controllen = 0;
060   msg.msg_flags      = 0;
061
062   return kernel_sendmsg(sock, &msg,
063                         &iov, 1, len);
064 }
065
066 static int
067 server_receive(struct socket *sptr,
068                unsigned char *buf, int len)
069 {
070   struct msghdr msg;
071   struct kvec iov;
072
073   if (!sptr->sk) return 0;
074   iov.iov_base       = buf;
075   iov.iov_len        = len;
076   msg.msg_control    = NULL;
077   msg.msg_controllen = 0;
078
079   return kernel_recvmsg(sptr, &msg,
080                         &iov, 1, len, 0);
081 }
082
083 static int
084 echo_server(void *data)
085 {
086   struct socket *clientsock;
087   unsigned char buff[1024];
088   static int len;
089
090   daemonize("kecho"); allow_signal(SIGTERM);
091   while (!signal_pending(current)) {
092     clientsock = socket_accept(serversock);
093     printk("clientsocket(%p)\n",
094            clientsock);
095     while (clientsock) {
096       if ((len = server_receive(clientsock,
097             buff, sizeof(buff))) > 0) {
098         server_send(clientsock, buff, len);
099       } else {
100         sock_release(clientsock);
101         clientsock = NULL;
102       }
103     }
104   }
105   complete(&threadcomplete);
106   return 0;
107 }
108
109 static int
110 __init server_init(void)
111 {
112   if (create_socket() < 0) return -EIO;
113
114   srvpid = kernel_thread(echo_server,
115                          NULL, CLONE_KERNEL);
116   printk("echo server pid: %d\n", srvpid);
117   if (srvpid < 0) {
118     printk(KERN_ERR "server: no thread\n");
119     sock_release(serversock);
120     return -EIO;
121   }
122   return 0;
123 }
124
125 static void
126 __exit server_exit(void)
127 {
128   printk("server_exit()\n");
129
130   if (srvpid) {
131     kill_pid(find_pid_ns(srvpid,
132              &init_pid_ns), SIGTERM, 1);
133     wait_for_completion(&threadcomplete);
134   }
135   if (serversock)
136     sock_release(serversock);
137 }
138
139 module_init(server_init);
140 module_exit(server_exit);
141 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 48

Kern-Technik

Einen Treiber in den Kernel einzubinden ist leider wenig intuitiv. Deshalb gibt es hier ein Grundgerüst als Ausgangspunkt. Es zeigt, wie der Kernel dem Treiber Gerätenummern zuweist und der Gerätedateiverwalter Udev eigenständig die zugehörigen Gerätedateien anlegt. Eva-Katharina Kunst, Jürgen Quade



Das einfachste Programm des Kernels, sein „Hello World“, ist übersichtlich: 16 Zeilen Code, zwei Funktionen und ein passendes Makefile reichen aus, um Selbstgeschriebenes mit dem Kernel zu verknüpfen (Listing 1). Als Ergebnis steigt im Syslog der vom C-Erfinder Brian Kernighan entwickelte Willkommensgruß aus den Tiefen des Betriebssystem auf (Abbildung 1).

Der kurze Code ist leicht nachzuvollziehen. Die erste Funktion (Zeile 3) ruft der Prozessor beim Laden, die zweite beim Entladen des Programms auf (Zeile 9). »printk()« (Zeile 5) steht für „Print Kernel“ und entspricht »printf()« in anderen Programmen. Für den Kernel-Neuling bleiben lediglich die ungewöhnlichen Schlüsselwörter »__init« (Zeile 3) und »__exit« (Zeile 9) und die Makros »module_init()«, »module_exit()« und »MODULE_LICENSE()« (Zeilen 14 bis 16) erklärungsbedürftig.

Listing 1: Hallo Kernelwelt (»hellofromkernel.c«)

```
01 #include <linux/module.h>
02
03 static int __init template_init(void)
04 {
05   printk("Hello world\n");
06   return 0;
07 }
08
09 static void __exit template_exit(void)
10 {
11   return;
12 }
13
14 module_init(template_init);
15 module_exit(template_exit);
16 MODULE_LICENSE("GPL");
```

Hallo Kernelwelt

»__init« klassifiziert die zugehörige Funktion als eine Initialisierungsfunktion, die nur ein einziges Mal in Aktion tritt, nämlich beim Laden des Code. Linux entfernt sie nach dem Aufruf und gibt den ansonsten unnütz belegten Speicher frei. Eine über »__exit« kategorisierte Funktion wandert unter bestimmten Umständen gar nicht erst in den Hauptspeicher. In beiden Fällen optimiert der Kernel den Programmablauf.

Die beiden Makros »module_init()« und »module_exit()« erlauben es dem Programmierer, die Namen der Modul-Initialisierungs- und Deinitialisierungsfunktion frei zu wählen. Der Programmtext des entstehenden Kernelmoduls wird dadurch besser lesbar. »MODULE_LICENSE()« schließlich verknüpft den Treibercode mit der vom Programmierer gewählten Lizenz (Zeile 16). Mit einer proprietären Lizenz steht ihm nur eingeschränkte Kernelfunktionalität zur Verfügung. Wer sich aber zur freien Software bekennt, greift auf alle vom Kernel bereitgestellten Routinen zu.

Garniert mit ein paar weiteren Codezeilen mutiert das vorliegende Hello-World-Kernelmodul schnell zu einem Gerätetreiber, auf den sein Schöpfer mit den Applikationsfunktionen »open()«, »close()«, »read()« und »write()« zugreift. Dazu erstellt er die Liste »struct file_operations« (Listing 2, Zeile 6), die die Adressen der im Treiber aufzurufenden Funktionen »driver_open()«, »driver_close()«, »driver_read()« und »driver_write()« zusammenfasst. Mit dieser Liste meldet sich der Kernelprogrammierer beim IO-Subsystem des Kernels an, indem er die Funktion »register_chrdev()« (Zeile 11) aufruft und eine Treiber-Identifikationsnummer angibt. Das spätere Abmelden übernimmt die Funktion »unregister_chrdev()« in Zeile 19.

Arbeitslos

Die Liste kann – wie in diesem Beispiel – leer sein. Dann erfüllt der Treiber einfach keine Funktion. Die erwähnte Identifikationsnummer oder Majornummer verbindet sich im Applikationsbereich durch Aufruf des Kommandos »mknod« mit einer Gerätedatei. Den Namen dieser Gerätedatei bekommt die Applikationsfunktion »open()« als ersten Parameter übergeben (Abbildung 2).

Mit Kernel 2.6 hat Linus Torvalds die antiquierten und in ihrer Zahl begrenzten

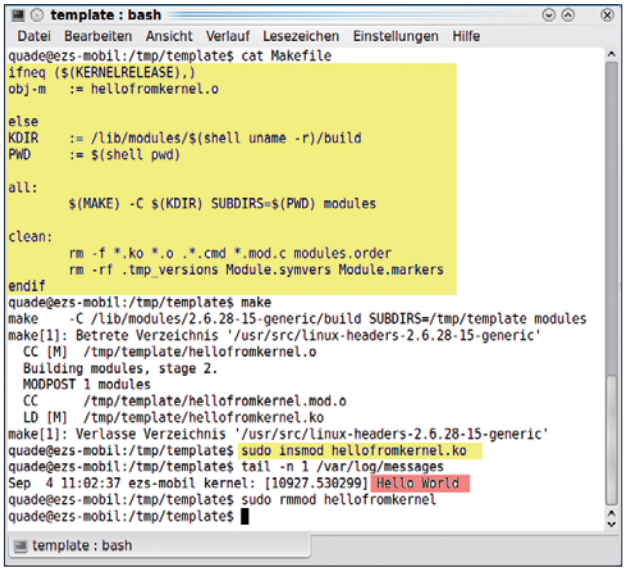


Abbildung 1: Mit einem geeigneten Makefile lässt sich aus dem Code von Listing 1 ein Kernelmodul generieren. Das Kommando »insmod« lädt das Modul in den Kern. Bei Erfolg grüßt der Kernel die Welt in seinen Messages mit einem Hallo.

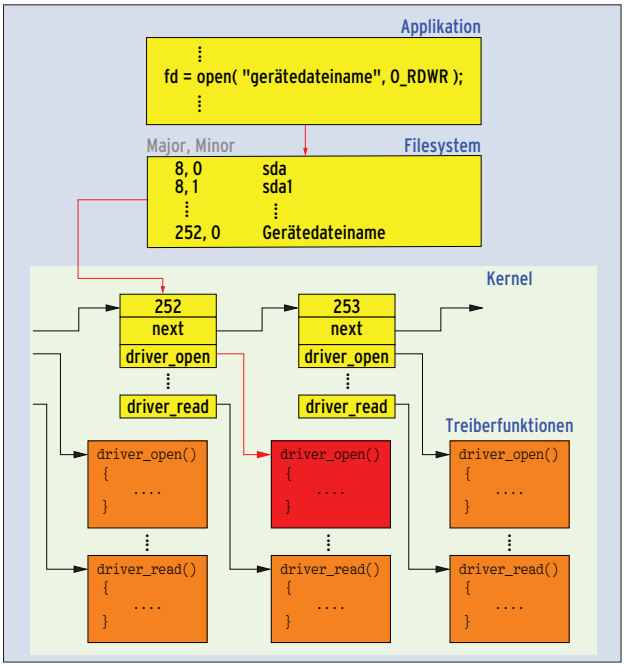


Abbildung 2: Die Majornummer ist der Angelpunkt für die Auswahl der aufzurufenden Treiberfunktion. Sie verbindet die Gerätedatei mit der Start-Funktion.

Majornummern gegen moderne Gerätenummern ausgetauscht. Für sie baute der Treiber-Chef Greg Kroah-Hartman ein neues Interface, mit dem sich Treiber beim IO-Subsystem des Kernels anmelden. Das neue Interface wirkt offenbar auf viele Entwickler weder intuitiv noch elegant. Sie greifen daher weiterhin auf die eben genannte und bewährte »register_chrdev()«-Methode zurück. Auch wenn alles wie hier beschrieben programmierbar und funktionstüchtig ist, so ist dies nicht im Sinne der Erfinder, moderne Treibereinbindung sieht anders aus.

Modern mit Udev

Listing 3 stellt ein universell einsetzbares Treibergerüst nach dem aktuellen Verfahren dar. Für diese von Kernelhackern empfohlene Methode reserviert der Aspirant zunächst einen Bereich von Gerätenummern (siehe Abbildung 3) – wenn nur ein Gerätetyp avisiert ist, dann ist auch nur eine Gerätenummer nötig. Zum Reservieren stehen die Funktionen »alloc_chrdev_region()« und »register_chrdev_region()« zur Verfügung (die Funktionen erklärt kurz Tabelle 1 am Ende des Artikels).

Der Unterschied: Bei »alloc_chrdev_region()« teilt der Kernel dem Treiber einen Bereich von Gerätenummern zu. Bei »register-chrdev_region()« fordert der Treiber hingegen selbst einen bestimmten Bereich an. Dank des neuen Gerätedatei-Verwalters Udev [1], der automatisch Gerätedateien erzeugt, ist es aber kaum noch nötig, mit vorbestimmten Gerätenummern zu arbeiten. Die Funktion »alloc_chrdev_region()« gilt daher als erste Wahl (ab Zeile 20).

Mit der einen Gerätenummer oder einem Bereich davon meldet der Programmierer den Treiber beim Kernel an. Dazu ist Speicher zu reservieren und zu initialisieren, im Fachjargon: Das Treiberprojekt ist zu instanzieren. Die Funktionen »cdev_alloc()« oder »kmalloc()« dienen wahlweise dazu, Speicher zu reservieren. Falls »kmalloc()« zum Einsatz kommt, findet die Initialisierung mit der Funktion »cdev_init()« statt. Sie nimmt als Parameter neben dem Treiberobjekt die Liste der Treibermethoden (»struct file_operations«) entgegen. Wenn der Programmierer hingegen »cdev_alloc()« verwendet, ordnet er die Liste der Treibermethoden direkt dem zugehörigen Strukturelement zu (Zeile 25). Um das Entladen des Treibers zu unterbinden, so lange dieser noch genutzt wird, weist er außerdem dem Attribut »owner« das vom Kernel vordefinierte Makro »THIS_MODULE« zu (Zeile 30). »THIS_MODULE« repräsentiert die Adresse des Modulobjekts, womit der Entladeschutz in Kraft tritt.

Einlass gewähren

Jetzt ist das Treiberobjekt so weit vorbereitet, um es dem Kernel durch Aufruf der Funktion »cdev_add()« zu übergeben (ab Zeile 32). Läuft diese Funktion erfolgreich

Listing 2: Der erste Treiber (»driver.c«)

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03
04 static int major;
05
06 static struct file_operations fops = {
07 };
08
09 static int __init template_init(void)
10 {
11   if ((major = register_chrdev(0, "template",
12                                &fops)) == 0) {
13     return -EIO;
14   }
15   return 0;
16 }
17
18 static void __exit template_exit(void)
19 {
20   unregister_chrdev(major, "template");
21   return;
22 }
23
24 module_init(template_init);
25 module_exit(template_exit);
26 MODULE_LICENSE("GPL");
```

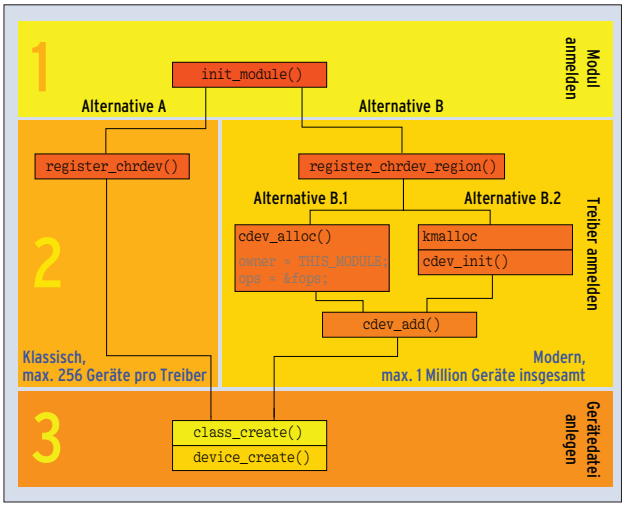


Abbildung 3: Ein Treibergrundgerüst lässt sich klassisch oder nach der weniger intuitiven, aber modernen Variante aufbauen.

Kobjects – Kernelobjekte

Der Linux-Kernel stellt Anwendungsprogrammen über das Sys-Filesystem eine Sicht auf die vorhandene Hardware und auf die sie bedienenden Softwarekomponenten (Treiber) zur Verfügung. Die vorhandene Hardware wird durch Ordner und Dateien modelliert, die im Kern selbst über Objekte repräsentiert sind. Diese Objekte ordnen sich sämtlich um das Kernelobjekt »kobject« herum an.

Objektorientiert ausgedrückt: Sys-Filesystem-Objekte leiten sich von Kobject ab. Das Kobject enthält die zentralen gemeinsamen Attribute, allen voran einen Referenzzähler, der die Freigabe des zum Objekt gehörenden Speichers koordiniert. Die Funktion »kobject_get()« inkrementiert den Zähler, ihre Schwester »kobject_put()« dekrementiert den Zähler und führt bei Bedarf die Freigabe durch.

ab, ist der Treiber im System etabliert. Andernfalls ist das Aufräumen des Speichers angesagt: »kobject_put()« (ab Zeile 45) gibt das Treiberobjekt frei (siehe Kasten „Kobjects – Kernelobjekte“) und die Funktion »unregister_chrdev_region()« (ab Zeile 47) die Gerätenummer.

Apropos Aufräumen: Am Ende, beim Entladen des Treibers, laufen alle seine Aktionen rückwärts ab. Als Erstes meldet die Funktion »cdev_del()« den Treiber beim Kernel ab (Zeile 61). Hier ist darauf zu achten, dass der Speicher des zugehörigen Treiberobjekts sofort mit dem Aufruf der Funktion frei wird, sofern kein anderes Kernelobjekt noch darauf zugreift. Keinesfalls sollte explizit die Funktion »kobject_put()« in den Editor wandern: »cdev_del()« verwendet intern bereits diese Funktion, der Speicher würde also zweimal mit kaum vorhersehbaren Effekten freigegeben. Als Zweites gibt ab Zeile 62 die Funktion »unregister_chrdev_region()« den Bereich der Gerätenummern wieder frei.

Nicht vergessen: Gerätedatei-Eintrag

Obwohl der Treiber im System verankert ist, kann eine Applikation noch nicht auf ihn zugreifen. Dafür muss erst ein Gerätedatei-Eintrag vorhanden sein. Da das hier verwendete Treibergerüst universell einsetzbar sein soll, lässt es sich die Gerätenummer vom System zuteilen. Daher muss auch der Eintrag als Gerätedatei dynamisch erfolgen.

Diese Aufgabe übernimmt seit einiger Zeit der Daemon Udevd. Allerdings erwartet er, dass der Treiber dabei mit anpackt. Dieser kennt seine Gerätenummern und erstellt für jede gewünschte Gerätedatei ein Verzeichnis unterhalb des Systemverzeichnisbaums »/sys/class/«. Udevd überwacht hier ständig die Einträge. Sobald ein neues Unterverzeichnis mit dem Namen des unterstützten Gerätetyps und darin eine neue Datei mit dem Namen »dev« auftaucht, wird – gemäß Konfiguration – vom Udevd eine entsprechende Gerätedatei angelegt.

Listing 3: Modernes Treibergerüst (»driver_template.c«)

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <linux/cdev.h>
04 #include <linux/device.h>
05
06 #define TEMPLATE "template"
07
08 static dev_t template_dev_number;
09 static struct cdev *driver_object;
10 struct class *template_class;
11 static int device_type_nr=1;
12 static struct file_operations fops = {
13   // Adressen der Treiberfunktionen ablegen
14 };
15
16 static int __init template_init(void)
17 {
18   int min;
19   // Als Treiber anmelden
20   if (alloc_chrdev_region(
21               &template_dev_number, 0,
22               device_type_nr, TEMPLATE) < 0)
23     return -EIO;
24   // Anmeldeobjekt reservieren
25   driver_object = cdev_alloc();
26
27   if (driver_object == NULL )
28     goto free_device_number;
29
30   driver_object->owner = THIS_MODULE;
31   driver_object->ops = &fops;
32   if (cdev_add(driver_object,
33                template_dev_number, 1))
34     goto free_cdev;
35
36   // Gerätedateien anlegen
37   template_class = class_create(THIS_MODULE,
38                                 TEMPLATE);
39   for (min = 0; min < device_type_nr; min++)
40     device_create(template_class, NULL,
41                   template_dev_number + min,
42                   NULL, "%s%d", TEMPLATE, min);
43   return 0;
44
45 free_cdev:
46     kobject_put(&driver_object->kobj);
47 free_device_number:
48     unregister_chrdev_region(
49                         template_dev_number, 1);
50   return -EIO;
51 }
52
53 static void __exit template_exit(void)
54 {
55   int min;
56
57   for (min = 0; min < device_type_nr; min++)
58     device_destroy(template_class,
59                    template_dev_number + min);
60   class_destroy(template_class);
61   cdev_del(driver_object);
62   unregister_chrdev_region(
63          template_dev_number, device_type_nr);
64   return;
65 }
66
67 module_init(template_init);
68 module_exit(template_exit);
69 MODULE_LICENSE("GPL");
```

Praktischerweise kann der Name des Unterverzeichnisses die Form eines Formatstrings inklusive der zugehörigen Parameter annehmen, so wie es jeder Programmierer von der Funktion »printf()« kennt. Wer den modernen Weg der Gerätenummern zugunsten des alten Schemas links liegen ließ, erzeugt mit Hilfe des Makros »MKDEV()« aus der Major- und Minornummer die Gerätenummer. Das Aufräumen erfolgt wieder in umgekehrter Reihenfolge: Als Erstes baut »device_destroy()« die Geräte-Einträge wieder zurück, dann entfernt »class_destroy()« die Sysfs-Klasse (Zeilen 57 bis 60).

Was bleibt

Listing 3 integriert den Treiber in den Linux-Kernel und informiert den Udev-Daemon genug, damit er die Gerätedatei anlegt. Nun fehlen noch die eigentlichen Treiberfunktionen, die gefragt sind, wenn eine Applikation »open()«, »close()«, »read()« oder »write()« aufruft. Die durch die Namen repräsentierten Adressen der zugehörigen Treiberfunktionen »driver_open()«, »driver_read()«, »driver_write()« oder »driver_close()« gehören in Zeile 13. Ebenfalls muss Powermanagement noch in den Programmtext wandern.

Leider ist das Sys-Filesystem nicht frei von Komplikationen. Der Programmierer ist nämlich fürs Erstellen und Löschen der gewünschten Device-Dateien zuständig. Zudem muss er sie in das richtige Sys-FS-Verzeichnis unterhalb des Klassenordners einsortieren. Im einfachsten Fall legt er dazu mit »class_create()« (Zeile 37) sein eigenes Klassenverzeichnis im Sys-FS an, in das er die Gerätedatei-Einträge schreibt. Die Funktion erzeugt das Klassenobjekt und gibt eine Referenz darauf zurück. Die Funktion »device_create()« hinterlässt für jedes Gerät einen Sys-FS-Eintrag. Sie braucht als Argumente die Adresse des Klassenobjekts, die Gerätenummer und den Namen, den die Gerätedatei später tragen soll.

Soll der Treiber mehrere Gerätetypen unterstützen, übernimmt das die Variable »device_type_nr«. Wer Hardware anvisiert, die etwa über PCI oder USB angeschlossen ist, meldet den Treiber noch beim zugehörigen Kernelsubsystem für PCI respektive USB an. Schließlich möge jeder Schreiber das Treibergerüst personalisieren, indem er den allgemeinen Namen »template« gegen einen sprechenderen austauscht (Zeile 6). Nun steht umfangreichen Kernel-Messages – oder auch sinnvollen Modulen – nichts mehr im Wege: Happy Hacking! *(ake/mg)* ■

Infos:

[1] Oliver Frommel, Jens-Christoph Brendel, „Wie dynamisches Gerätemanagement mit Udev funktioniert“: Linux-Magazin 09/06, S. 32

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux-Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Tabelle 1: Kurzübersicht wichtiger Funktionen

Funktion	Beschreibung
int alloc_chrdev_region(dev_t *devicenumbervar, unsigned minor_start, unsigned minor_count, const char *drivername);	Reserviert den nächsten freien Bereich Gerätenummern. Die Startadresse des Bereichs befindet sich nach Aufruf in »devicenumbervar«.
struct cdev *cdev_alloc(void);	Reserviert ein Verwaltungsobjekt, das einen Treiber beim Kernel anmeldet. Zur späteren Freigabe des beim Aufruf der Funktion reservierten Speichers wird »kobject_put()« oder »cdev_del()« verwendet, abhängig davon, ob »cdev_add()« erfolgreich abgelaufen ist oder nicht.
int cdev_add(struct cdev *driver_object, dev_t device_nr, unsigned count);	Registriert den Treiber beim Kernel für die definierten Gerätenummern.
int cdev_del(struct cdev *driver_object);	Meldet den zuvor erfolgreich angemeldeten Treiber beim Kernel ab. Die Funktion verwendet intern »kobject_put()« zur Speicherfreigabe.
void cdev_init(struct cdev *cdev, struct file_operations *fops);	Grundinitialisierung des Treiber-Verwaltungsobjekts.
struct class * class_create(struct module *owner, const char *name);	Erzeugt im Sys-Filesystem unterhalb des Klassenordners die Klasse »name«.
void class_destroy(struct class *cls);	Löscht die Klasse »cls«.
struct device *device_create(struct class *cls, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...);	Erzeugt das über »fmt« spezifizierte Verzeichnis, das einen Gerätetyp über die innerhalb des Ordners automatisch angelegten Dateien repräsentiert.
void device_destroy(struct class *cls, struct dev_t *device_nr);	Löscht die zur Gerätenummer gehörenden Einträge im Sys-Filesystem.
void kobject_put(struct kobject *kobj);	Dekrementiert den Referenzzähler des Kernelobjekts. Verwendet keine Instanz das Objekt, wird der zugehörige Speicher frei.
struc dev_t MKDEV(int major, int minor);	Erzeugt aus »major« und »minor« eine Gerätenummer.
int register_chrdev(unsigned int major, const char *driver_name, struct file_operations *fops);	Meldet einen Treiber beim Kernel an. Wird »major« mit 0 vorinitialisiert, wählt der Kernel die nächste freie Majornummer für den Treiber aus (Rückgabewert der Funktion). Wird eine Majornummer vorgegeben (»major > 0«), signalisiert der Rückgabewert »0« ein Okay, ein Wert kleiner null signalisiert Fehler.
int register_chrdev_region(dev_t from, unsigned count, const char *name);	Reserviert einen Bereich von Gerätenummern, die der Kernel unter dem Namen »name« verwaltet.
int unregister_chrdev(unsigned int major, const char *driver_name);	Gibt den Treiber mit der Majornummer »major« und dem Namen »driver_name« frei.
void unregister_chrdev_region(dev_t from, unsigned int count);	Gibt den Gerätenummernbereich beginnend bei »from« frei.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 49

Kern-Technik

Vorgefertigte Codestücke helfen dabei, die Komplexität von Treibercode zu beherrschen. Makros und Muster unterstützen Treiberentwickler bei kritischen Abschnitten und dem Powermanagement. Eva-Katharina Kunst, Jürgen Quade



Open-Source-Programmierer wissen es längst: Vorhandene Software nutzen, um damit neue Software zu gestalten, lohnt sich. Das als Software-Reuse bezeichnete Vorgehen vermeidet Doppelarbeit, Fehler und Unvollständigkeiten. Die Effizienz-Programmierer setzen dabei auf Bibliotheken, Frameworks und Templates. Hinzu kommen die Entwurfsmuster, die eine grundsätzliche, Programmiersprachen- und Umgebungs-unabhängige Beschreibung eines Problems geben.

Das Linux-Magazin und insbesondere die Kern-Technik-Serie versucht Programmierern das Leben zu erleichtern und stellt allgemein gehaltene und vielfältig nutzbare Codeschnipsel vor, die sich in eigene Entwicklungen einbauen lassen. So hat die vorige Folge beispielsweise ein Template behandelt, das den Kernel um eigenen Code erweitert [1].

Diese Kern-Technik ergänzt die Sammlung um weitere Funktionstemplates und Entwurfsmuster. Denn für einen neuen Treiber sind immer wieder bestimmte Funktionen zu implementieren, die einen ähnlichen Aufbau haben (siehe Abbildung 1). Bei den zu implementierenden Routinen handelt es sich um die wesentlichen Zugriffsfunktionen »driver_open()«, »driver_close()«, »driver_read()«, »driver_write()« und »driver_poll()«. Dazu gehören aber auch die Funktionen »template_suspend()« und »template_resume()« für das Powermanagement.

Wenn ein Treiberentwickler die Funktionen schreibt, sollte er sowohl blockierende als auch nicht-blockierende Zugriffe ins Auge fassen. Das führt letztlich zur hohen Komplexität der ansonsten recht einfach strukturierten Lese- und Schreibfunktionen (siehe Abbildung 2).

Geben und nehmen

Die Lese- oder Schreibfunktion eines Treibers aktiviert der Kernel dann, wenn eine Applikation die Systemcalls »read()« oder »write()« für die zugehörige Gerätedatei aufruft. Falls ständig Daten zum Lesen bereitliegen oder der Kernel sie direkt schreiben darf, sind die zu implementierenden Aktionen gradlinig: Beim »read()« transferiert der Treiber die Daten von der Hardware in den Kernelspace und von dort schließlich in den Speicherbereich der Applikation. Der Syscall »write()« macht es umgekehrt: Er transferiert die Daten aus dem Speicher der Applikation in den Kernel und von hier in die Hardware (siehe Abbildung 3).

Ist der Hardwarezugriff jedoch nicht sofort möglich, weil beispielsweise die Hardware noch mit anderen Aufgaben beschäftigt ist, wird es komplizierter. Dann muss der Treiber nämlich die in den Aufrufparametern übergebene Zugriffsart auswerten.

Warten auf die Hardware

Als Default implementiert ein Treiber einen blockierenden Zugriff. Dabei legt er die Applikation und damit die im gleichen Kontext laufende Treiberfunktion so lange (und immer wieder) in den Zustand Schlafen, bis er endlich Daten transferieren kann. Beim nicht-blockierenden Zugriff legt die Treiberfunktion die aufrufende Applikation nicht schlafen, sondern signalisiert ihr mit dem Rückgabewert »-EAGAIN«, dass sie zurzeit nicht lesen oder schreiben darf und sie es später erneut probieren soll.

Der Code, der prüft, ob sich Daten unmittelbar lesen oder geschreiben lassen, stellt einen kritischen Abschnitt dar, den es beispielsweise über Spinlocks zu schützen gilt. Das ist alles andere als trivial und in der Vergangenheit sehr häufig falsch implementiert worden. Daher hat Linus Torvalds das für eine korrekte Implementierung zugrunde liegende Entwurfsmuster in Form eines Makros realisiert.

Das Makro implementiert die in Abbildung 2 sichtbare Schleife. Für den Entwickler heißt das, im Wesentlichen nur noch eine Wait-Queue und den C-Code für eine Bedingung zu implementieren. Anhand der Bedingung muss erkenntlich sein, ob – im Fall des Lesezugriffs – Daten für die Applikation bereitliegen oder nicht. Beim »write()« gibt die Bedingung darüber Auskunft, ob der Treiber Daten schreiben kann. Die Wait-Queue ist ein Objekt, das für das eigentliche Schla-

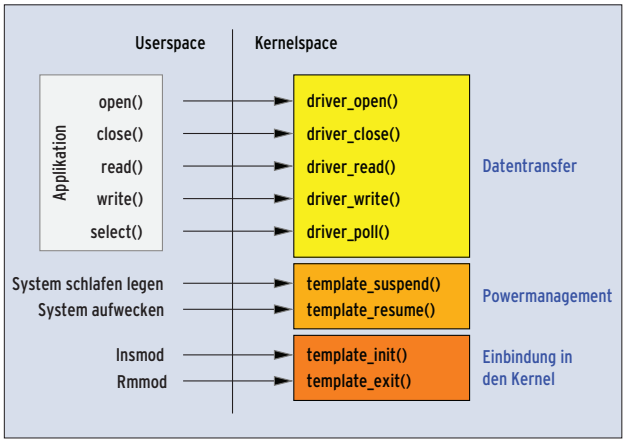


Abbildung 1: Die Grundfunktionen zum Einbinden eines Moduls in den Kern, zum Datenzugriff und zum Powermanagement muss jeder Treiber implementieren.

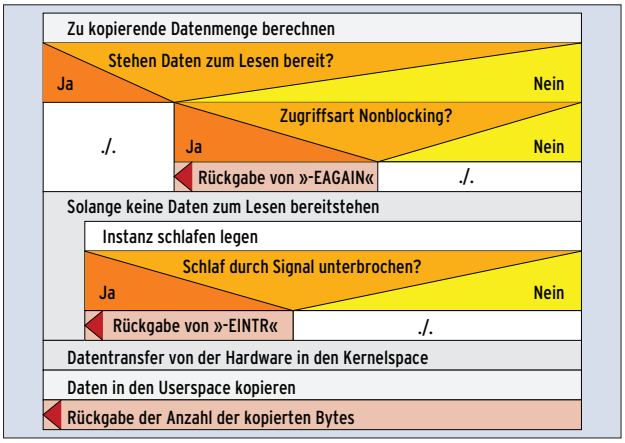


Abbildung 2: Falls Treiber mehrere Zugriffsarten unterstützen, ist die Implementation nicht mehr trivial, da es diverse Fälle zu berücksichtigen gilt.

fenlegen und Aufwecken notwendig ist. Linux initialisiert es beispielsweise beim Laden des Moduls in der Funktion »template_init()« [2].

Seltsame Rückgabewerte

Beim eigentlichen Datentransfer zwischen Kernel- und Userspace gibt es drei Aspekte zu beachten. Erstens: Die Anzahl der real zu transferierenden Bytes muss der Treiber auf Basis des Minimums von angeforderten und zur Verfügung stehenden Daten berechnen. Zweitens: Das Transferieren der Daten kann teilweise oder komplett fehlschlagen; die Funktionen »copy_to_user()« und »copy_from_user()« geben die Anzahl der nicht übertragenen Bytes zurück. Und drittens: Die Funktionen »driver_read()« beziehungsweise »driver_write()« sollten die Anzahl der wirklich übermittelten Bytes zurückgeben.

Das Kopieren kann im Übrigen dann fehlschlagen, wenn die Applikation den Kernel mit falschen Adressen versorgt, beispielsweise mit einem Nullpointer. Die Funktionen »copy_to_user()« und »copy_from_user()« prüfen die Adressen vor dem eigentlichen Transfer und kopieren nur so viele Daten, wie aus ihrer Sicht in Ordnung ist.

Schablonen führen dem Entwickler die Hand

Listing 1 zeigt das Codegerüst für das Entwurfsmuster einer Funktion »driver_read()«. Damit es zum Teil eines funktionierenden Treibers wird, ergänzt der Programmierer noch den Datentransfer zwischen Hardware und Kernel – falls es überhaupt eine Hardware gibt.

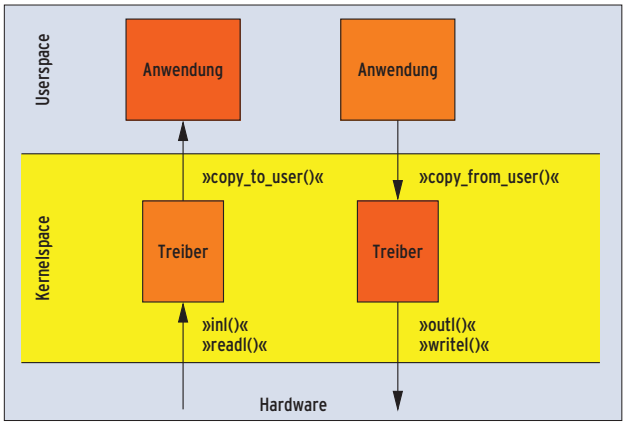


Abbildung 3: Das Grundprinzip jedes Treibers ist ein Datentransport in zwei Zügen: Erst koordiniert er den Datentransfer von der Hardware in den Kernelspace, danach kopiert er sie aus dem Kernel in den Userspace.

Beispielsweise eine Interrupt-Service-Routine initiiert so einen Datentransfer. Kopiert sie die Daten dabei in einen globalen Kernelspeicherbereich, ist dieser durch ein Spinlock zu schützen. Im einfachsten Fall aber liest der Code innerhalb der Funktion »driver_read()« ein Hardwareregister aus.

In beiden Fällen – der Treiber kopiert die Daten außerhalb von »driver_read()« in den globalen Kernelspace oder holt sie innerhalb der Funktion direkt von der Hardware ab – passt der Entwickler die globale Variable »bytes_available« an. Sie reflektiert im vorliegenden Template die Anzahl der Bytes, die eine Applikation zurzeit lesen darf. Stehen neue Bytes zur

Listing 1: Codegerüst für Lesefunktion im Treiber

```
01 static atomic_t bytes_available = ATOMIC_INIT(0);
02
03 #define READ_POSSIBLE (atomic_read(&bytes_available) != 0)
04
05 ssize_t driver_read(struct file *instance, char __user *buffer,
06                     size_t max_bytes_to_read, loff_t *offset)
07 {
08   size_t to_copy, not_copied;
09   char kernelmem[128]; // lokale Variable: Schutz über Spinlocks kann entfallen
10
11   if (!READ_POSSIBLE && (instance->f_flags & O_NONBLOCK)) {
12     // keine Daten vorhanden und Nonblocking-Mode
13     return -EAGAIN;
14   }
15   if (wait_event_interruptible( wq_read, READ_POSSIBLE)) {
16     // während des Schlafens durch ein Signal unterbrochen
17     return -ERESTARTSYS;
18   }
19   to_copy = min((size_t)atomic_read(&bytes_available), max_bytes_to_read);
20   not_copied = copy_to_user(buffer, kernelmem, to_copy);
21   atomic_sub(to_copy - not_copied, &bytes_available);
22   return to_copy - not_copied;
23 }
```

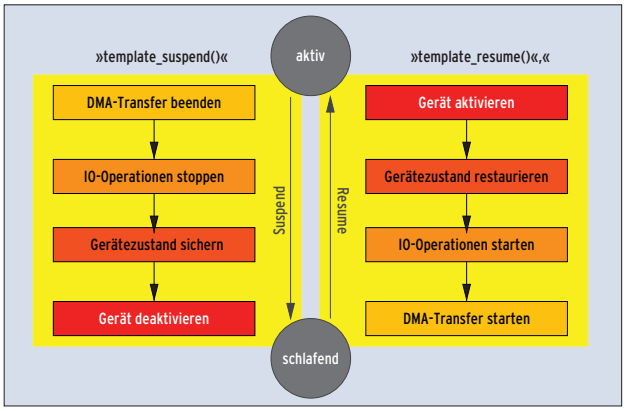
Verfügung, erhöht der Code die Variable entsprechend. Hat er Daten gelesen, erniedrigt er sie um die zugehörige Anzahl Bytes. Da die Variable global ist, könnten mehrere Instanzen sie gleichzeitig modifizieren. Also schützt sie der umsichtige Programmierer durch die Wahl des Datentyps »atomic_t« und die darauf definierten Zugriffsfunktionen [3].

Listing 2 zeigt das Codegerüst für das Entwurfsmuster der Funktion »driver_write()«. Ähnlich wie bei »driver_read()« ist auch hier der Datentransfer zwischen der Hardware und dem Kernelspace zu ergänzen. Die globale Variable »bytes_that_can_be_written« ist vor parallelen Zugriffen zu schützen. Dazu bietet sich wieder der atomare Datentyp an.

Zum Abholen bereit

Jeder Treiber sollte zusätzlich zu den Transferfunktionen auch »poll()« und das funktional weitgehend gleichwertige »select()« anbieten, damit Anwendungen herausfinden können, ob der Treiber für einen Datenaustausch bereit ist. Dazu implementiert der Programmierer eine Funktion »driver_poll()«, mit der er überprüft, ob er Daten lesen oder schreiben darf. Das Ergebnis hiervon hält er in Form einer Bitmaske fest (siehe Listing 3). Typischerweise lässt sich diese Funktion ohne Änderungen in einen neuen Treiber übernehmen.

Für »driver_open()« sowie »driver_close()« hält der Kernel Default-Routinen bereit, die der Programmierer einsetzt, falls er keine Hardware-Initialisierungen vorzunehmen braucht oder keine zusätzlichen, erweiterten Zugriffsrechte prüfen will. Implementiert ein Treiber diese beiden Funktionen nicht selbst, dann greift der Kernel automatisch auf die Defaults zurück.



▶ **Abbildung 4: Zum Schlafenlegen einer Hardwarekomponente führt der Treiber mehrere Operationen durch. Wacht das Gerät wieder auf, durchläuft sie der Kernel in umgekehrter Reihenfolge.**

Neben der Initialisierung von eventuell vorhandener Hardware testen Kernelhacker des Öfteren in der »open()«-Funktion zusätzliche Zugriffsberechtigungen. Eine Anwendung macht das Betriebssystem durch Aufruf von »open()« darauf aufmerksam, dass sie und wie sie auf Daten zugreifen möchte: Zur Auswahl stehen blockierende und nicht blockierende sowie lesende und schreibende Zugriffe.

Bitte einzeln eintreten!

Das Betriebssystem prüft zunächst, ob die zugreifende Instanz entsprechend dem geäußerten Zugriffswunsch dazu die Berechtigung hat. Ist das der Fall, ruft es im zugehörigen Treiber die Funktion »driver_open()« auf. Innerhalb von »driver_open()« hat der Entwickler die Option, die Anzahl der parallel zugreifenden Instanzen zu kontrollieren und etwa zu verhindern, dass Anwender den Treiber zweimal gleichzeitig öffnen, wenn das nicht sinnvoll ist.

Um dieses Verhalten zu realisieren, verwaltet der Treiber eine globale Variable, die die Anzahl der zugreifenden Instanzen zählt. Beim Aufruf von »driver_open()« zählt er die Variable hoch. Hat sie einen Wert von 1, darf die Instanz zugreifen. In jedem anderen Fall verweigert der Treiber den Zugriff. Der Aufruf von »driver_close()« dekrementiert die Variable wieder.

Diese Sequenz ist zunächst unspektakulär ist, aber es gilt, einen kritischen Abschnitt zu schützen. Schließlich können beliebig viele Instanzen gleichzeitig die Funktion »driver_open()« triggern. Zum Schutz eignen sich atomare Variablen wie in »driver_read()«. Aufgrund der Eigenart atomarer Variablen und der zugehörigen Zugriffsfunktionen ist eine Überwachung auf den Wert 0 einfacher: Daher initialisiert der Code die Variable mit -1. Listing 4 implementiert so eine Kontrolle der Anzahl zugreifender Instanzen.

Grundsätzlich sollte ein moderner Gerätetreiber Powermanagement unterstützen und eventuell vorhandene Hardware in den Schlafzustand versetzen bezie-

Listing 2: Codegerüst für Schreibfunktion

```
01 static atomic_t bytes_that_can_be_written = ATOMIC_INIT(0);
02
03 #define WRITE_POSSIBLE (atomic_read(&bytes_that_can_be_written) != 0)
04
05 ssize_t driver_write(struct file *instance, const char __user *buffer,
06                      size_t max_bytes_to_write, loff_t *offset)
07 {
08   size_t to_copy, not_copied;
09   char kernelmem[128]; // Größe anpassen
10
11   if (!WRITE_POSSIBLE && (instance->f_flags & O_NONBLOCK)) {
12     // Hardware oder Treiber sind nicht bereit die Daten zu
13     // schreiben und der Zugriff erfolgt im NON_BLOCKING-Mode
14     return -EAGAIN;
15   }
16   if (wait_event_interruptible(wq_write, WRITE_POSSIBLE)) {
17     // Während des Schlafens durch ein Signal unterbrochen
18     return -ERESTARTSYS;
19   }
20
21   // Rückgabewerte berechnen
22   to_copy = min((size_t)atomic_read(&bytes_that_can_be_written),
23                 max_bytes_to_write);
24   not_copied = copy_from_user(kernelmem, buffer, to_copy);
25
26   // Kernel darf jetzt Daten in die Hardware schreiben
27   atomic_sub(to_copy - not_copied, &bytes_that_can_be_written);
28
29   return to_copy - not_copied;
30 }
```

hungsweise wieder aus diesem Zustand wecken können. Hierfür sind zwei Funktionen zu schreiben: Die Funktion »template_suspend()« ruft der Kern auf, wenn er sich komplett schlafen legt. Sie bekommt im Aufrufparameter vom Typ »pm_message_t« mitgeteilt, ob es sich um Suspend oder um Hibernate handelt. Innerhalb der Funktion kodiert der Entwickler die notwendigen Kommandos, um den Stromverbrauch der durch den Treiber bedienten Hardware auf ein Minimum zu senken.

Einschlafen und aufwachen

Linux ruft beim Wecken die Funktion »template_resume()« auf und reaktiviert schlafende Komponenten. Da das Powermanagement programmtechnisch über das Gerätemodell realisiert ist, binden Linux-Entwickler die beiden Callback-Funktionen in der Initialisierungsfunktion »template_init()« entweder an das Klassen- oder ans Geräteobjekt. Welches der beiden Objekte sie wählen, entscheidet im Wesentlichen über den Zeitpunkt, an dem Linux die Callbacks aufruft: Rückrufe vom Klassenobjekt kommen früher an die Reihe [2]. Listing 5 demonstriert diese Variante.

Da sich der Linux-Quellcode an verschiedene Anforderungen anpassen lässt, benötigt er nicht immer alle Subsysteme. Umsichtige Entwickler rahmen deshalb ihren Powermanagement-Code in ein »#ifdef CONFIG_PM« ein. Solch ein Kernel kommt dann wahlweise auch ohne Powermanagement aus.

Umfangreiches Rahmenwerk

Der Kernel im Allgemeinen und Treiber im Besonderen sind selbst in rudimentären Funktionen ziemlich komplex: Die Codegerüste der vorigen beiden Kern-Techniken umfassen immerhin gut drei Seiten Quellcode. Der Umfang spiegelt auch wider, dass Linux dem Kernelhacker viel Know-how und gute Planung abverlangt, soll produktiver Code entstehen. Aber dank Software-Reuse und mit Hilfe der vorgestellten Codefragmente ist der Einstieg „quick and clean“ möglich. Das Treibergerüst gibt's wie immer zum Download [4]. *(mg)* ■

Listing 3: Test, ob Daten warten

```
01 unsigned int driver_poll(struct file *instance,
02           struct poll_table_struct *event_list)
03 {
04   unsigned int mask = 0;
05
06   poll_wait(instance, &wq_read, event_list);
07   poll_wait(instance, &wq_write, event_list);
08   // Flags zeigen Status an
09   if (READ_POSSIBLE)
10     mask |= POLLIN | POLLRDNORM;
11   if (WRITE_POSSIBLE)
12     mask |= POLLOUT | POLLWRNORM;
13   return mask;
14 }
```

Listing 4: Nur eine Applikationsinstanz erhält Zugriff

```
01 // Lock sperrt nach Increment bei Null:
02 static atomic_t access_count = ATOMIC_INIT(-1);
03
04 static int driver_open(struct inode *devicefile,
05                         struct file *instance)
06 {
07   if (atomic_inc_and_test(&access_count) != 0)
08     // true, falls == 0
09     return 0; // Zugriff erlaubt
10   atomic_dec(&access_count);
11   return -EBUSY;
12 }
13
14 static int
15 driver_release(struct inode *devicefile,
16                struct file *instance)
17 {
18   atomic_dec(&access_count);
19   return 0;
20 }
```

Listing 5: Zwei Funktionen für das Powermanagement

```
01 #ifdef CONFIG_PM // Powermanagement
02
03 static int template_suspend(struct device *dev,
   pm_message_t state)
04 {
05   switch (state.event) {
06   case PM_EVENT_ON:
07     printk("on ...\n"); break;
08   case PM_EVENT_FREEZE:
09     printk("freeze ...\n"); break;
10   case PM_EVENT_SUSPEND:
11     // Stromsparmodus einschalten
12     printk("suspend ...\n"); break;
13   case PM_EVENT_HIBERNATE:
14     // Tiefschlaf einschalten
15     printk("hibernate ...\n"); break;
16   default:
17     printk("pm_event: 0x%x\n", state.event);
18     break;
19   }
20   printk("template_suspend(%p)\n", dev);
21   return 0;
22 }
23
24 static int t emplate_resume(struct device *dev)
25 {
26   // Sparmodus ausschalten, Gerät reaktivieren
27   printk("template_resume(%p)\n", dev);
28   return 0;
29 }
30 #endif
31
32 [...]
33
34 static int __init template_init(void)
35 {
36   init_waitqueue_head(&wq_read);
37   init_waitqueue_head(&wq_write);
38   [...]
39   template_class = class_create(THIS_MODULE,
   TEMPLATE);
40   if (IS_ERR(template_class)) {
41     printk("template: no udev support\n");
42     goto free_cdev;
43   }
44 #ifdef CONFIG_PM
45   template_class->suspend = template_suspend;
46   template_class->resume = template_resume;
47 #endif
48   [...]
```

Infos

[1] Kunst, Quade: „Kern-Technik: Treiberegerüste“, Folge 48, Linux-Magazin 11/09, S. 98
[2] Kunst, Quade: „Kern-Technik: Powermanagement“, Folge 42, Linux-Magazin 11/08, S. 102
[3] Quade, Kunst: „Linux-Treiber entwickeln“: Dpunkt-Verlag, 2. Auflage, Juni 2006
[4] Listing und Makefiles zur Kerntechnik: [http://www.linux-magazin.de/Service/Listings/2010/01/Kern-Technik]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 50

Kern-Technik

Zum Jubiläum: Pinguine, die ins Gefängnis kommen, und Anleitungen, die Programmierer verbrennen sollten: Der Linux-Kernel bietet auch Nicht-Entwicklern allerlei Kurzweiliges. Eva-Katharina Kunst, Jürgen Quade



Was haben ein Mobiltelefon, ein Supercomputer, Fernseher, Netbooks und Armbanduhren, ein Flaschenrücknahme-Automat und Navigationsgeräte sowie der Rechner von Chefredakteur Jan Kleinert gemeinsam? Alle verwenden Linux als Betriebssystemkern. Diese wichtige Komponente liefert problemlos Stoff für jede Menge Magazinartikel. Nach 49 Folgen Technik wird es nun Zeit für eine Zäsur: Diese Kern-Technik befasst sich mit Sonderbarem, Kurzweiligem und Statistischem rund um den Kernel.

Von allem gibt es genug, schließlich ist es ein Projekt der Superlative: Gemäß der Studie „Linux Kernel Development“, herausgegeben im August 2009 von der Linux-Foundation [1], haben 5000 Programmierer aus allen Teilen der Welt rund sechs Millionen Zeilen Code erstellt, 500 Firmen sind beteiligt, darunter mit 20 000 Zeilen auch Microsoft, die Linux ehemals als Krebsgeschwür bezeichneten. Täglich kommen rund 12 000 Zeilen Code neu hinzu.

Netto beträgt der Zuwachs allerdings nur 7000 Zeilen, denn 5000 Zeilen entfernen die Maintainer täglich. 2500 Zeilen erfahren im 24-Stunden-Rhythmus eine Modifikation. Alle zwölf Minuten kommt ein neues Patch hinzu. Das summiert sich auf gut 10 000 Patches für eine neue Kernelversion (siehe Kasten „Abfallprodukt Git“).

Diese Codezeilen sind durchaus ein paar Blicke wert: Der aus Klassen und Objekten realisierte Kern ist ein Zeichen dafür, dass Objektorientierung nicht eine Sache der Programmiersprache ist, sondern primär eine des Kopfes. Die Objekte sind im Linux-Kernel als Datenstrukturen (»struct«) implementiert, Methoden als Funktionspointer.

Code ordentlich notieren

Auch Linux-Interessierte, die nicht täglich zu Editor und Compiler greifen, entdecken bei genauem Hinschauen unterhalb von »/usr/src/linux-*Version*/« viel Interessantes. Selbst mit nur begrenzten Programmierkenntnissen finden sie sich recht gut zurecht, denn gemessen an der Zahl der beteiligten Entwickler herrscht ein erfreulich einheitlicher Stil. Das ist nicht zuletzt der Kodier-Richtlinie von Linus Torvalds zu verdanken, die er im Verzeichnis »Documentation/« unter dem Namen »CodingStyle« ablegte.

In der C-Welt konkurrieren – vereinfacht ausgedrückt – der GNU-Stil mit dem der C-Erfinder Kernighan und Ritchie [2]. Während Klammern bei im GNU-Stil geschriebener Software mittig zum Schlüsselwort stehen, ordnen Kernighan und Ritchie die schließende Klammer der jeweiligen Schlüsselwort-Ebene zu (siehe Listings 1 und 2). Vorteil: Der beim Programmieren sichtbare Bildschirmausschnitt zeigt mehr relevante Codezeilen. Linus Torvalds hat hierzu Stellung genommen: Wer sich für den Linux-Kern interessiert, sollte zumindest den Anfang des Dokuments lesen und seine Anweisungen befolgen (siehe Abbildung 1). Torvalds vielleicht größte Leistung dabei ist das Augenmaß: Trotz seiner Vorgaben toleriert der Linux-Erfinder auch gewisse Abweichungen.

Rekordhalter

Wie es bei einem so umfangreichen Projekt zu erwarten ist, lesen nicht alle Programmierer die Kodier-Richtlinien, geschweige denn, dass sie sie verinnerlichen. Das verdeutlicht beispielsweise die Analyse der Codezeilen-Längen. Linus Torvalds wünscht sich keine Zeilen mit mehr als 80 Zeichen. Erfahrene Programmierer kennen das: Lange Zeilen sind nicht nur am Bildschirm unübersichtlich, insbesondere die gedruckte Version ist unbrauchbar.

Abfallprodukt Git

Torvalds hat nicht nur die Erfolgsstory „Linux-Kernel“ geschrieben. Quasi als Abfallprodukt der Kernelentwicklung ist das Source-Control-System Git berühmt geworden. Der in Oregon lebende Finne hat mit Git nicht nur vorhandene Systeme nachgebaut und in Grundstrukturen oder Details verbessert, sondern hat mit seinen lokalen Repositories eine neue Art der Sourcecode-Verwaltung konzipiert und realisiert. Es vergeht kaum ein Monat, in dem nicht ein bekanntes Projekt auf Git umsteigt, um damit seine Quelltexte zu verwalten.

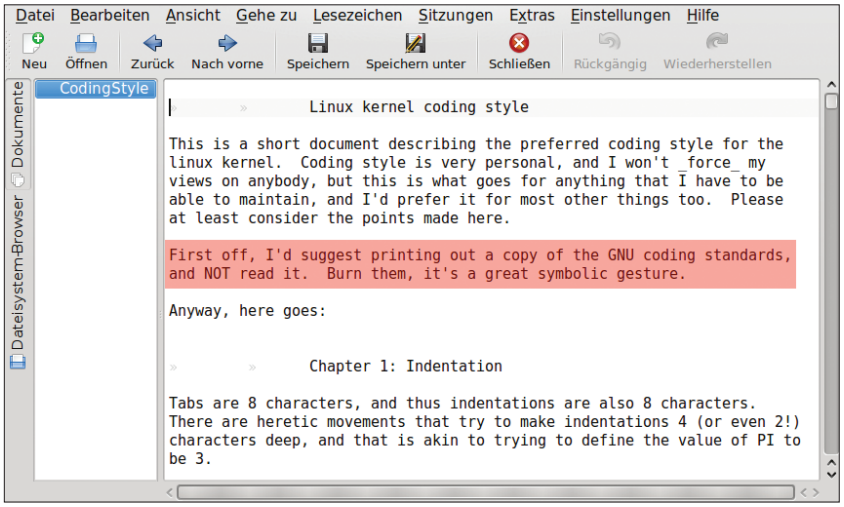


Abbildung 1: Die Anweisung ist eindeutig und die Message klar: Der GNU-Programmierstil verträgt sich nicht mit dem Linux-Kernel. Im weiteren Text schlägt Torvalds jedoch auch versöhnlichere Töne an.

Und im Linux-Kernel? Mit 1992 Zeichen in einer einzigen Zeile hält die Quellcodedatei »APCI1710_Tor.c« einen traurigen Rekord. Und einen weiteren außerdem, nämlich bezüglich der Anzahl der Bedingungen innerhalb einer einzelnen If-Abfrage (siehe Abbildung 2). Was die Entwickler zu diesem Superlativ geritten hat, ist nicht überliefert, doch zur Ehrenrettung von Linus und seinen Mannen: Die Datei befindet sich noch im Staging-Zweig des Kernels.

Ansonsten hat im Linux-Kernel eine Zeile im Durchschnitt eine Länge von 28 Zeichen, obwohl es außerhalb des Staging-Zweiges durchaus einzelne Dateien gibt, etwa die Registerdefinitionen eines Ethernet-Controllers in »arch/blackfin/mach-bf537/include/mach/defBF537.h«, die mit im Schnitt 90 Zeichen pro Zeile deutlich zu lang sind. Bezüglich der Länge einer einzelnen Quellcodedatei haben die Kernelentwickler keine Einschränkungen formuliert. Die längste Datei – betrachtet man die Anzahl der Zeilen – ist mit 17 664 Zeilen die Datei »sound/pci/hda/patch_realtek.c« (siehe Tabelle 1). Sie implementiert den Realtek-HD-Audio-Codec und initialisiert Datenstrukturen zur Konfiguration für verschiedenste Hardwareplattformen, vor allem auf diversen Notebooks.

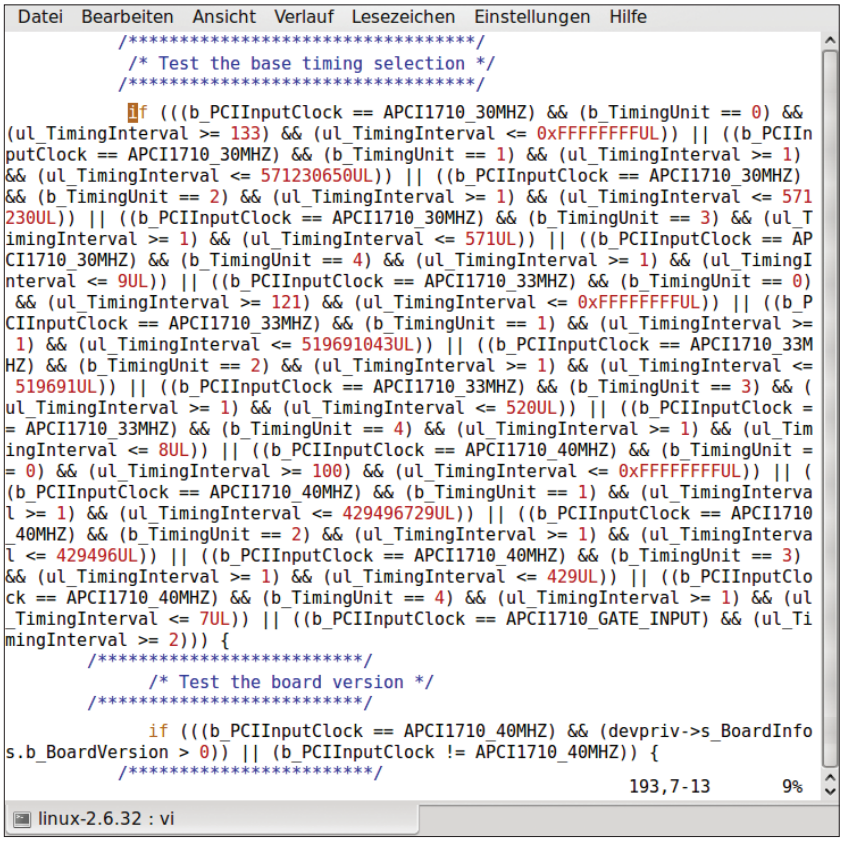


Abbildung 2: Abschreckend: Die längste Codezeile im Kernel 2.6.32 prüft eine Konfigurationseinstellung.

Größenwahn

Ansonsten hält die automatisch generierte Quellcodedatei »fs/nls/nls_cp949.c« mit 875 265 Zeichen den Größenrekord. Sie enthält im Wesentlichen die Kodierung des koreanischen Zeichensatzes. Im Mittel hat jede Datei aber eine Länge von 12 KByte und implementiert in 440 Zeilen rund 17 Routinen. Die zentrale Quellcodedatei »kernel/sched.c« enthält mit 435 die meisten Funktionen (siehe Tabelle 2). Man findet im Linux-Kernel auch quasi leere Dateien oder Routinen, die Dummyfunktionen haben. Wer diejenigen mit echtem Inhalt betrachtet, misst

Listing 1: K&R-Stil

```
01 int f(int x, int y, int z)
02 {
03         if (x < foo(y, z)) {
04                 haha = bar[4] + 5;
05         } else {
06                 while (z) {
07                         haha += foo(z, z);
08                         z--;
09                 }
10                 return ++x + bar();
11         }
12 }
```

Listing 2: GNU-Stil

```
01 int f (int x, int y, int z)
02 {
03     if (x < foo (y, z))
04         haha = bar[4] + 5;
05     else
06       {
07         while (z)
08           {
09             haha += foo (z, z);
10             z--;
11           }
12         return ++x + bar ();
13       }
14 }
```

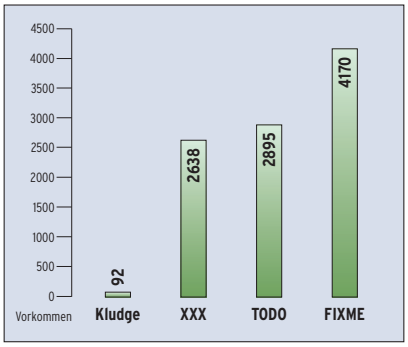


Abbildung 3: Durch Begriffe wie »Kludge«, »XXX«, »TODO« oder »FIXME« markieren Programmierer Code, der verbesserungswürdig ist. Knapp 10 000 solcher Stellen finden sich im aktuellen Kernel.

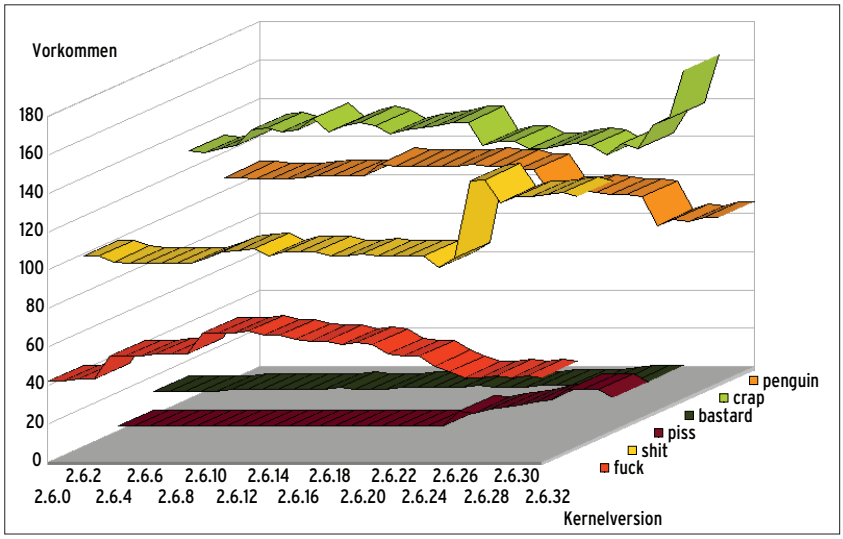


Abbildung 4: Entwickler sind auch vor mitunter etwas wüsten Emotionen nicht gefeit. Kernelentwickler Vidar Holen hat zusammengetragen, wie häufig Verwünschungen in den Kernelversionen vorkommen.

die Datei »lib/reciprocal_div.c« als kürzeste. In neun Zeilen implementiert sie die Kernel-Bibliotheksfunktion »reciprocal_div(k)«. Eine der kürzesten sinnvollen Funktionen realisiert das Schreiben auf die Admins gut bekannte Gerätedatei »/dev/null« (siehe Listing 3).

Gelegenheitsprogrammierer kommen im Code aber auch Unerwartetem auf die Spur. In jedem C-Kurs lernen Neulinge: Benutzt kein Goto, es führt zu unübersichtlichem Spaghetticode und widerspricht dem Paradigma des strukturierten Programmierens. Quellcode-Stöberer hingegen finden im Kernel 2.6.32 mehr als 72 000 Gotos, also mehr als eines auf 100 Codezeilen. Doch auch hier gibt es Entwarnung: Das ist kein Zeichen eines schlechten Programmierstils, sondern eine Hommage an effiziente Code-Generierung. Richtig eingesetzt wird der Quelltext zusätzlich übersichtlicher [3].

Geheimsprache

Bedenklicher sind da eher die „besonderen“ Kommentare. Wenngleich nicht offiziell oder standardisiert, hat es sich unter richtigen Hackern eingebürgert, problematische Codestellen mit den Schlüsselwörtern »XXX«, »FIXME«, »Kludge« oder zumindest als »TODO« zu kennzeichnen. Insbesondere die ersten drei weisen auf einen Hack, auf eine Behelfsmethode, eine temporäre und typischerweise nicht glücklich konzipierte Lösung hin, die einer Überarbeitung bedarf.

What the F**k?

Eigentlich würde man diese Tags nicht in einer Produktivsoftware erwarten. Im Linux-Kernel kommt »Kludge« gut 100-mal, »XXX« 2000-mal, »TODO« 3000-mal und »FIXME« 4000-mal vor (siehe Abbildung 3). Diese finden sich mehr oder minder gleichmäßig über den Quellcode verteilt.

Es gibt aber auch andere Kommentare, die sich nicht nur statistisch auswerten lassen, sondern zudem zeigen, dass auch Programmierer Emotionen haben: Worte wie »shit« (auch in den Abwandlungen »apeshit« und »bullshit«) kommen im Kernel 2.6.32 etwa 130-mal vor, »piss« und »bastard« (ebenfalls mit Abwandlungen) jeweils 17-mal und »crap« 170-mal. Vidar Holen hat die Verwendung dieser Emotionskommentare über die verschiedenen Linux-Versionen hinweg untersucht [4]. Das Ergebnis zeigen die Abbildungen 4 und 5: Auch wenn die absolute Anzahl leicht steigt, die relative Häufigkeit nimmt ab.

Die Emotionslage der Programmierer ist jedoch nicht allein an der statistischen Auswertung einzelner Wörter ablesbar. Oft sind es einzelne Sätze, wie

```
// People celebrate: We love our President!
```

in der TCP-Implementation des Netzwerkcode, die zum Nachdenken oder auch Schmunzeln anregen. Die genaue

Tabelle 1: Die längsten Dateien im Kernel 2.6.32

Zeilenanzahl	Datei
17 664	sound/pci/hda/patch_realtek.c
15 071	drivers/isdn/hardware/eicon/message.c
14 344	drivers/net/tg3.c
13 947	fs/nls/nls_cp949.c
12 826	drivers/scsi/advansys.c
12 279	drivers/net/bnx2x_main.c
12 020	drivers/net/wireless/ipw2x00/ipw2200.c
11 727	drivers/scsi/lpfc/lpfc_sli.c
11 260	drivers/video/sis/init301.c
11 165	drivers/scsi/aic7xxx_old.c

Tabelle 2: Meisten Funktionen pro Datei im Kernel 2.6.32

Anzahl Funktionen	Datei
435	kernel/sched.c
343	drivers/net/niu.c
342	drivers/net/wireless/ipw2x00/ipw2200.c
315	sound/pci/hda/patch_realtek.c
260	drivers/net/tg3.c
258	drivers/net/bnx2x_main.c
250	drivers/platform/x86/thinkpad_acpi.c
232	security/selinux/hooks.c
225	fs/nfs/nfs4xdr.c
223	mm/slub.c

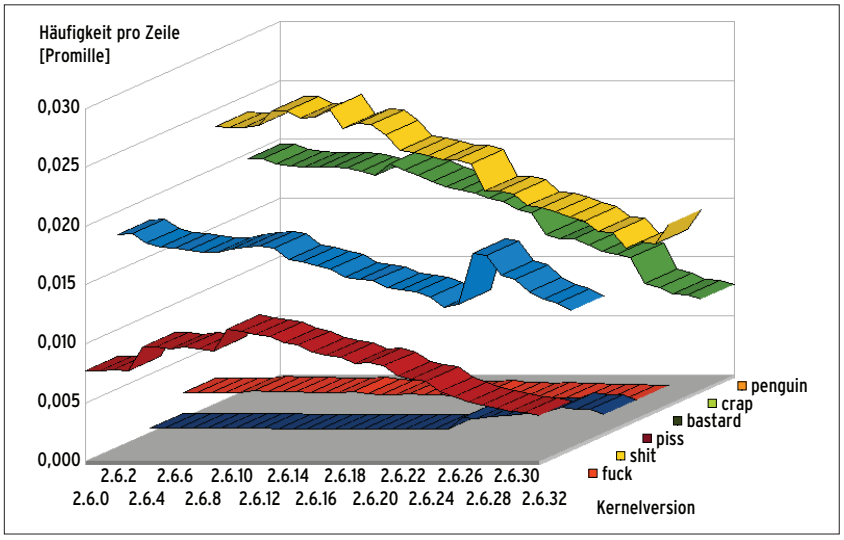


Abbildung 5: Vidar Holen weist aber auch darauf hin, dass das Fluchen im Verhältnis zur Code-Menge weitgehend rückläufig ist. Eine Besserung der Umgangsformen ist also in Sicht (relative Häufigkeit in Promille).

Kernspalterei: Online-Quiz

In 50 Folgen Kern-Technik kam so manches auf den Tisch: Von Interrupt-Service-Routinen über Gerätertreiber bis hin zu Firmware-Unterstützung und Assembler-Integration reicht der Kanon der Themen. Wer prüfen will, ob er in der Linux-Kernel-Mailingliste (LKML) auch einem hitzigen Wortgefecht Torvalds, Cox und Hellwig trotzen kann, nimmt am Online-Quiz des Linux-Magazins teil.

Jeder Teilnehmer, der bis zum 28. Februar 2010 die Quizfragen beantwortet, erhält eine Urkunde. Die besten zeichnen die Kern-Technik-Autoren Kunst und Quade mit einem Ehrendiplom aus. Zum Examen geht es unter [http://linux-magazin.de/kernspalterei/]. Die Kern-Technik-Redaktion wertet alle Antworten aus und erläutert nach Einsendeschluss die korrekten Einsendungen.

Bedeutung des vorstehenden Kommentars bleibt im Übrigen verborgen. Wahrscheinlich ist es eine Art Gedankenstütze, den folgenden Code noch einmal zu überarbeiten. Eine Liste weiterer lesenswerter Kommentare enthält Tabelle 3.

Der Code in »arch/sparc/kernel/smp_64.c« erheitert ebenfalls (siehe Listing 4). In ihm schickt der Programmierer Pinguine ins Gefängnis (Zeile 1118), begnadet sie aber glücklicherweise ein paar Zeilen später wieder (Zeile 1136).

Verkleidete Teufel

Pinguine spielen bei Linus Torvalds und im Linux-Kernel eine tragende Rolle. Typischerweise repräsentieren sie die Prozessoren beziehungsweise Prozessorkerne, weshalb der Code beim Booten – nach entsprechender Kernelkonfiguration – für jeden Prozessorkern einen Pinguin anzeigt. In der Kernelversion 2.6.29 hatte Linus seinen Tux ins Sabbatical geschickt und an seiner Stelle den tasmanischen Teufel Tuz – verkleidet als Tux – auftreten lassen: der Linux-Kernel, ein Werkzeug der Tierschützer (Abbildung 6).

Damit der Linux-Kern sowohl auf einer Armbanduhr als auch auf einem Supercomputer läuft, bringt er zwei besondere Eigenschaften mit: Skalierbarkeit und Portierbarkeit. Skalierbarkeit bedeutet, dass sich der Kernel unterschiedlichen Anforderungen exakt anpasst. Dazu lässt er sich mit Hilfe von etwa 5000 Optionen passgenau auf einen bestimmten Einsatzzweck vorbereiten. Wer etwa kein Netzwerk benötigt, spart sich die Ressourcen und konfiguriert den Kernel ohne eine solche Unterstützung.

Für eine x86-Architektur bedeutet diese Skalierbarkeit einen Unterschied im Umfang des Kernels zwischen 716 KByte und 29 MByte. Der erste Fall entspricht einem mit »make allnoconfig« erzeugten minimalen Kernel, im zweiten Fall schaltet der Entwickler alle möglichen Optionen mit dem Argument »allyesconfig« ein.

Überall zu Hause

Portierbarkeit wiederum bedeutet, dass der Kernel auf verschiedenen Hardwareplattformen lauffähig ist. Dazu gehören nicht nur unterschiedliche Prozessoren (x86, ARM, PowerPC und so

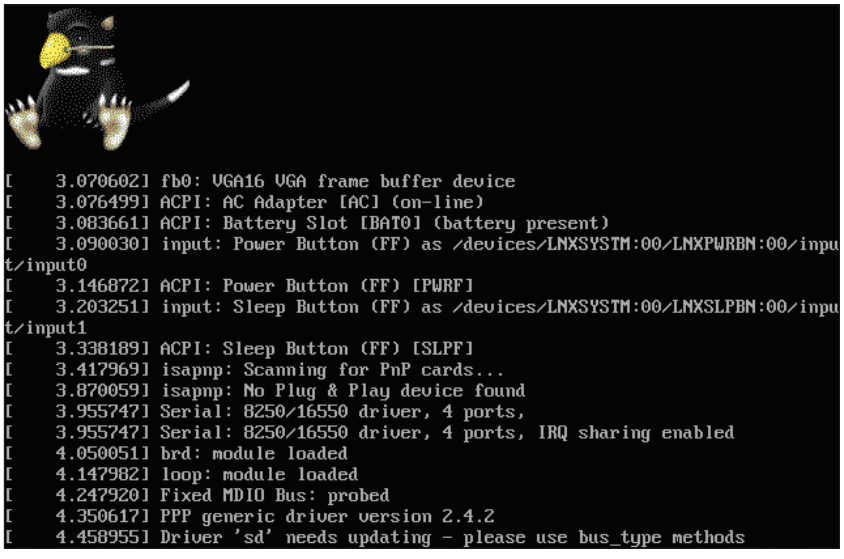


Abbildung 6: Der tasmanische Teufel Tuz ersetzte im Linux-Kernel 2.6.29 den in Urlaub geschickten Tux.

Listing 3: Knappes »/dev/null«

```
01 static ssize_t write_null(struct file * file,
02  const char __user * buf, size_t count, loff_t *ppos)
03 {
04         return count;
05 }
```

Listing 4: „Gehen sie nicht über Los" fordert »smp_64.c«

```
1118 printk("CPU[%d]: Sending penguins to jail...",
1119     smp_processor_id());
[...]
1136 printk("CPU[%d]: Giving pardon to "
1137     "imprisoned penguins\n",
1138     smp_processor_id());
```


comp.os.minix

Diese Gruppe d

LINUX is obsolete

Optionen

Nachrichten 1 - 25 von 73 - Alle ausblenden - Alles übersetzen in die Sprache: Deutsch

Neuere >

Andy Tanenbaum Profil anzeigen ★★★★★ (21 Nutzer) Weitere Optionen 29 Jan. 1992, 14:23

I was in the U.S. for a couple of weeks, so I haven't commented much on LINUX (not that I would have said much had I been around), but for what it is worth, I have a couple of comments now.

As most of you know, for me MINIX is a hobby, something that I do in the evening when I get bored writing books and there are no major wars, revolutions, or senate hearings being televised live on CNN. My real job is a professor and researcher in the area of operating systems.

As a result of my occupation, I think I know a bit about where operating are going in the next decade or so. Two aspects stand out:

1. MICROKERNEL VS MONOLITHIC SYSTEM
Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in 'kernel mode.' This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.

The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel. They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O. Examples of this design are the RC4000, Amoeba, Chorus, Mach, and the not-yet-released Windows/NT.

◀ Abbildung 7: Linux ist überflüssig – hier irrte Betriebssystem-Papst und Minix-Erfinder Andrew S. Tanenbaum. 18 Jahre später behauptet das kaum noch jemand.

weiter), sondern auch unterschiedliche Architekturen, beispielsweise 32- und 64-Bit-Systeme, Ein- und Mehrprozessorsysteme sowie Numa-Architekturen. Für Linux ist das kein wirkliches Problem. Zurzeit unterstützt der Kernel offiziell 20 Prozessortypen, daneben gibt es aber noch mehrere inoffizielle Portierungen.

Irren ist menschlich

Diese Eigenschaften hatte Linux nicht von Anfang an. Ursprünglich war das System nur für Maschinen gedacht, die mit einem x86-kompatiblen Prozessor bestückt sind. Dies und der Umstand, dass Torvalds auf einen monolithischen Systemkern und nicht auf eine Mikrokernel-Architektur gesetzt hat, brachte ihm 1992 die harsche Kritik des Betriebssystem-Professors Andrew S. Tanenbaum ein, der Linux als überflüssig, als „obsolete“ bezeichnete (siehe Abbildung 7). Ein großer Irrtum, wie IT-Experten heute wissen. Die Debatte zwischen Torvalds und Tanenbaum wurde berühmt und gilt heute als Teil der Linux-Folklore [5].

Diplom in Kern-Technik

Linus Torvalds hat sich durch solche Angriffe jedoch nie beirren lassen und so läuft auf Mobiltelefonen, Supercomputern, Fernsehern, Netbooks, Flaschenrücknahme-Automaten, Armbanduhren, Navigationsgeräten und auf dem Rechner von Chefredakteur Jan Kleinert der Linux-Kernel. Wer meint, das alles schon gewusst zu haben, darf sich an dem im Kasten „Kernspalterei: Online-Quiz für Linux-Entwickler“ beschriebenen Quiz versuchen und selbst prüfen, ob er die wichtigsten Inhalte aus 50 Ausgaben Kern-Technik verinnerlicht hat. *(mg)* ■

Infos

[1] Kroah-Hartman, Corbet, McPherson, „Linux Kernel Development“: Linux Foundation, 2009: [http://www.linuxfoundation.org/publications/whowriteslinux.pdf].
[2] Einrückungsstil: [http://de.wikipedia.org/wiki/Einrückungsstil].
[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 3: Linux-Magazin 10/03, S. 81
[4] Vidar Holen, „Linux kernel swear counts“: [http://vidarholen.net/contents/wordcount/]
[5] Andrew S. Tanenbaum, „Linux is obsolete“: [http://www.educ.umu.se/~bjorn/mhonarc-files/obsolete/msg00000.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Tabelle 3: Ausgewählte Quellcode-Kommentare im Linux-Kernel 2.6.32

Datei	Zeile	Kommentar
arch/sparc/include/asm/obio.h	76	/* My Penguins are burning. Are you able to smell it? */
net/ipv4/tcp_input.c	2675	/* People celebrate: „We love our President!“ */
drivers/spi/spi_bfin5xx.c	830	/* oh man, here there be monsters ... and i dont mean the fluffy cute ones from pixar, i mean the kind that'll * eat your data, kick your dog, and love it all. do *not* try */
drivers/scsi/wd33c93.c	65	/* there are lots of lurking bugs and “stupid places“. */
drivers/scsi/qla1280.c	2121	/* Now, I would love the magic decoder ring for this one, the header file provided by QLogic seems to be * bogus or incomplete at best. */
drivers/ide/setup-pci.c	574	/* fixme: mom, mom, they stole me the helper function */
drivers/scsi/qla1280.c	1573	/* I *LOVE* this code! */
arch/m68k/kernel/head.S	1358	/* A friend and I once noted that Apple hardware engineers should be wacked twice each day: once when * they show up at work (as in, Whack!, “This is for the screwy hardware we know you're going to design * today.“), and also at the end of the day (as in, Whack! “I don't know what you designed today, but I'm sure * it wasn't good.“). – rst */
arch/m68k/mac/via.c	50	/* eight address bits. Why can't Apple engineers at least be _consistently_ lazy? - 1999-05-21 (jmt) */
arch/m68k/mac/macints.c	105	/* third layer of indirection. why oh why did the apple engineers do that? */
arch/alpha/kernel/sys_miata.c	250	case LINUX_REBOOT_CMD_RESTART: /* Who said DEC engineers have no sense of humor? ;-) */ if (alpha_using_srm) { *(vuip) PYXIS_RESET = 0x0000dead;
drivers/atm/horizon.c	1992	// they will always be zero for the SAMBA. Ha! Bloody hardware engineers. It'll never work.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 51

Kern-Technik

Zum Schutz kritischer Abschnitte haben Mutexe im Linux-Kernel Semaphore verdrängt. Hauptgrund ist die bessere Performance. Das patentierte Lock-Debugging dagegen ist ein Abfallprodukt. Eva-Katharina Kunst, Jürgen Quade



Kritische Abschnitte zu schützen gehört zu den Herausforderungen der Programmierung moderner Systeme. Mehrere Methoden, beispielsweise atomare Integer- und Bitvariablen, Semaphore, Spinlocks, Sequence-Locks, Memory-Barrier [1], RT-Mutexe oder auch Futexe [2], waren bereits Gegenstand der Kern-Technik. Nur Mutexe, die im Linux-Kernel seit einiger Zeit die klassischen Semaphore ablösen, blieben außen vor.

Das scheint zunächst nicht tragisch, handelt es sich bei einem Mutex doch ohnehin um eine Sonderform des Semaphors. Bei ihnen dürfen konfigurierbar viele Instanzen gleichzeitig einen kritischen Abschnitt betreten. Das Mutex – begrifflich von Mutual Exclusion (gegenseitiger Ausschluss) abstammend – fixiert die Instanzenanzahl auf eine. Ein kritischer Abschnitt ist eine Codesequenz, in der die CPU auf gemeinsam genutzte Betriebsmittel zugreift. Arbeiten mehrere Instanzen parallel in diesem Code, kommt es zu unvorhersehbaren Ergebnissen. Semaphore und Mutexe verhindern das, indem sie den Eintritt sequenzialisieren.

Die Begrenzung auf eine Instanz ermöglicht im Vergleich zum universellen Semaphor eine effizientere Implementierung sowohl bezüglich der Laufzeit als auch des Speicherverbrauchs [3]. Und da Entwickler Semaphore ohnehin vorwiegend als Mutexe einsetzen, spricht vieles für deren Verwendung.

Entweder – oder

Jedoch gibt es bei den Kernel-Mutexen im Vergleich zum Semaphor weitere Einschränkungen: Während der Kern Semaphore unabhängig vom so genannten Eigentümer einer Instanz hält und andere Instanzen sie freigeben dürfen, ist dies mit den Linux-Mutexen nicht zu machen. Hier gilt das Eigentümerprinzip: Die Instanz, die ein Mutex erfolgreich reserviert hat, gibt es auch frei. Ein Consumer-Producer-System lässt sich damit also nicht aufbauen. Das stellt aber kein Problem dar, denn Semaphore stehen ja ohnehin weiter zur Verfügung.

Erfreulicherweise ist der Übergang vom klassischen Semaphor oder auch Spinlock zum effizienten Mutex schmerzlos. In den meisten Fällen ersetzt ein Systemaufruf einen anderen. Konkret gilt das für die Definition und Initialisierung des Mutex, für das Halten und Freigeben und für das Prüfen, ob eine Instanz ein Lock hält oder nicht.

Mutexe richtig reservieren

Für die Definition und Initialisierung gibt es statische und dynamische Methoden. Den Torwächter spielt in beiden Fällen eine Variable vom Typ »struct mutex«. Die statische Methode definiert die Mutexvariable über das Makro »DEFINE_MUTEX()«. Das Beispielmodul von Listing 1 zeigt demgegenüber mit Zeile 3 in Kombination mit Zeile 9 die dynamische Variante.

Um das Mutex zu reservieren, ruft der Programmierer im einfachsten Fall die

Mutex versus Spinlock

Zwei Methoden regeln den Zugang zu Codestücken, die exklusiven Zugriff auf Ressourcen benötigen: Ein Mutex legt eine Instanz schlafen, während das Spinlock aktives Warten realisiert – das so genannte Spinning.

Bei einem kurzen kritischen Abschnitt ist aktives Warten effizienter, da kein Kontextwechsel notwendig ist. Außerdem lässt es sich auch im Kontext von Hard-IRQs und Soft-IRQs einsetzen. Dort schützt es kritische Abschnitte, die sich beispielsweise eine Treiberfunktion mit einer Interrupt-Service-Routine teilt. Das Mutex spielt seinen Vorteil demgegenüber bei längeren zu schützenden Codesequenzen aus. Durch das Schlafenlegen der Instanz können andere rechenbereite Prozesse die vorhandene CPU-Zeit nutzen. Außerdem ist das Verfahren auch auf Singlecores einsetzbar.

Adaptives Spinning nutzt Multicores

Zudem haben die Kernelprogrammierer dem Linux-Mutex ein so genanntes adaptives Spinning beigebracht und es damit zum Zwitter gemacht. Wird auf einer Mehrkernmaschine ein zu reservierendes Mutex von einer Instanz auf einer anderen CPU gehalten, entscheidet sich der Kernel für das aktive Warten (Spinning) und führt keinen Kontextwechsel durch.

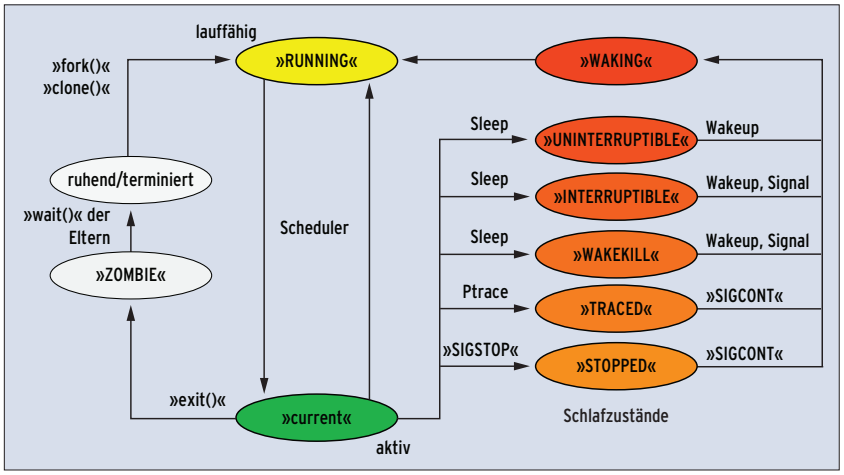


Abbildung 1: Im Zustand »TASK_WAKEKILL« weckt der Kernel seit 2.6.25 schlafende Prozesse auch dann auf, wenn sie das tödliche Signal »SIGKILL« empfangen.

Funktion »mutex_lock()« auf, »mutex_unlock()« gibt es wieder frei:

```
mutex_lock(&mutex_dynamisch);
    /* ... Kritischer Abschnitt */
mutex_unlock(&mutex_dynamisch);
```

Besser reserviert der Entwickler jedoch nach der dynamischen Methode mit den Funktionen »mutex_lock_killable()« oder »mutex_lock_interruptible()«. Diese Funktionen sind zwar komplizierter anzuwenden, weil ihr Rückgabewert auszuwerten ist:

```
if (mutex_lock_killable(&mutex_statisch)
      == -EINTR) {
    return -EIO;
}
    /* ... Kritischer Abschnitt */
mutex_unlock(&mutex_dynamisch);
```

Doch ermöglicht es diese Variante, eine auf das Mutex wartende Instanz per Signal wieder aufzuwecken.

Die Funktionen mit dem Namenszusatz »_killable« sind übrigens vergleichsweise neu im Linux-Kernel, bisher kaum genutzt und wenig bekannt. Das möchten die Kernelhacker unbedingt ändern. Die Funktionen »mutex_lock_interruptible()« und »mutex_lock_killable()« haben gemeinsam, eine schlafende Treiberinstanz auch dann aufzuwecken, wenn sie ein Signal erhält.

Im Schlaf töten

Während die unterbrechbare Variante aber auf jedes Signal reagiert, muss es bei der Killable-Variante schon ein Signal sein, das den Prozess sicher beendet. Das Signal »SIGKILL« bricht den Prozess in jedem Fall ab, bei den übrigen Signalen hängt es aber davon ab, welche Signale die Instanz abfängt. Sie unterbrechen den mit »mutex_lock_killable()« eingeleiteten Schlaf nicht.

Kernel 2.6.25 führte dazu den Prozesszustand »TASK_WAKEKILL« ein, um Applikationsprogrammieren entgegenzukommen (siehe Abbildung 1). Die meisten Anwendungen werten erfahrungsgemäß die Rückgabewerte von Systemaufrufen wie beispielsweise »read()« und »write()« nicht hinreichend aus. Amateurprogrammierer gehen oft fälschlicherweise davon aus, dass Signale die Funktionen nicht abbeziehungsweise unterbrechen. Daraus zogen die Kernelhacker die Konsequenz, die nicht unterbrechbaren Varianten den an sich geschickteren unterbrechbaren vorzuziehen: Lieber ein paar ewig schlafende Jobs in der Prozesstabelle als undefiniert arbeitende Applikationen. Die neuen »_killable«-Funktionen lösen diese Gewissenskonflikte [4].

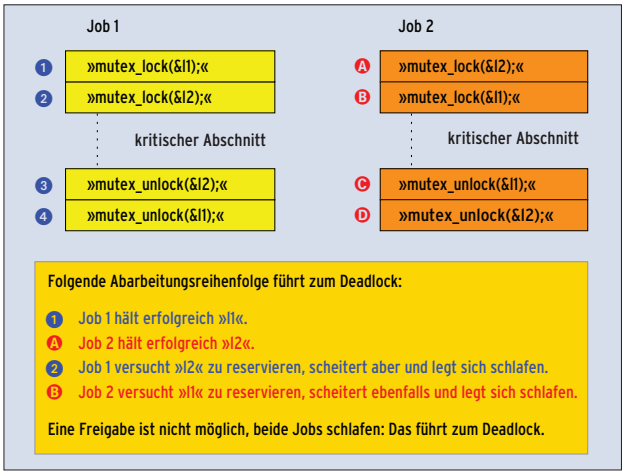
Außerdem gibt es für Mutexe die von Semaphoren und Spinlocks abgeleitete Funktion »mutex_trylock()«. Sie verhindert das Schlafenlegen der aufrufenden Instanz, wenn eine andere Instanz das Mutex bereits hält. In diesem Fall liefert die Funktion den Wert 0 zurück. War die Reservierungsanfrage hingegen erfolgreich, erhält der Aufrufer eine 1 als Rückgabe. Das entspricht dem Verhalten von »spin_lock_trylock()«. Umgekehrte Logik bei »down_trylock()«: Die Funktion gibt in der gleichen Situation 0 zurück. Wer nur den Zustand, ob frei oder gehalten, des Mutex prüfen will, wählt die Funktion »mutex_is_locked()«.

Atomare Variablen

Die Funktion »atomic_dec_and_mutex_lock()« korrespondiert mit der von Spinlocks bekannten Funktion »atomic_dec_and_lock()«. Sie dekrementiert die übergebene atomare Variable und schnappt sich das Lock in einem Rutsch. Falls das Ergebnis 0 ist, sperrt der Kern das Mutex per »mutex_lock()«, reserviert es so und zeigt dies durch den Rückgabewert 1 an. Ist die Variable jedoch ungleich 0, gibt

Listing 1: Bewusst fehlerhaftes Modul »mutextest.c«

```
01 #include <linux/module.h>
02
03 static struct mutex kern_technik;
04
05 static int __init mod_init(void)
06 {
07   /* Mutex-Debugging reaktivieren */
08   debug_locks = 1;
09   mutex_init(&kern_technik);
10   if (mutex_lock_killable(&kern_technik)
11             == -EINTR) {
12     pr_info("Durch Signal unterbrochen!\n");
13     return -EIO;
14   }
15   // FEHLER: Mutexe dürfen nicht zweimal
16   // (rekursiv) reserviert werden
17   if (mutex_lock_interruptible(&kern_technik)
18             == -EINTR) {
19     pr_info("Durch Signal unterbrochen!\n");
20     mutex_unlock(&kern_technik);
21     return -EIO;
22   }
23   // Hier steht eigentlich der Code des per
24   // Mutex geschützten kritischen Abschnitts.
25   mutex_unlock(&kern_technik);
26   return -EIO;
27 }
28
29 static void __exit mod_exit(void)
30 {
31   ;
32 }
33
34 module_init(mod_init);
35 module_exit(mod_exit);
36 MODULE_LICENSE("GPL");
```



◀ Abbildung 2: Im Zusammenspiel mit mehreren Locks kommt es schnell zu Verklemmungen.

die Routine 0 zurück und versucht erst gar nicht das Mutex zu halten.

Trotz der scheinbar einfachen Semantik der Mutex-Funktionen ist ihr Einsatz alles andere als trivial. Ein Fehler wie etwa die falsche Reihenfolge beim Reservieren führt leicht zu einem sporadisch auftretenden Deadlock, wie das Beispiel in Abbildung 2 zeigt. Hinzu kommen noch generell einzuhaltende Randbedingungen beim Einsatz (siehe Kasten „Verwendungsrichtlinien für Mutexe“).

Beinahe alle Einschränkungen kann der Torvalds'sche Betriebssystemkern seit einiger Zeit überprüfen: das Eigentümerprinzip, das Verbot des rekursiven Reservierens, die Gebote, nur über vorgegebene Funktionen zu initialisieren, den Job aufrechtzuerhalten, so lange eine Instanz das Mutex noch hält, und währenddessen seinen Speicher nicht freizugeben sowie den Bann innerhalb von Interruptkontexten.

Ingo Molnar hat dem Kernel mit seinem Lockdep-Validator das dazu notwendige Rüstzeug mitgegeben, das zugleich den korrekten Einsatz von Semaphoren und Spinlocks überwacht. Da die Überwachung jedoch mit Speicherplatz- und Performance-Einbußen verbunden ist, bleibt der Validator standardmäßig deaktiviert.

Bevor ein Entwickler diese Funktionalität aktivieren darf, steht ihm jedoch ein Neubau des Kernels mit der Konfigurationsoption »DEBUG_MUTEXES« ins Haus. Der Kasten „Kernel 2.6.33 unter Ubuntu 9.10 selbst übersetzen“ weist den Weg, da sich die Tools zur Kernelgenerierung zurzeit nicht mit dem Quellcode des aktuellen Kernels vertragen: Am Ende des oft länglichen Kompiliervorgangs beschwert sich ein Werkzeug über eine fehlerhaft erzeugte Headerdatei. Mit einem Patch – wie in Abbildung 3 zu sehen – gelingt das Übersetzen trotzdem. Aber vielleicht ist das bereits Geschichte, wenn der Artikel erscheint.

Zu viele Fehlermeldungen

Ein zweites Patch benötigt der Kernel selbst. Sobald die Überwachungsmimik einen möglichen Fehler in der Logik der Sperren erkennt, reicht er dazu Debug-Informationen an den Syslog-Daemon weiter. Danach schaltet das Subsystem aber jedes weitere Mutex-Debugging ab. Zu leicht würde der Entwickler ansonsten mit einer Kaskade von Fehlermeldungen überhäuft.

Molnars System meldete den platzierten Fehler im Testmodul zunächst nicht: Stattdessen schlug auf dem Testsystem der Autoren bereits beim Booten eines ansonsten unveränderten Kernels die Überwachung mit der Meldung »Possible circular locking dependency detected« im Modul »pulseaudio« an – ein mögliches Problem, das an anderer Stelle zu untersuchen ist.

Es gibt keine Option, ein deaktiviertes Mutex-Debugging wieder einzuschalten. Das Deaktiveren des Mutex-Debuggers steuert die Kernelvariable »debug_locks«, die in der Datei »/usr/src/linux-2.6.33/lib/debug_locks.c« definiert ist. Hat sie den Wert 1, meldet der Kernel ein potenzielles Lockproblem, bei 0 schweigt er. Um das Debug-Subsystem zu kontrollieren, benötigen Entwickler also Zugriff auf die Variable und fügen die Zeile

```
EXPORT_SYMBOL_GPL(debug_locks);
```

hinter deren Definition ein. Alternativ geht das auch am Ende der Quelldatei »debug_locks.c«.

Listing 1 demonstriert nicht nur den Einsatz von Mutexen in eigenem Kernelcode, sondern auch das Mutex-Debugging. Sobald der Entwickler das Modul mit Hilfe des Makefiles aus Listing 2 und »make« übersetzt hat, lädt er es anschließend

Verwendungsrichtlinien für Mutexe

Die Quellcodedatei »linux/mutex.h« listet eine Reihe von Einschränkungen beim Einsatz eines Mutex auf. Das beginnt bei der Initialisierung, schließt das mehrfache Halten ein und endet bei Vorgaben zum Freigeben der Datenstruktur. So dürfen Programmierer nur über die erwähnten Funktionen beziehungsweise Makros Mutexe initialisieren. Das bedeutet konkret, dass es nicht erlaubt ist, den Speicherbereich eines Mutex-Objekts einfach zu kopieren, um auf diese Weise ein zweites Mutex zu erhalten. Ebenfalls verboten ist die mehrfache Initialisierung eines Linux-Mutex, insbesondere ist dies nicht erlaubt, wenn das Mutex bereits von einer Instanz gehalten wird.

Da ein Mutex die aufrufende Instanz unter Umständen schlafen legt, darf der Programmierer die Funktion nur im Prozess- oder im Kernelkontext aufrufen. Die Verwendung innerhalb einer Interrupt-Service-Routine, eines Soft-IRQ, eines Tasklet oder eines Timers ist also ausgeschlossen. Normale Treiberfunktionen wie »driver_ope n()«, »driver_read()«, »driver_write()« oder »driver_close()« können Mutexe aber problemlos verwenden. Ähnlich steht es um Kernelthreads oder um den Code von Systemcalls. Das Eigentümerprinzip besagt, dass nur die Instanz, die das Mutex reserviert, es auch wieder freigeben darf.

Zu ergänzen ist noch, dass Entwickler Mutexe weder mehrfach reservieren (rekursives Locken) noch mehrfach freigeben dürfen. Und schließlich darf eine Instanz, die ein Mutex hält – beispielsweise ein Kernelthread –, sich nicht beenden, ohne das Mutex vorher freizugeben. Entwickler dürfen den Speicher, in dem sie das Mutex-Objekt ablegen, nicht freigeben, so lange ihr Code das Mutex noch hält.

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m    := mutextest.o
03
04 else
05 KDIR     := /lib/modules/$(shell uname -r)/build
06 PWD      := $(shell pwd)
07
08 default:
09         $(MAKE) -C $(KDIR) M=$(PWD) modules
10 endif
```

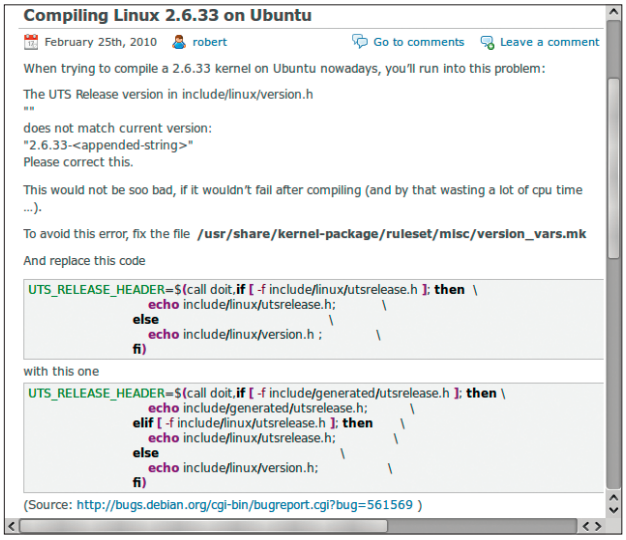


Abbildung 3: Auf einem nicht ganz aktuellen Ubuntu 9.10 müssen Entwickler vor dem Übersetzen noch die Datei »/usr/share/kernel-package/ruleset/misc/version_vars.mk« patchen, damit die Skripte eine Headerdatei korrekt generieren.

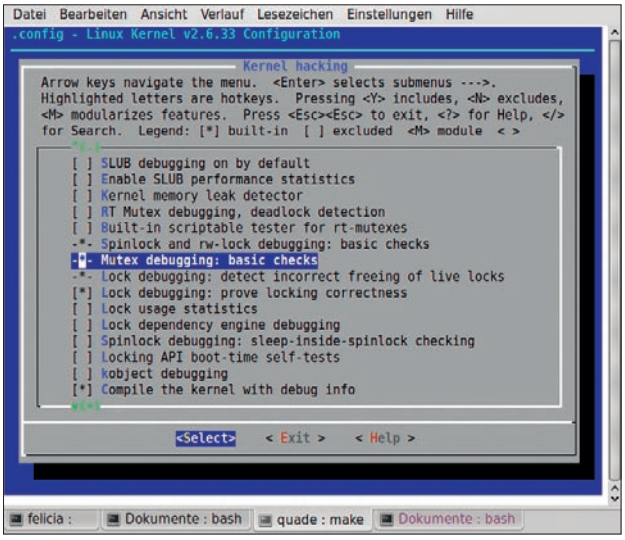


Abbildung 4: Wer im Kernel die Option »DEBUG_MUTEXES« einschaltet, ist in der Lage, eine fehlerhafte Verwendung der Synchronisationselemente aufzuspüren. Die Voreinstellung deaktiviert den Code, da er sich primär zum Debuggen eignet.

mittels »insmod mutextest.ko«. In einem zweiten Terminal lassen sich per »tail -f /var/log/kern.log« die Ausgaben des Syslog-Daemon verfolgen.

Dosierter Regelverstoß

Um das Beispiel in Listing 1 nicht unnötig aufzublasen, reserviert der Code das Mutex direkt in der Funktion »mod_init()«, die der Kernel beim Laden des Moduls per »insmod« aufruft. Um den Mutex-Debugger zu triggern, reserviert das Modul das Mutex fehlerhafterweise zweimal, nämlich in den Zeilen 9 und 17. Das verstößt gegen die Verwendungsrichtlinien. Der durch »debug_locks = 1;« wieder aktivierte Lockdep-Validator erkennt das Problem und gibt es im Syslog aus (Abbildung 5). Er meldet nicht nur die genaue Ursache mit »possible recursive locking detected«, sondern spannenderweise auch den im Programmcode verwendeten Namen des Lock, in diesem Fall »kern_technik«.

Weniger für Mutexe, sehr wohl aber für Spinlocks ist der in geschweiften Klammern stehende Geheimcode interessant, der dem Mutex-Namen in der Ausgabe folgt: »{ . . . }«. Eine Hilfe zur Auflösung dieses Rätsels findet sich in der Datei »/usr/src/linux-2.6.33/Documentation/lockdep-design.txt« und in Abbildung 6.

Grundsätzlich registriert der Lock-Validator, in welchem Kontext (Hard-IRQ, Soft-IRQ, Reclaim-FS) ein Kernelthread, eine ISR, ein Tasklet oder ein Timer die jeweilige Sperre hält. Mit Hard-IRQ bezeichnen die Kernelentwickler eine Interrupt-Umgebung, in der Interrupts typischerweise gesperrt sind. Das betrifft hauptsächlich Interrupt-Service-Routinen. Sie sind ihrerseits in der Lage, Code wie Tasklets oder Timer im Soft-IRQ-Kontext zu unterbrechen. Die dritte Umgebung schließlich kennzeichnet, dass sie Zugriffe auf Dateisysteme zulässt.

Der Validator differenziert darüber hinaus zwischen normalen Locks und Lese-Schreib-Locks. Letztere gewähren nur

Kernel 2.6.33 unter Ubuntu 9.10 selbst übersetzen

Der Standardkernel eines Ubuntu-Systems unterstützt das Debuggen von Locks (Mutex, Spinlocks, Semaphore) nicht im Auslieferungszustand. Entwickler müssen sich ihren eigenen Kernel bauen. Das aber ist – wie im Artikel ausgeführt – mit dem noch aktuellen Ubuntu 9.10 und dem aktuellen Kernel 2.6.33 nicht ohne Weiteres möglich. Bis die Distribution ihre Tools anpasst, hilft folgendes Rezept:

- Pakete »kernel-package«, »build-essential« und »libncurses5-dev« zum Generieren von Kernelpaketen installieren.
- Pakete patchen: Abbildung 3 zeigt, wie Entwickler in der Datei »/usr/share/kernel-package/ruleset/misc/version_vars.mk« ein für die Generierung von Kernel 2.6.33 ungeeignetes Codestück gegen die funktionierende Variante austauschen [5].
- Quellcode runterladen: »wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.tar.bz2«.
- Quellcode auspacken: »tar xjvf linux-2.6.33.tar.bz2 -C /usr/src«.
- Kernel konfigurieren:

```
cd /usr/src/linux-2.6.33
cp /boot/config-$(uname -r) ➥
/usr/src/linux-2.6.33/.config
make menuconfig
```

Die Option »DEBUG_MUTEXES« im Menüpunkt »Mutex debugging: basic checks« – wie in Abbildung 4 erkennbar – einschalten.

- Kernel patchen: Für die im Artikel beschriebenen Tests ist es notwendig, die Variable »debug_locks« für den Modulzugriff freizugeben. Dazu fügt der Entwickler die Zeile »EXPORT_SYMBOL_GPL(debug_locks);« ans Ende der Datei »/usr/src/linux-2.6.33/lib/debug_locks.c« an.
- Image- und Headerpakete kompilieren und Reste vorheriger Kernel-Bauten aufräumen:

```
make-kpkg clean
CONCURRENCY_LEVEL=4 ➥
make-kpkg --initrd --rootcmd=fakeroot ➥
--revision=20100228 kernel_image ➥
kernel-headers
```

- Die entstandenen Image- und Headerpakete »linux-image-2.6.33_20100228_amd64.deb« und »linux-headers-2.6.33_20100228_amd64.deb« mit »dpkg« installieren (hier für das 64-Bit-System).
- Den neuen Kernel booten.

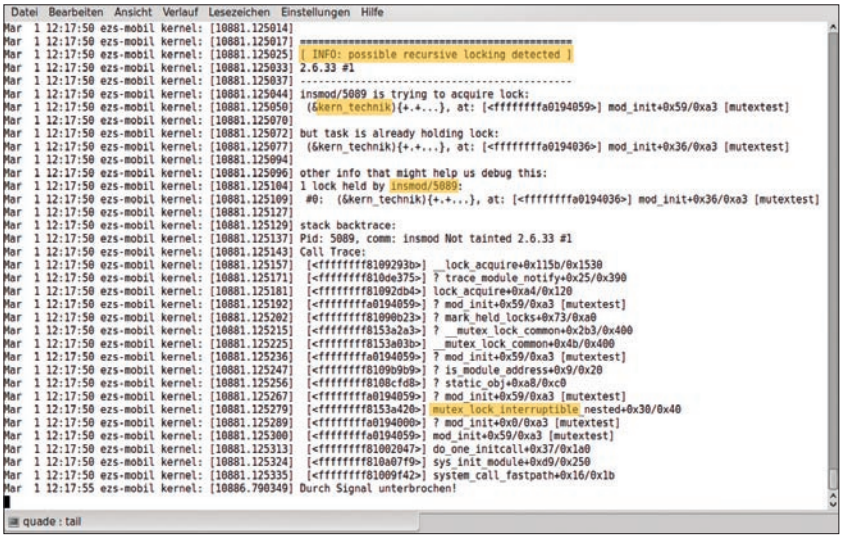


Abbildung 5: Die Ausgabe des Lockdep-Validators zeigt im Klartext den vermeintlichen Fehler.

jenen Prozessen exklusiven Zugriff auf den kritischen Abschnitt, die dort Daten verändern wollen. Die hier erwähnten Mutexe gibt es allerdings nur als normale Locks. Jede der sechs Varianten kann einen von vier Zuständen annehmen. Abbildung 6 verdeutlicht das Prinzip. Im Beispiel halten sowohl ein normaler Hard-IRQ als auch ein normaler Soft-IRQ das Lock »kern_technik« bei freigegebenen Interrupts. Für die Fehlersuche ist das hilfreich: Der Programmierer kann daraus schließen, ob er eine Sperre in einem unerlaubten Kontext reserviert hat oder nicht.

Der Lockdep-Validator organisiert Locks in Klassen, wobei jeweils gleichartige Locks eine Klasse bilden. Gleichartig bedeutet, das Lock sichert eine bestimmte Datenstruktur, beispielsweise »struct inode«. Der Lockdep-Validator registriert, in welchem Kontext und von welcher Instanz ein Lock reserviert wird. Reservierte und nicht freigegebene Locks protokolliert er in durchaus umfangreichen Tabellen inklusive der Zeitstempel mit.

Lockdep findet Deadlocks

Der Lockdep-Validator hängt sich in alle fürs Locking relevanten Systemcalls (zum Beispiel »exit()«) beziehungsweise Kernelfunktionen (etwa »mutex_lock()«) ein. Er hat eine Reihe von Regeln eingebaut, mit denen er das Reservieren und Freigeben von Locks (Spinlocks, Semaphore, Mutexe) während des normalen Betriebs überwacht. Beim Aufruf von »exit()« zum Beispiel sieht der Validator in den Tabellen nach, ob der zugehörige Job noch ein Lock hält. Wenn ja, kommt es zu einer Meldung im Syslog und zum Abschalten des Überwachungscode.

Weitere Regeln identifizieren beispielsweise mehrfaches Reservieren, mehrfaches Freigeben, Reservieren und Freigeben in unerlaubtem Kontext, verschachteltes Halten von Locks oder sogar das Erkennen von Deadlock-Situationen. Der Validator funktioniert sowohl auf Ein- als auch auf Mehrprozessormaschinen. Schade, dass das Subsystem so viel Rechenzeit verschlingt [6].

Der Lock-Validator kann also weitaus mehr als nur rekursiv gehaltene Mutexe melden [7]. Dank seiner ausgeklügelten Mechanismen hat er dabei geholfen, viele Probleme im Kernel zu identifizieren und zu beseitigen. Ende 2008 beantragte sein Entwickler Ingo Molnar beim amerikanischen Patentamt USPTO die Methode zu patentieren [8]. *(mg)* ■

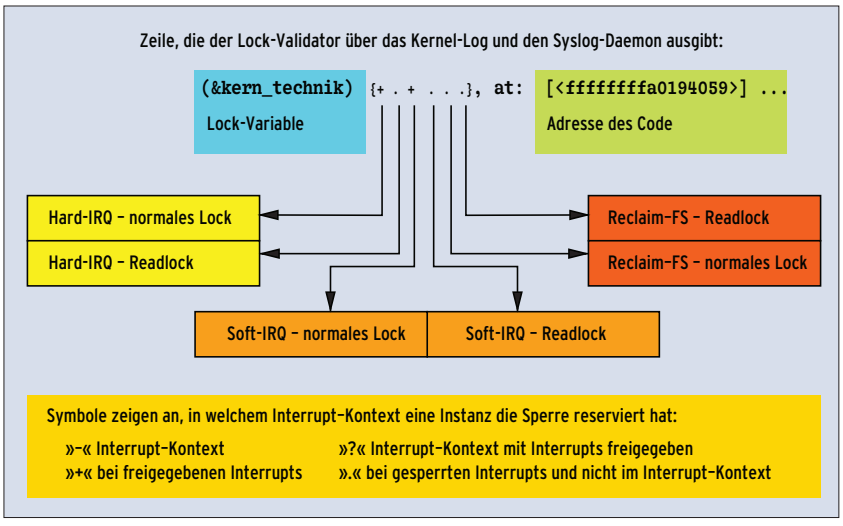


Abbildung 6: Die seltsame Zeichenfolge im Syslog entpuppt sich mit etwas Nachhilfe tatsächlich als sinnvolle Information: Jedes Symbol steht für einen eigenen Lock-Kontext.

Infos

[1] Jürgen Quade, Eva-Katharina Kunst, „Schutz kritischer Abschnitte“: Kern-Technik 5, Linux-Magazin 12/03, S. 82

[2] Jürgen Quade, Eva-Katharina Kunst, „Realtime-Mutexe“: Kern-Technik 44, Linux-Magazin 03/09, S. 88

[3] Ingo Molnar et al., „Generic Mutex Subsystem“, Kerneldokumentation zu Sperren: »Documentation/mutex-design.txt«

[4] Jonathan Corbet, „TASK_KILLABLE“: [http://lwn.net/Articles/288056/]

[5] Nerdy Room, „Compiling Linux 2.6.33 on Ubuntu“: [http://www.nrtm.de/index.php/2010/02/25/compiling-linux-2-6-33-on-ubuntu/]

[6] Ingo Molnar, „Runtime locking correctness validator“: [http://www.mjmwired.net/kernel/Documentation/lockdep-design.txt]

[7] Jonathan Corbet, „The Kernel Lock Validator“: [http://lwn.net/Articles/185666/]

[8] Ingo Molnar, „Method and system for a kernel lock validator“, Patent: [http://www.faqs.org/patents/app/20080294892]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Zurzeit arbeiten Sie an der dritten Auflage ihres Buches „Linux Treiber entwickeln“.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 52

Kern-Technik

Simulierende Module ermöglichen es, auf Hardware zuzugreifen, die gar nicht im Rechner eingebaut ist. Dreh- und Angelpunkt dafür ist ein modifizierter Gerätetreiber. Eva-Katharina Kunst, Jürgen Quade



Virtualisierung macht eine typischerweise nur einmal vorhandene Hardware mehreren Komponenten scheinbar exklusiv zugänglich. Es geht aber noch mehr: Per Simulation greifen Linux-Anwender auf Hardware zu, die in ihren Rechnern überhaupt nicht existiert.

Das ist nützlich, wenn diese sich beispielsweise noch in der Entwicklung befindet, ihre Entwickler aber bereits notwendige Treiber und erste Applikationen programmieren und testen wollen – oder wenn eine Hardware schlichtweg zu teuer ist. Dann sammeln die Entwickler bereits erste Erfahrung mit der nicht vorhandenen Hardware, selbst wenn Zeitverhalten und eventuell auch der Funktionsumfang nicht vollständig dem Original entsprechen.

Hardwarezugriffe umbiegen

Was Linux-Interessierte dafür benötigen: ein Simulationsprogramm und ein Treiber (siehe Abbildung 1). Bei ihm darf es sich auch um ein bereits existierendes Exemplar handeln, das für die Simulation nur ein wenig zu modifizieren ist. In jedem Fall aber ist der Treiber der Dreh- und Angelpunkt des Ganzen, denn er leitet die Anfragen der Applikation entweder an die real existierende Hardware oder eben an die alternativ oder ausschließlich eingesetzte Simulation weiter.

Ein gutes Softwaredesign vorausgesetzt muss der Entwickler einen Treiber nur minimal verändern, um eine Simulation möglich zu machen. Konkret heißt dies, die Zugriffe auf die reale Hardware (siehe Kasten „Memory-mapped versus IO-mapped“), die über die Maschinenbefehle »in*X*«, »out*X*« beziehungsweise über die Funktionen »read*X*()« sowie »write*X*()« erfolgen, auf »simulation_in*X*()«, »simulation_out*X*()«, »simulation_read*X*()« und »simulation_write*X*()« umzubiegen (siehe Abbildung 2). Die so modifizierten Funktionen reichen die übergebenen Daten dann bei Bedarf an das eigentliche Simulationsprogramm weiter.

Für den erfahrenen C-Programmierer ist das Umbiegen von Funktionen ein klarer Fall von Funktionspointern. Die könnten sie beispielsweise »mapper_in*X*«, »mapper_out*X*«, »mapper_read*X*« oder »mapper_write*X*« nennen. Das Anhängsel »*X*« steht übrigens für den jeweils ersten Buchstaben von Byte, Word oder Long, je nachdem, ob ein 1-, 2- oder 4-Byte-Zugriff erfolgt. »mapper_readb« zeigt also auf eine Funktion, die genau ein einzelnes Byte einliest.

Das Simulationsprogramm selbst lässt sich bei hohen Anforderungen an das Zeitverhalten im Kernel realisieren. Attraktiver ist jedoch eine Applikation im Userland, die vor allem einfacher zu implementieren und zu konfigurieren ist. Sie erleichtert auch eine Visualisierung der simulierten Hardware.

Simulator ankoppeln

Als Schnittstelle zwischen Simulationsprogramm und modifiziertem Treiber kommen vor allem das Proc-Filesystem [1], das Sys-Filesystem [2], das Netlink-Interface [3] oder die normale Treiberschnittstelle in Frage. Letztere ist mit den Systemcalls »open()«, »close()«, »read()« und »write()« gerade dem Applikations-

Memory-mapped versus IO-mapped

Peripherie, die ein Rechner wie Hauptspeicher über den normalen Adressbus in den Adressraum einer CPU einblendet, nennt man Memory-mapped. Einige Prozessorhersteller – allen voran Intel – haben zusätzlich zum normalen Adressbus einen zweiten, abgespeckten Adressbus eingebaut, der vor allem der Ankopplung von IO-Hardware dient (IO-mapped). Historisch betrachtet erweiterte das nicht nur den Adressraum, sondern sparte gleichzeitig den Hardware-Aufwand für die Dekodierung der Peripherie. Software-technisch muss die CPU natürlich erfahren, wenn sie eine Adressinformation an den neuen Adressbus, dem so genannten IO-Bus, anlegen soll. Die Assemblerbefehle, die den Standardbefehl »mov« für den Zugriff auf Speicher ergänzen, nennt Intel »in« und »out«.

Linux stellt dem Treiberprogrammierer Wrapperfunktionen für den direkten Hardwarezugriff zur Verfügung: Per »read*X*()« und »write*X*()« greift er auf Memory-mapped-Hardware zu, mit den Makros »in*X*« und »out*X*« auf die Port-mapped-Hardware.

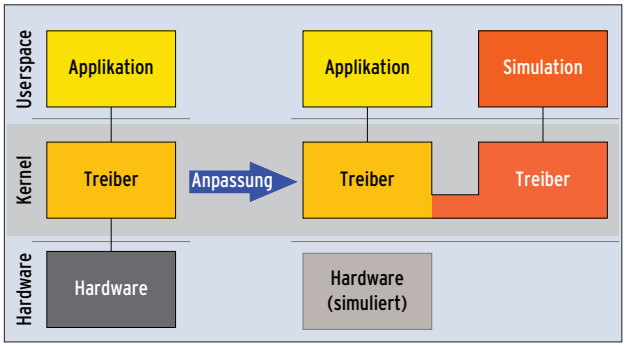


Abbildung 1: Ein modifizierter Treiber in Kombination mit einem Simulationsprogramm ermöglicht es, Hardware vorzutäuschen, ohne dass Entwickler ihre Applikationen dafür verändern müssen.

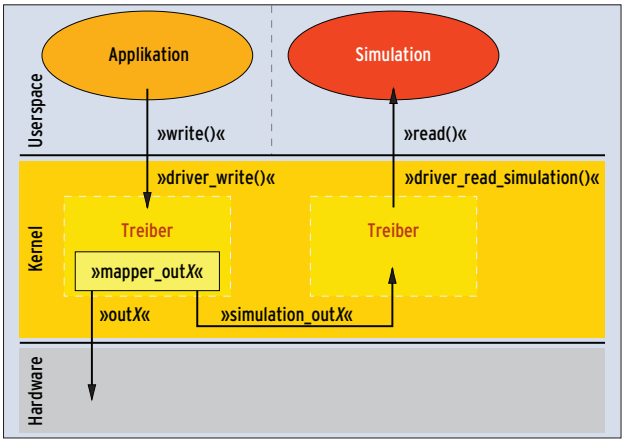


Abbildung 2: Funktionspointer biegen im Treiber den Hardwarezugriff auf die Simulation um. Dargestellt ist hier nur das Schreiben auf die Hardware.

programmierer gut bekannt. Das Simulationsprogramm erhält Informationen vom Treiber dadurch, dass es per »read()« auf eine zusätzliche Gerätedatei zugreift. Einen neuen Hardwarestatus übermittelt der Kernel dem modifizierten Treiber mit Hilfe von »write()«.

Etwas tricky

Die zusätzliche Gerätedatei stellt der Treiber zur Verfügung. Implementierungstechnisch reserviert der Kernelentwickler dazu einfach eine zusätzliche Gerätenummer und legt das zugehörige Sys-FS-Verzeichnis an. Dadurch erzeugt Udev automatisch die Gerätedatei, über die die Simulationsapplikation zugreift [4]. Allerdings muss der Treiber später differenzieren, ob das Simulationsprogramm oder die eigentliche, unveränderte Applikation die Daten transferiert.

Dieses Verhalten implementiert häufig Code in den Funktionen »driver_read()« und »driver_write()«, indem sie die Gerätenummer auswerten. Die sich hierdurch ergebende Modifikation der Lese- und Schreibfunktionen im Treiber ist unschön und lässt sich mit einem kleinen Trick umgehen. Typischerweise sind nämlich die Adressen der Zugriffsfunktionen »driver_read()« und »driver_write()« in der Struktur »struct file_operations« hinterlegt. Die »struct file«, die die zugreifende Applikationsinstanz repräsentiert, referenziert die Funktionen.

Die Idee besteht also darin, die »struct file_operations« auszutauschen, sobald ein Simulationsprogramm den Treiber öffnet (»driver_open()«). Als Ersatz kommt dann eine neue Struktur, die auf die speziellen Zugriffsfunktionen »driver_read_simulation()« und »driver_write_simulation()« verweist (siehe Listing 2, Zeile 55). Die nachfolgenden Aufrufe von »read()« und »write()« des Simulationsprogramms sind damit automatisch auf die eigenen Zugriffsroutinen umgelenkt (siehe Abbildung 3).

Bei einem Treiber, der sowohl reale als auch die simulierte Hardware kennt, fehlt noch die Mimik zur Aktivierung beziehungsweise Deaktivierung der Simulation. Dazu gibt es wieder einmal eine Reihe Möglichkeiten: Entwickler könnten beispielsweise einen Parameter definieren, den sie beim Laden des Treibers setzen [5].

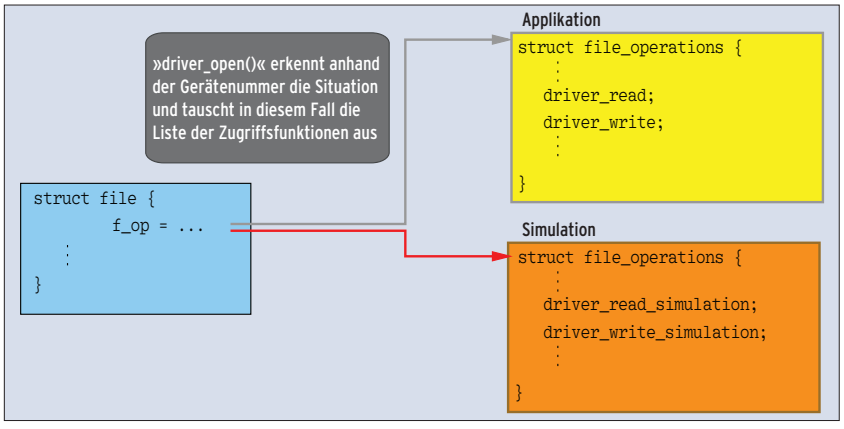


Abbildung 3: Beim Öffnen eines simulierten Geräts verweist das Modul fortan auf einen Satz neuer Routinen. Durch den Austausch der Zugriffsfunktionen bleibt der ursprüngliche Treibercode weitgehend unverändert.

Elegant aktivieren

Etwas eleganter ist es, eine Sys-Datei anzulegen, die nach dem Laden des Treibers zwischen realer Hardware und Simulation hin- und herschaltet. Am elegantesten gelingt der Code, wenn er im Treiber zusätzlich zum Sys-File automatisch die Simulation aktiviert, sollte er keine Hardware erkennen. Bei einem Treiber für eine PCI-Hardware wertet der Programmierer dazu zum Beispiel den Rückgabewert von »pci_register_driver()« in der Funktion »mod_init()« aus. Falls beim Anmelden kein Gerät existiert, gibt die Funktion »-ENODEV« zurück (siehe Listing 1).

Listing 2 zeigt zusammen mit Listing 3 und Listing 4 exemplarisch die Komponenten einer simulierten Hardware. Anschauungsobjekt ist ein Hardwarezähler, der seinen Wert im Sekundentakt

Listing 1: Ohne PCI auf Simulation umschalten

```
01 if (pci_register_driver(&pci_drv) == -ENODEV) {
02     /* Simulation aktivieren */
03     mapper_inl  = simulation_inl;
04     mapper_outl = simulation_outl;
05 } else {
06     mapper_inl  = inl;
07     mapper_outl = outl;
08 }
```

Listing 2: Modifizierter Treiber »lm_driver.c«

```
001 #include <linux/fs.h>
002 #include <linux/cdev.h>
003 #include <linux/device.h>
004 #include <asm/uaccess.h>
005
006 #define DEV_NAME "linux-magazin"
007 #define DEV_NAME_SIMU "linux-magazin.simu"
008
009 static dev_t lm_dev_number;
010 static struct cdev *driver_object;
011 struct class *lm_class;
012 static struct device *lm_dev, *lm_dev_simu;
013
014 static u32 data_register;
015 static u32 (*mapper_readl)(void *addr);
016
017 static int driver_open(struct inode *devicefile, struct file *filp);
018
019 ssize_t
020 application_read(struct file *instance, char __user *buffer,
021     size_t max_bytes_to_read, loff_t *offset)
022 {
023     size_t to_copy, not_copied;
024     u32 reg = mapper_readl((void *)0x12345678);;
025
026     to_copy    = min(sizeof(reg), max_bytes_to_read);
027     not_copied = copy_to_user(buffer, &reg, to_copy);
028     return to_copy - not_copied;
029 }
030
031 static ssize_t
032 simulation_write(struct file *instance, const char *userbuf,
033                  size_t length, loff_t *o)
034 {
035     int to_copy, not_copied;
036
037     to_copy    = min(length, sizeof(data_register));
038     not_copied = copy_from_user(&data_register, userbuf, to_copy);
039     printk("simulation_write: data_register=0x%x\n", data_register);
040     return to_copy - not_copied;
041 }
042
043 static struct file_operations fops_application = {
044     .open = driver_open,
045     .read = application_read,
046 };
047
048 static struct file_operations fops_simu = {
049     .write = simulation_write,
050 };
051
052 static int
053 driver_open(struct inode *devicefile, struct file *filp)
054 {
055     if (MINOR(devicefile->i_rdev) == 0)
056         filp->f_op = &fops_application;
057     else
058         filp->f_op = &fops_simu;
059     return 0;
060 }
061 static u32
062 simulation_readl(void *addr)
063 {
064     return data_register;
065 }
066
067 static int
068 __init mod_init(void)
069 {
070     mapper_readl = simulation_readl; // Simulation aktivieren
071
072     if (alloc_chrdev_region(&lm_dev_number, 0, 2, DEV_NAME) < 0)
073         return -EIO;
074     driver_object = cdev_alloc();
075     if (driver_object == NULL)
076         goto free_device_number;
077
078     driver_object->owner = THIS_MODULE;
079     driver_object->ops = &fops_application;
080
081     if (cdev_add(driver_object, lm_dev_number, 2))
082         goto free_cdev;
083
084     lm_class = class_create(THIS_MODULE, DEV_NAME);
085     if (IS_ERR(lm_class)) {
086         pr_err("template: no udev support\n");
087         goto free_cdev;
088     }
089     lm_dev = device_create(lm_class, NULL, lm_dev_number,
090                            NULL, "%s", DEV_NAME);
091     lm_dev_simu = device_create(lm_class, NULL, lm_dev_number + 1,
092                                 NULL, "%s", DEV_NAME_SIMU);
093     return 0;
094
095 free_cdev:
096     kobject_put(&driver_object->kobj);
097
098 free_device_number:
099     unregister_chrdev_region(lm_dev_number, 2);
100     return -EIO;
101 }
102
103 static void
104 __exit mod_exit(void)
105 {
106     device_destroy(lm_class, lm_dev_number);
107     device_destroy(lm_class, lm_dev_number + 1);
108
109     class_destroy(lm_class);
110
111     cdev_del(driver_object);
112
113     unregister_chrdev_region(lm_dev_number, 2);
114
115     return;
116 }
117
118 module_init(mod_init);
119 module_exit(mod_exit);
120 MODULE_LICENSE("GPL");
```

erhöht. In der realen Variante liest der Code den Zählwert durch Zugriff mittels »readl()« auf eine Memory-Adresse aus. Damit das Beispiel übersichtlich bleibt, ist der Treiber selbst auf die wesentliche Funktionalität reduziert und legt zugreifende Applikationen nicht schlafen. Im Normalfall aktiviert er die Simulation (siehe Listing 2, Zeile 70). Weitere Details des Treibers beschreibt [4].

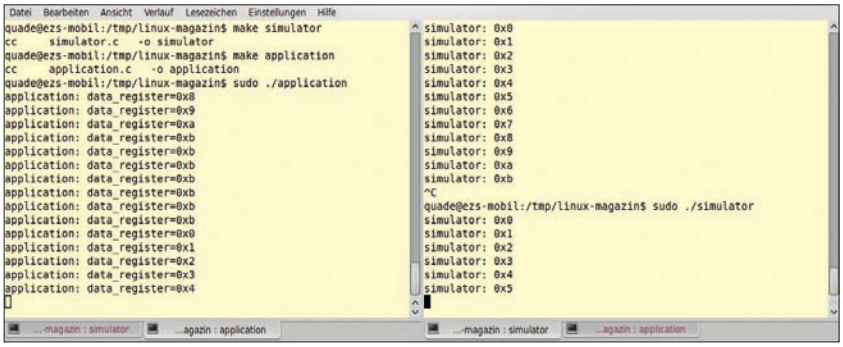


Abbildung 4: Der Zählwert der Applikation (links) ändert sich nur bei aktivem Simulator (rechts).

Hardware im Userland

Listing 3 zeigt das zugehörige Simulationsprogramm. Es erhöht sekündlich in einer Endlosschleife den Zählerwert und übergibt ihn per Systemcall »write()« auf das Gerät »/dev/linux-magazin.simu« an den Treiber. Der Simulationscode im Treiber speichert den Zählwert in der globalen Variablen »data_register« ab, sodass die Funktion »simulation_readl()« nur deren Wert zurückgibt (siehe Listing 2, Zeile 64). Die Applikation (siehe Listing 4) greift per »read()«-Systemcall auf die Gerätedatei »/dev/linux-magazin« zu und gibt den Wert unspektakulär auf der Konsole aus. Sie merkt dabei nicht, ob die Hardware real vorhanden ist oder ob Linux sie nur simuliert.

Wer den Code testen will, kompiliert den Treiber (siehe Listing 2), das Simulationsprogramm (siehe Listing 3) und die Applikation (siehe Listing 4). Er lädt zunächst den Treiber, startet dann das Simulationsprogramm und parallel dazu die Applikation. Die beiden Userland-Programme erfordern meist Root-Rechte, da Udev die Gerätedateien ohne weitere Konfiguration zunächst nur für den Superuser freigibt. Die Konsole der Applikation zeigt, dass sich der Wert des Zählers so lange erhöht, wie die Simulation aktiv ist (siehe Abbildung 4).

Noch ein Hinweis zum Simulationsprogramm: Damit es ein- und ausgehende Daten unabhängig behandelt, wählen erfahrene Programmierer einen auf Threads fußenden Ansatz. Der erste von ihnen wartet auf die Daten, die an die simulierte Hardware gerichtet sind, der zweite Thread generiert die Daten, die von der simulierten Hardware kommen.

Taktfrequenz erhöhen

Übungswillige ergänzen das Beispiel, das Abbildung 5 noch einmal zusammenfasst, um schreibenden Hardwarezugriff, mit dem sie zum Beispiel die Zählfrequenz ändern. Die Implementation erfolgt analog: Die Schreibfunktion »driver_write()« kopiert den von der Applikation zur Verfügung gestellten Frequenzwert vom

1. Variablen anlegen:
   Funktionszeiger für Hardware-Zugriffsfunktionen »struct file_operations« vorbereiten
2. »mod_init()« anpassen:
   Initialisierung
   Zusätzliche Gerätenummern anfordern
   Zusätzliches Sys-Filesystem-Verzeichnis anlegen
   Simulationsmodus ein- und ausschalten
   - »mapper_in*X*« »mapper_read*X*«
   - »mapper_out*X*« »mapper_write*X*«
3. »mod_exit()« anpassen:
   Deinitialisierung
   Sys-Filesystem-Verzeichnis löschen
   Gerätenummer freigeben
4. Zugriffsfuktionen für den Simulator schreiben:
   »driver_read_simulation()«
   »driver_write_simulation()«
5. Modul an den Simulator ankoppeln:
   - »simulation_in*X*« »simulation_read*X*()«
   - »simulation_out*X*« »simulation_write*X*()«
6. »driver_open()« anpassen:
   »struct_fileoperations« austauschen

Abbildung 5: Sechs Schritte benötigen Entwickler, um bestehende Treiber simulationsfähig zu machen.

Listing 3: Simulator »simulator.c«

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <linux/types.h>
04
05 int main(int argc, char **argv, char **envp)
06 {
07     int fd;
08     __u32 data_register = 0;
09
10     fd = open("/dev/linux-magazin.simu",
11               O_RDWR);
12     if (fd < 0) {
13        perror("/dev/linux-magazin.simu");
14        return -1;
15     }
16     while (1) {
17        sleep(1);
18
19        write(fd, &data_register,
20              sizeof(data_register));
21        printf("simulator: 0x%x\n",
22               data_register);
23        data_register++;
24     }
25     return 0;
26 }
```

Listing 4: Testanwendung »application.c«

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <linux/types.h>
04
05 int main(int argc, char **argv, char **envp)
06 {
07     int fd;
08     __u32 data_register;
09
10     fd = open("/dev/linux-magazin", O_RDWR);
11     if (fd < 0) {
12        perror("can't open /dev/linux-magazin");
13        return -1;
14     }
15
16     while (1) {
17        read(fd, &data_register,
18             sizeof(data_register));
19        printf("%s: data_register=0x%x\n",
20               argv[0], data_register);
21        sleep(1);
22     }
23     return 0;
24 }
```



Abbildung 6: Insgesamt 49 anspruchsvolle Fragen waren beim Kern-Technik-Diplom zu beantworten.

Userspace in den Kernel und übergibt ihn per »mapper_outl()« an die Hardware. Da die Funktion »mapper_outl()« auf den Simulator zeigt, ruft Linux im Treiber »simulator_outl()« auf. Im einfachsten Fall pollt der Simulator ständig den Treiber nach neuen Daten, effizienter ist es natürlich, wenn er per blockierendem »read()« auf neue Informationen wartet. Die »simulator_outl()«-Funktion würde dann die schlafende Instanz aufwecken. Sowohl Simulation wie auch Umlenkung im Treiber sind für die Darstellung auf das Minimum reduziert.

Schwieriges Hardwaredesign

Die gezeigte Basisfunktionalität lässt sich leicht erweitern, etwa um simulierte Interrupts oder auch um vorgegaukelte USB-Geräte. Allerdings soll die kompakte Darstellung nicht den Blick dafür verdunkeln, dass die Simulation realer Hardware oft komplex gerät und wesentlich mehr Code verlangt. Das ändert aber nichts an der grundsätzlichen Lösung, im Treiber die Hardwarezugriffe umzubiegen und über beispielsweise ein zweites Device dem Simulator zuzuführen. *(mg)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 7: Linux-Magazin 02/04, S. 48
[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 6: Linux-Magazin 01/04, S. 94
[3] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 40: Linux-Magazin 07/08, S. 96
[4] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 48: Linux-Magazin 11/09, S. 98
[5] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 10: Linux-Magazin 05/04, S. 94

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Zur Zeit arbeiten sie an der dritten Auflage ihres Buches „Linux-Treiber entwickeln“.

Kern-Technik-Diplom

Zusammen mit der 50. Folge der Kern-Technik-Serie stellte das Linux-Magazin das anspruchsvolle Wissens-Quiz „Kern-Technik-Diplom“ online (siehe Abbildung 6). Dabei ging es ins Eingemachte: Gefragt war unter anderem, wie viele Bits der Datentyp »int« auf einer 64-Bit-Maschine breit ist (32 Bit), was die Kernelfunktion »round_jiffies()« mit Powermanagement zu tun hat (der Kernel spart durch längeres Schlafen Strom), wofür im Kontext von USB die Abkürzung URB steht (USB Request Block) oder mit welchen Funktionen Kern-Hacker Zeitvergleiche programmieren (»time_after()«, »time_before()«, »time_after_eq()« und »time_before_eq()«). 49 Multiple-Choice-Fragen – zu jeder Folge der Kern-Technik eine – prüften das Know-how in der Breite und der Tiefe.

Harter Tobak trotz NSA-Fragen

Für Auflockerung sorgten einige der vordefinierte Antworten: Selbstverständlich steht „Inotify-Subsystem“ nicht für „Intelligence Notification System“, das Informationen an einen geheimen NSA-Port weiterleitet, bei „Darthvader“ handelt es sich nicht um einen IO-Scheduler, und die „Seek-Distanz“ gibt keineswegs die räumliche Entfernung der Festplatte zu Fort Meade an.

Kern-Techniker schnitten gut im Test ab

Das Ergebnis der Diplomprüfung kann sich sehen lassen: Im Durchschnitt 80 Prozent korrekte Antworten zeugen von der hohen Kompetenz der Teilnehmer. Mit 98 Prozent richtigen Antworten sticht Stefan Roas hervor, der als bester Teilnehmer nahezu jede Aufgabe korrekt löste. Die „Top Ten beim Kern-Technik-Diplom“ erhalten vom Linux-Magazin neben einer Urkunde das digitale Themenpaket „Kern-Technik“ mit den Folgen 1 bis 49. Und sobald das Buch der Autoren „Linux-Treiber entwickeln“ Ende des Jahres in aktualisierter Auflage erscheint, erhält Muster-Entwickler Stefan Roas ein Exemplar für sein beeindruckendes Kernel-Wissen.

Top Ten beim Kern-Technik-Diplom

Platz	Name	Punkte
1	Stefan Roas	97,7
2	Erik Braun	95,0
3	Marcus Nutzinger	94,0
4	Tobias Klauser	90,8
5	Bernhard M. Wiedemann	90,7
6	Matthias Vogt	88,2
7	Rolf Eike Beer	87,8
8	Andreas Bießmann	87,5
9	Florian Pfanner	83,8
10	Jan Engelhardt	83,2

Wer neugierig auf die Fragen ist, kann seine Kenntnisse immer noch unter [http://www.linux-magazin.de/kernspalterei] unter Beweis stellen. Anders als noch bei der Prüfung anlässlich der Jubiläumsausgabe teilt der Web-Test den Teilnehmern aber jetzt die korrekte Lösung sofort mit.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 53

Kern-Technik

Der Kthreadd ist der Vater aller Kernelthreads. Die von ihm abgeleiteten Codesequenzen erben einen eindeutigen Kontext. Für Entwickler von Vorteil: weniger Arbeit und weniger Fehler im Code. Eva-Katharina Kunst, Jürgen Quade



Kernelthreads haben einen festen Platz im Betriebssystemkern. Ein Ubuntu 10.04 beispielsweise aktiviert – je nach Rechnerhardware – zwischen 50 und 80 derartiger Verarbeitungssequenzen. Sie sind unabhängig vom Userland, Linux weckt sie nur bei Bedarf auf und sie belasten das System kaum. Sie sind effizient und dank der Realzeit-Eigenschaften des Linux-Kernels nur mit geringen Latenzzeiten behaftet. Daher setzen die Kernelhacker sie ein, um Jobs zwischen Rechnerkernen zu verschieben, regelmäßig Daten des Pagecache auf die Festplatte zu sichern oder um Daten zu ver- beziehungsweise zu entschlüsseln.

Fußgängern droht Stolpergefahr

Da die rudimentäre Variante, einen Thread durch Aufruf der Funktion »kernel_thread()« zu erzeugen, fehlerträchtig ist, wie der Kasten „Kernelthreads“ beschreibt, stellt Linux mit dem im Kernel laufenden Kthreadd ein risikolos verwendbares Framework bereit. Es sorgt nämlich anders als ein schlichter neuer Thread für eine eindeutige Erbmasse: Die vom Kthreadd erzeugten Threads belegen keine Ressourcen im Userspace, öffnen keine Dateien, blockieren alle Signale, dürfen auf jeder CPU eines Multicore rechnen und besitzen anfangs eine eindeutige Priorität.

Daher delegieren clevere Entwickler die Aufgabe, einen neuen Thread zu starten, an die Komponente, die etwas davon versteht: den Kthreadd. Der Kernel aktiviert ihn als Vater aller Kernelthreads direkt nach dem Start des Init-Prozesses. Folglich besitzt der Threadverteiler auch die PID 2 (Abbildung 1). Er wartet auf Aufträge, neue Prozesse zu erzeugen und mit einem Namen zu versehen, um sie dann umgehend ausführen zu lassen.

Der Programmierer meldet dem Kthreadd drei Werte: erstens die Adresse der im Kontext des Thread abzuarbeitenden Funktion, zweitens ein Argument, das der Daemon der Funktion beim Aufruf übergibt, und drittens den neuen Threadnamen. Das zweite Argument ermöglicht es dem Programmierer, Threadfunktionen mehrfach zu verwenden: Er identifiziert damit die jeweilige Instanz.

Innerhalb der gewählten Funktion sind die für Kernelthreads üblichen Einschrän-

Kernelthreads

Kernelthreads sind unabhängige Codesequenzen, die der Scheduler wie normale Applikations-Threads aktiviert und die CPU abarbeitet. Folglich besitzen sie einen eigenen Prozess-Kontrollblock (siehe Kasten „Prozess-Kontrollblock“) und eine eigene PID.

Sie unterscheiden sich von normalen Applikations-Threads dadurch, dass Code und Daten im Kernelspace – und eben nicht im Userspace wie bei den Benutzerprogrammen – liegen. Außerdem wird ihre Abarbeitung im Kernel nur dann unterbrochen, wenn höher priore Aufgaben anliegen.

Kernelthreads erzeugt der Programmierer durch Aufruf der Funktion »int kernel_thread(int (*fn)(void*), void *arg, unsigned long flags)«. Das Argument »fn« bezeichnet die Adresse der im Rahmen des Thread zu startenden Funktion und »arg« ein Argument, das ihr der Kernel beim Starten übergibt. Der Parameter »flags« steuert die Thread-Erzeugung über verschiedene, in der Include-Datei »linux/sched.h« definierte Konstanten. Bei Kernelthreads verwenden Entwickler dazu typischerweise die Kombination aus »CLONE_FS«, »CLONE_FILES« und »CLONE_SIGHAND«, um sich der nicht mehr benötigten Filesystem-Information zu entledigen und zunächst die gleichen Filedeskriptoren und Signalhandler des „Erzeugers“ (Elternprozess) zu übernehmen.

Innerhalb der Threadfunktion geben Programmierer durch Aufruf der Funktion »daemonize()« zum einen die geerbten Ressourcen im Userland frei (Code- und Datensegmente), zum anderen geben sie dem Kernelthread einen Namen. Verantwortungsbewusste Entwickler legen ihre Threads regelmäßig schlafen, damit der Scheduler auch gleich oder niedriger priorisierte Prozesse auswählen darf.

Schließlich verhindern ein Completion-Objekt und der Aufruf der Funktion »complete_and_exit()«, dass Linux den Threadcode entlädt, bevor der Thread sich beendet. Unterlässt der Programmierer das oder sorgt darüber hinaus nicht dafür, dass der neue Kernelthread eine eindeutige Priorität besitzt und auf allen Rechnerkernen einer Mehrprozessormaschine arbeiten darf, sind Fehlersituationen programmiert!

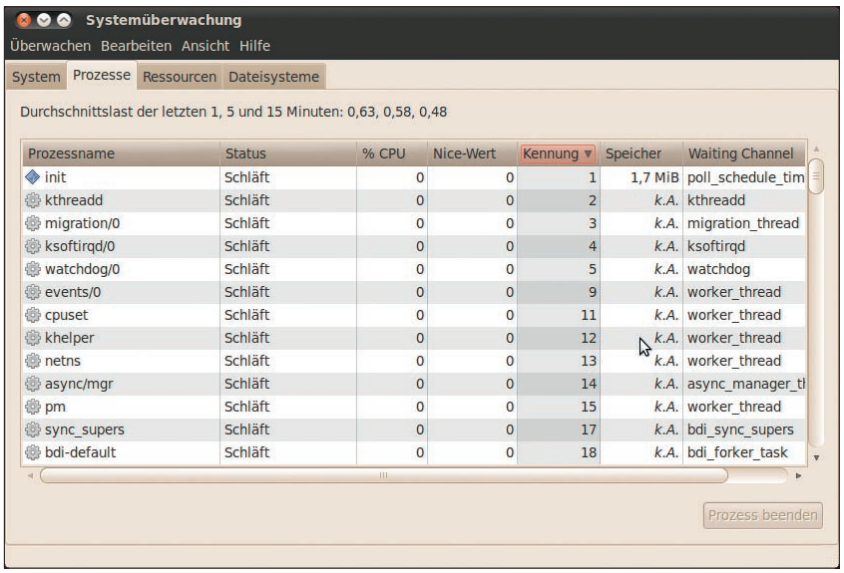


Abbildung 1: Linux startet den Kthreadd als zweiten Prozess. Von ihm leitet das Betriebssystem die meisten übrigen Kernelthreads ab (im Bild alle nachfolgenden).

kungen zu beachten: Der Programmierer gibt daher nur die Signale frei, die zu einer Unterbrechung führen dürfen. Sind Threads einmal aktiv, werden nur noch Geschwister sie unterbrechen, die eine höhere Priorität genießen. Daher verhält sich der Entwickler auch bei den über den Kthreadd erzeugten Threads kooperativ und legt seine Codesequenzen regelmäßig schlafen (siehe Abbildung 2). So hat jeder Thread eine Chance auf die CPU. Nach dem Aufwachen prüft er, ob ein Signal den Job geweckt hat. War dies der Fall, behandelt er das Signal und beendet normalerweise den Thread.

Reine Kernelthreads

Falls ein Kernelmodul den Thread startet, fängt der Programmierer noch eine Race Condition ab: Auf keinen Fall darf die Bearbeitungssequenz aktiv sein, wenn ein Anwender das Modul entlädt. Vorher beendet es seinen Thread beispielsweise mit Hilfe eines Signals und wartet auf eine Vollzugsmeldung. Dabei hilft der im Kasten „Completion-Objekt“ beschriebene Mechanismus.

Ein Programmierinterface macht die Auftragsvergabe leicht. Es besteht aus den in Tabelle 1 gelisteten Funktionen. Um einen neuen Kernelthread zu erzeugen und gleichzeitig auch zu aktivieren, wählt der Programmierer das Makro »kthread_run()«. Es bekommt mindestens drei Parameter übergeben.

Mit dem ersten Parameter nennt er die Adresse der Funktion, die Linux im Rahmen des Thread ausführen soll. Zweitens übergibt er einen Parameter, den das Makro seinerseits der Threadfunktion überantwortet. Drittens schließlich spezifiziert er den Namen des Thread. Ihn gibt er – wie bei »printf()« – in Form eines Formatstrings mit nachfolgenden Parametern an: Die Funktion hat also tatsächlich eine variable Anzahl von Argumenten (Listing 1, Zeile 31).

Die eigentliche Funktion für das Anlegen des Kernelthread als Kindprozess des

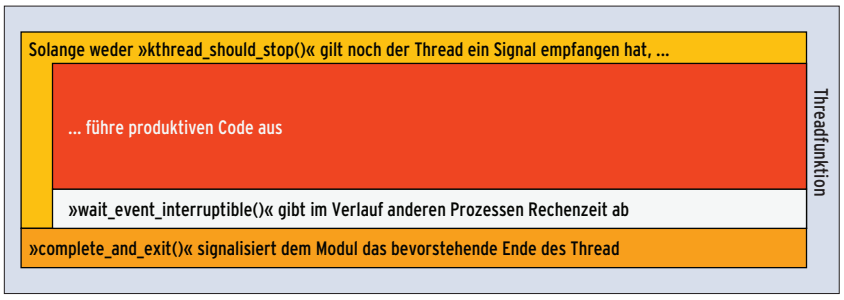


Abbildung 2: Der Funktionscode wertet »kthread_should_stop()« aus, um sich bei Signalen zu beenden.

Prozess-Kontrollblock

Jeder Rechenprozess, also auch die Kernelthreads, ist im Betriebssystemkern über eine Datenstruktur, den so genannten Prozess-Kontrollblock (PCB), eindeutig identifiziert. Im PCB sind beispielsweise der Prozesszustand, der Name, die PID, die Priorität, die zur Abarbeitung freigegebenen Rechnerkerne ebenso hinterlegt wie die Adressen der Segmente, in denen sich Code und Daten des Jobs befinden. Bei der Prozessorganisation untersucht der Scheduler die Liste der PCBs, um daraus den nächsten Job auszuwählen, den er dann abarbeitet.

Im Linux-Kernel lautet der Name des PCB »struct task_struct«. Die Struktur ist in der Headerdatei »linux/sched.h« definiert.

Kthreadd ist allerdings »kthread_create()«. Die Parameter sind identisch zum Makro »kthread_run()«, unterschiedlich ist allein, dass der Thread nur angelegt und die eigentliche Threadfunktion (produktiver Code) nicht direkt gestartet (ausgeführt) wird.

Erst auf Kommando starten

Zum Starten muss der Programmierer erst ein »wake_up_process()« mit der Adresse des Prozess-Kontrollblocks aufrufen (siehe Kasten „Prozess-Kontrollblock“). So versetzt er den Thread schon mal in eine Warteposition und kann ihn dann mit kurzer Latenz aktivieren.

Falls er es sich nach dem Starten und vor dem Aktivieren jedoch anders überlegt hat und den Thread beenden will, bevor der produktive Code überhaupt abgearbeitet ist, ruft er vor dem Aufwecken die Funktion »kthread_stop()« auf. Diese setzt in einer zum Thread gehörenden Datenstruktur das Flag »should_stop«. Der Zustand des gesetzten Flag lässt sich mit der Funktion »kthread_should_stop()« prüfen (Abbildung 3).

Eine weitere Funktion ist »kthread_bind()«. Mit ihr fesselt der Entwickler auf einem Mehrkernsystem den Kernelthread an einen oder an mehrere Prozessoren und zwingt den Kernel damit, ihn dort exklusiv abzuarbeiten.

Threadfunktionen, die der Kthreadd startet, müssen nicht zwangsläufig, dürfen aber »do_exit()« am Ende ihres Code aufrufen. Ein einfaches »return« reicht, weil der Kthreadd der eigentlichen Thread-

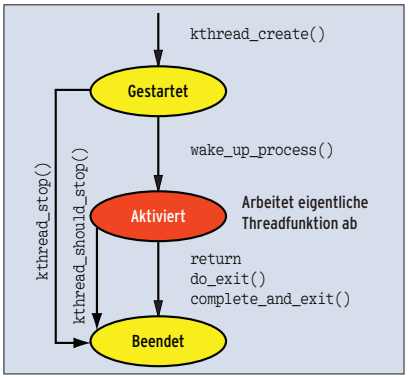


Abbildung 3: Per Kthreadd erzeugte Kernelthreads durchlaufen bis zu drei Zustände.

Tabelle 1: Funktionsübersicht

Funktion	Beschreibung
struct task_struct *kthread_create(int (*threadfn) (void *data), void *data, const char namefmt[], ...)	Erzeugt einen Kernelthread, »wake_up_process()« startet ihn
struct task_struct *kthread_run(int (*threadfn) (void *data), void *data, const char namefmt[], ...)	Das Makro erzeugt und startet anschließend einen neuen Kernelthread
int kthread_should_stop(struct task_struct *process);	Das Makro überprüft das per »kthread_stop()« gesetzte Flag
int kthread_stop(struct task_struct *process);	Beendet einen gestarteten, aber nicht aktiven Thread, ohne seine Funktion abzuarbeiten
int kthread_bind(struct task_struct *process, unsigned int cpu);	Lässt den Kernelthread »process« auf den in »cpu« angegebenen Kernen arbeiten
int wake_up_process(struct task_struct *process);	Aktiviert den Thread »process«, der sich schlafen gelegt hatte

funktion noch Code vor- beziehungsweise nachlagert (Abbildung 4).

Intern erzeugt der Kthreadd die neuen Kernelthreads. Der Einstieg in den Daemon aus Kernelcode, etwa einen Treiber, ist die Funktion »kthread_create()«. Sie hängt in eine Auftragsliste Namen, Parameter und Funktionsadresse des Thread ein und weckt den Kthreadd auf. Der holt sich den Auftrag und ruft seinerseits die Funktion »kernel_thread()« auf und erzeugt somit einen neuen Thread, startet allerdings noch nicht die eigentliche Threadfunktion. Stattdessen legt vorgeschalteter Code den neuen Thread erst einmal schlafen. Er wartet auf den Aufruf von »wake_up_process()«.

Erfolgt der, überprüft der Daemon, ob er den Thread nicht direkt wieder beenden soll, indem er »should_stop« auf 1 hin prüft – und zwar noch bevor er die Threadfunktion tatsächlich aufruft. Steht das Ende unmittelbar bevor, folgt »do_exit()«, ansonsten führt der Kthreadd den produktiven Code aus, nämlich die vom Kernelcode beziehungsweise Treiber registrierte Threadfunktion.

Konsistenz erhalten

Besonders diffizil ist das Sichern der diversen kritischen Abschnitte. So müssen die Programmierer verhindern, dass die Liste, in der Linux die Aufträge einträgt, inkonsistent wird. Dazu verwenden sie ein Spinlock. Zudem unterbindet das Betriebssystem, dass anderer Code »wake_up_process()« aufruft, bevor sein Erzeuger den neuen Thread überhaupt gestartet hat. Hierzu setzen die Kernelentwickler auf ein Completion-Objekt, das Element »done« in der zur Verwaltung des Thread verwendeten Datenstruktur. Ein weiteres Objekt kommt zum Einsatz, um das Ende des neuen Thread abzuwarten, wenn sein Erzeuger den Thread vor dem eigentlichen Aktivieren gleich wieder beenden will.

Listing 1 zeigt Code, der einen Thread mittels Kthreadd startet. Dessen Funktion kodieren die Zeilen 8 bis 26. Das pas-

Completion-Objekt

Das Completion-Objekt »struct completion« synchronisiert einen Ablauf mit dem Ende einer Codesequenz und realisiert die beiden Methoden »void complete(struct completion *)« und »void wait_for_completion(struct completion *)«. Die erste setzt der Entwickler an das Ende des zu überwachenden Abschnitts, die zweite Methode ruft der wartende Kernelthread oder die wartende Treiberfunktion auf.

Die Variante »complete_and_exit()« der Methode »complete()« setzen Entwickler bei Kernelthreads ein. Das Ende mehrerer Codesequenzen zeigt »void complete_all(struct completion *)« an.

Zu »wait_for_completion()« gibt es drei Varianten: »unsigned long wait_for_completion_timeout(struct completion *, unsigned long timeout)«, »int wait_for_completion_interruptible(struct completion *)« und »unsigned long wait_for_completion_interruptible_timeout(struct completion *, unsigned long timeout)«. Dazu kommen noch Makros und Funktionen, die das Objekt initialisieren. Das Makro »DECLARE_COMPLETION(struct completion *name)« legt für das Objekt statischen Speicher unter »name« an. Initialisiert der Programmierer ein Completion-Objekt zur Laufzeit, reserviert er erst den Speicher und übergibt dann die Adresse von »void init_completion(struct completion *)«.

Technisch steckt hinter dem Completion-Objekt eine Integer-Variable (Abbildung 6), die die Funktion »complete()« beim Aufruf inkrementiert. Die Funktion »wait_for_completion()« legt den Thread schlafen, wenn die Variable bereits 0 ist. Andernfalls dekrementiert sie die überwachende Variable.

Aktiviert der Kernel zum Beispiel erst einen Thread, der wegen »wait_for_completion()« auf das Ende einer anderen Codesequenz wartet, ist die Variable »done« noch 0. Folglich legt der Kernel den Thread schlafen. Er wird durch Aufruf von »complete()« wieder aufgeweckt. Ist die Reihenfolge jedoch andersrum – erst wird die Codesequenz mit »complete()« aufgerufen –, dann ist der Wert der Variablen »done« 1. Der später kommende Thread legt sich nicht schlafen, sondern dekrementiert nur »done«, sodass die Variable wieder den Wert 0 hat.

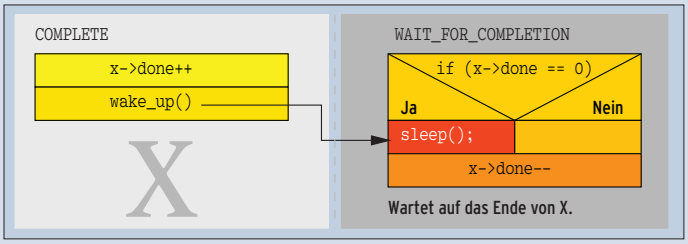


Abbildung 6: Vereinfacht dargestellt schläft der Thread, der auf das Ende einer Codesequenz wartet, bis die Variable »done« positiv ist.

Der Trick mit der Variablen ermöglicht es, mit einem Objekt auch auf das Ende mehrerer Codesequenzen auf einmal zu warten. Die Funktionen, auf die das Objekt warten soll, führen jeweils am Ende »complete()« aus, während die abwartende Instanz mehrfach die Funktion »wait_for_completion()« aufruft.

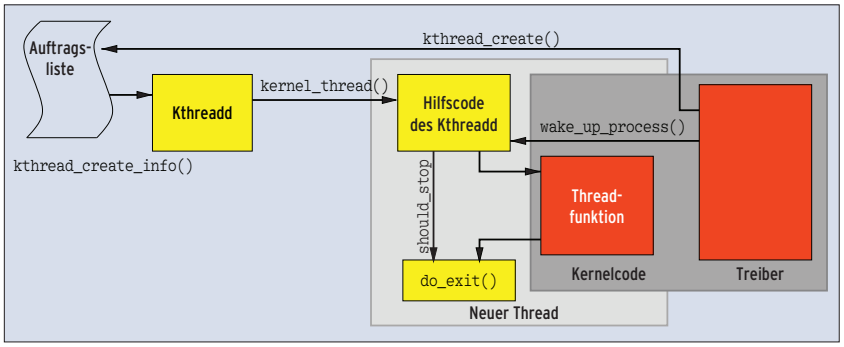


Abbildung 4: Treiber erzeugen Kernelthreads nicht mehr selbst, sondern delegieren diese Aufgabe an den Kthreadd. Der kümmert sich insbesondere um die Einbettung in den Schedulerbetrieb.

sende Completion-Objekt und der Aufruf der Funktion »complete_and_exit()« am Ende (Zeilen 6 und 25) sind nötig, wenn der Kernelthread aus einem Modul heraus starten soll. Das Completion-Objekt verhindert, dass Linux den Modulcode entlädt, bevor es den Kernelthread vollständig beendet hat. Es schießt den Thread durch Aufruf der Funktion »kill_pid()« ab und schickt so dem im ersten Parameter angegebenen Thread ein Signal. Um bei gegebenem Prozess-Kontrollblock an die zugehörige PID zu gelangen, verwenden Entwickler am einfachsten das Makro »task_pid()« wie in Zeile 41.

Zukunftsaussichten

Experimentierfreudige übersetzen den Code mit dem Makefile in Listing 2. Mit »insmod« laden sie das kompilierte Kernelobjekt und lesen im Syslog mit

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

die per »printk()« erzeugten Ausgaben des gestarteten Kernelthread. Abbildung 5 dokumentiert das Übersetzen und einen erfolgreichen Testlauf.

Kernelthreads spielen in modernen, echtzeitfähigen Kerneln eine besondere Rolle [1]. So ist es folgerichtig, dass Linus Torvalds und sein Team mit dem Kthreadd eine einfach handhabbare Umgebung schaffen, um ohne Risiko Codesequenzen zu erzeugen. Sie halten sich damit an den amerikanischen Geschäftsmann Rockefeller, der meinte: „Ich arbeite nach dem Prinzip, dass man niemals etwas selbst tun soll, was jemand anderes für einen erledigen kann.“ *(mg)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 34: Linux-Magazin 07/07

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Zur Zeit arbeiten Sie an der dritten Auflage ihres Buches „Linux-Treiber entwickeln“.

```
root@ezs-net: /tmp/linux-magazin
Datei Bearbeiten Ansicht Terminal Hilfe
root@ezs-net:/tmp/linux-magazin# make
make    -C /lib/modules/2.6.32-23-generic/build M=/tmp/linux-magazin modules
make[1]: Betrete Verzeichnis '/usr/src/linux-headers-2.6.32-23-generic'
  CC [M]  /tmp/linux-magazin/kthread.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/linux-magazin/kthread.mod.o
  LD [M]  /tmp/linux-magazin/kthread.ko
make[1]: Verlasse Verzeichnis '/usr/src/linux-headers-2.6.32-23-generic'
root@ezs-net:/tmp/linux-magazin# insmod kthread.ko
root@ezs-net:/tmp/linux-magazin# tail -n5 /var/log/messages
Jul  3 20:34:29 ezs-net kernel: [14386.052202] thread_function: woke up ... 0
Jul  3 20:34:30 ezs-net kernel: [14387.052152] thread_function: woke up ... 0
Jul  3 20:34:31 ezs-net kernel: [14388.052201] thread_function: woke up ... 0
Jul  3 20:34:32 ezs-net kernel: [14389.052195] thread_function: woke up ... 0
Jul  3 20:34:33 ezs-net kernel: [14390.052223] thread_function: woke up ... 0
root@ezs-net:/tmp/linux-magazin#
```

Abbildung 5: Wer mit Threads experimentieren möchte, übersetzt das Modul mit »make« und lädt es mit »insmod« in den Kernel. Die Ausgaben erscheinen im Syslog.

Listing 1: »kthread.c«

```
01 #include <linux/module.h>
02 #include <linux/kthread.h>
03
04 static struct task_struct *thread_id;
05 static wait_queue_head_t wq;
06 static DECLARE_COMPLETION(on_exit);
07
08 static int thread_code(void *data)
09 {
10    unsigned long timeout;
11    int i;
12
13    allow_signal(SIGTERM);
14    for (i = 0; i < 5; i++) {
15       timeout = HZ; // wait 1 second
16       timeout = wait_event_interruptible_timeout(wq,
17                                (timeout == 0), timeout);
18       printk("thread_function: woke up ... %ld\n",
19              timeout);
20       if (timeout==-ERESTARTSYS) {
21          printk("got signal, break\n");
22          break;
23       }
24    }
25    complete_and_exit(&on_exit, 0);
26 }
27
28 static int __init kthread_init(void)
29 {
30    init_waitqueue_head(&wq);
31    thread_id = kthread_create(thread_code, NULL,
32                               "mykthread%d", 0);
33    if (thread_id == 0)
34       return -EIO;
35    wake_up_process(thread_id);
36    return 0;
37 }
38
39 static void __exit kthread_exit(void)
40 {
41    kill_pid(task_pid(thread_id), SIGTERM, 1);
42    wait_for_completion(&on_exit);
43 }
44
45 module_init(kthread_init);
46 module_exit(kthread_exit);
47
48 MODULE_LICENSE("GPL");
```

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := kthread.o
03 else
04 KDIR    := /lib/modules/$(shell uname -r)/build
05 PWD     := $(shell pwd)
06
07 default:
08     $(MAKE) -C $(KDIR)  M=$(PWD) modules
09 endif
10
11 clean:
12     rm *.o *.ko
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 54

Kern-Technik

Der Blockdevice-Layer ist das Arbeitspferd bei Linux, was die Datenbeschaffung angeht – von der Festplatte bis zur Anwendung ist es ein langer Weg. Der I/O-Scheduler behält dabei den Überblick. Eva-Katharina Kunst, Jürgen Quade



Mit der Zunahme moderner Speichermedien wie Flash- und Solid-State-Disks fällt der Fokus der Kernelentwickler zunehmend auf das Blockgeräte-Subsystem. Festplatten, CD-ROMs und USB-Sticks spricht der Kernel bekanntlich anders an als zeichenorientierte Geräte wie Tastatur oder serielle Schnittstelle. Während diese ihre Daten Zeichen für Zeichen liefern, geben blockorientierte Geräte ihre Daten nur im Schwall ab. 512 Byte in einem Zug sind üblich.

Da Festplatten aufgrund physikalischer Gesetze, beispielsweise das der Trägheit, für den Datentransfer etwas mehr Zeit benötigen, entkoppelt der Unix-Kernel Applikations- und Hardware-Zugriffe durch einen Zwischenspeicher. Diesen füllt der Kernel auch schon mal in vorauseilendem Gehorsam beziehungsweise er sammelt Schreibaufträge der Applikation, bevor er der trägen Festplatte die Daten zur Ablage auf die Magnetscheiben übergibt. Der erzielte Performance-Gewinn durch diesen Page- oder Buffercache genannten Zwischenspeicher ist erheblich.

Diese Architektur sorgt für eine saubere Aufgabenverteilung. Oberhalb des Pagecache nimmt der Virtual Filesystem Switch (VFS) Aufträge von den Applikationen entgegen, die diese beispielsweise mit »open()«, »read()«, »write()« und »close()« erteilt haben.

Ablagetechnik

Der VFS ordnet dem Auftrag ein block- oder zeichenorientiertes Gerät zu. Da Applikationen Daten über Dateinamen referenzieren, folgt im Blockgeräte-Subsystem die Umsetzung auf Festplattenadressierung gemäß der im verwendeten Filesystem festgelegten Vorschrift, zum Beispiel Ext 4.

Die Festplatte adressiert Linux über eine Geräteadresse, die Sektornummer und der Anzahl der Sektoren. Hält es eine Kopie der adressierten Daten im Pagecache, findet der Transfer zwischen ihm und der Applikation statt und lässt den Blockgerätetreiber außen vor. Befindet sich im Pagecache jedoch keine Kopie, dann vergibt Linux einen Auftrag an den zugehörigen Blockgerätetreiber.

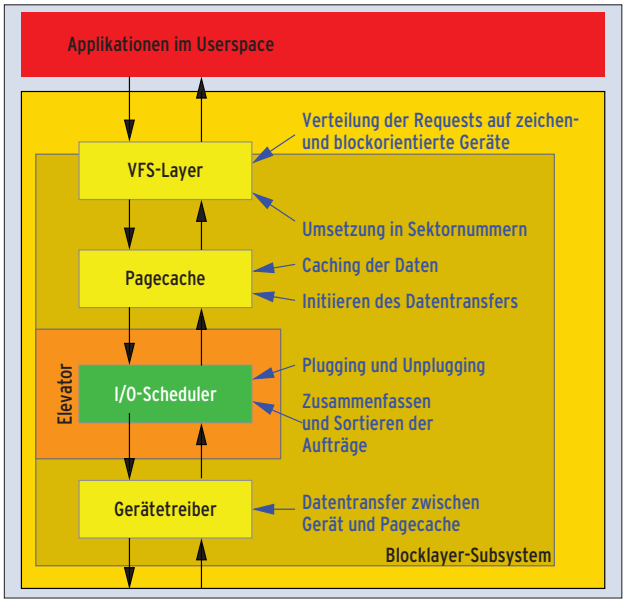
Ein solcher Request besteht aus einer Quelladresse, einer Zieladresse, der Transferrichtung und der zu transferierenden Datenmenge. Dieser Auftrag landet beim I/O-Scheduler ([1], [2]).

Der I/O-Scheduler ist die Instanz, die typischerweise eine Reihe von Aufträgen einsammelt. Zusätzlich fasst er einzelne Aufträge zusammen und sortiert sie. Das vermeidet unnötige und zeitaufwändige Kopfbewegungen bei der Festplatte. Der I/O-Scheduler schreibt die Aufträge dann schließlich in eine für jeden Hintergrundspeicher vorhandene Request-Queue und aktiviert den Gerätetreiber. Dieser arbeitet seine Request-Queue ab und transportiert die Daten zwischen Festplatte und Pagecache. Sobald das erledigt ist, organisiert der Kernel den Transfer zur Applikation.

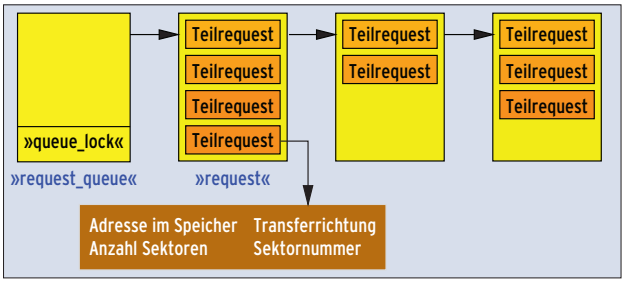
Wenngleich sich der grundsätzliche Aufbau des Blockgeräte-Subsystems nicht geändert hat, haben die Entwickler an den Schnittstellen gearbeitet. Sie tauschten mit der Zeit nicht nur Namen aus, sondern änderten auch vereinzelt Abläufe. Daher reicht es nicht, innerhalb eines vorhandenen Blockgerätetreibers beispielsweise der Version 2.6.30 nur die Funktionsnamen und Funksionsparameter zu ändern. Ein so kompiliertes Modul führt zum Stillstand des Gesamtsystems. Erst ein Reboot macht dann alles wieder gut. Für ein stabiles Linux tauschten die Entwickler zusätzlich Funktionen aus und passten Abläufe an.

Kern der Sache

Ein blockorientierter Gerätetreiber besteht im Kern aus den beiden Datenstrukturen »struct gendisk« und »struct request_queue« sowie einer Request-Funktion. Jeder vom Treiber bediente Hintergrundspeicher benötigt eine »struct gendisk«: Der Kernel entnimmt ihr unter anderem die Gesamtgröße der Festplatte in Sektoren, die Treiber-Identifikationsnummer (Majornummer, siehe **Kasten „Gerätenummern im Blockgerätetreiber“**), die maximale Anzahl der Partitionen (Minornummer) und den Namen der Festplatte, der maximal 32 Zeichen lang sein darf. Außerdem findet der Kernel dort den Verweis auf die Request-Queue.



◀ **Abbildung 1: Das Blockgeräte-Subsystem besteht aus mehreren Schichten.**



▲ **Abbildung 2: Der Kernel legt in der Request-Queue Aufträge ab, die in Teilaufträge (Segmente) unterteilt sind.**

Mit der Größe eines Blockes oder Sektors initialisiert der Kernel die vom Treiber zu reservierende Request-Queue. Mit der Adresse eines Lock legt er einen Schutz vor parallelem Zugriff an. Für die Reservierung dieser Datenstrukturen stellt der Kernel mit »alloc_disk()« und »blk_init_queue()« Funktionen zur Verfügung, die für die notwendige Initialisierung sorgen. Die »struct request_queue« beispielsweise belegt Linux dabei mit der Adresse der Request-Funktion des Treibers. Nach der Initialisierung übergibt das Betriebssystem die »struct gendisk« per »add_disk()« dem Blockgeräte-Subsystem. Es reserviert die zugehörigen Gerätedateien und gibt den Zugriff auf die Disk frei.

Vorarbeiter

Die Request-Funktion ist das zentrale Element eines Blockgerätetreibers, führt sie doch den eigentlichen Transfer durch. Der Aufbau dieser Funktion ist für viele Treiber ähnlich: Sie arbeitet Auftrag für Auftrag ab, wobei Aufträge aus mehreren Teilaufträgen, den Segmenten, bestehen dürfen (siehe Abbildung 2).

Linux entnimmt der Request-Queue mit Hilfe der Funktion »blk_fetch_request()« einen Auftrag. Da es nicht nur solche für den Datentransport gibt, überprüft der Treiber durch Aufruf von »blk_fs_request()« den Typ. Der Beispieltreiber in Listing 1 unterstützt nur Datentransfer und quittiert andere Aufträge ohne weitere Aktion. Andere Treiber verarbeiten hier Aufträge, um etwa die Festplatte in den Stromsparmodus zu schalten oder sie wieder aufzuwecken.

Ein Request, genauer gesagt jedes Segment, besteht aus den Transferinformationen Quelladresse, Zieladresse, Transferrichtung und Mengenangabe. Eine der beiden Adressen liegt im Hauptspeicher im Kernelspace, die andere auf dem Hintergrundspeicher. Adressen auf ihm geben Kernelhacker – ebenso wie die Mengenangabe – in Sektoren an. Dazu nennen sie den Startsektor und die Anzahl der zu transferierender Sektoren.

Transferleistung

Kompliziert wird der Transfer dadurch, dass der I/O-Scheduler Aufträge für den optimierten Zugriff auf Festplatten zusammenfasst. Liegen Datenblöcke auf der Festplatte in direkt aufeinanderfolgenden Sektoren, lassen die sich prima in einem Rutsch lesen. Jedoch muss der Kernel diese Datenpakete unter Umständen an

```
Datei  Bearbeiten  Ansicht  Verlauf  Lesezeichen  Einstellungen  Hilfe
quade@ezs-mobil:/tmp/linux-mag$ make
make    -C /lib/modules/2.6.32-24-generic/build M=/tmp/linux-mag modules
make[1]: Betrete Verzeichnis '/usr/src/linux-headers-2.6.32-24-generic'
  CC [M]  /tmp/linux-mag/ramdisk.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/linux-mag/ramdisk.mod.o
  LD [M]  /tmp/linux-mag/ramdisk.ko
make[1]: Verlasse Verzeichnis '/usr/src/linux-headers-2.6.32-24-generic'
quade@ezs-mobil:/tmp/linux-mag$ sudo insmod ramdisk.ko
[sudo] password for quade:
quade@ezs-mobil:/tmp/linux-mag$ sudo mke2fs /dev/bds0
mke2fs 1.41.11 (14-Mar-2010)
Dateisystem-Label=
OS-Typ: Linux
Blockgröße=1024 (log=0)
Fragmentgröße=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
32 Inodes, 256 Blöcke
12 Blöcke (4.69%) reserviert für den Superuser
Erster Datenblock=1
1 Blockgruppe
8192 Blöcke pro Gruppe, 8192 Fragmente pro Gruppe
32 Inodes pro Gruppe

Schreibe Inode-Tabellen: erledigt
Schreibe Superblöcke und Dateisystem-Accountinginformationen: erledigt

Das Dateisystem wird automatisch nach jeweils 24 Einhäng-Vorgängen bzw.
alle 180 Tage überprüft, je nachdem, was zuerst eintritt. Veränderbar mit
tune2fs -c oder -t .
quade@ezs-mobil:/tmp/linux-mag$ sudo mount /dev/bds0 /mnt
quade@ezs-mobil:/tmp/linux-mag$ ls /mnt
lost+found
quade@ezs-mobil:/tmp/linux-mag$
linux-mag : bash
```

Abbildung 3: Nach dem Kompilieren und dem Laden des Moduls legen Experimentierfreudige ein Filesystem an und mounten das Gerät. Die hervorgehobenen Kommandos benötigen dazu Root-Rechte.

unterschiedlichen Stellen im Pagecache des Hauptspeichers ablegen, da dieser gelegentlich fragmentiert. Daher zerlegt der Treiber den Request in Teilaufträge und arbeitet sie einzeln ab.

Der Treiber liest aus dem Request über die Makros »blk_rq_pos()« und »blk_rq_cur_sector()« oder »blk_rq_cur_bytes()« die Adressenangaben eines Teilauftrags für den Zugriff auf die Disk. Die Funktion »rq_data_dir()« ermittelt die Transferrichtung. Erst auf Basis dieser Angaben transportiert Linux die Daten zum Ziel.

Der Kernel antizipiert, dass er innerhalb der Request-Funktion ohnehin Manipulationen an der Request-Queue vornimmt. Daher sperrt er selber das Lock, ruft die Request-Funktion auf und gibt es hinterher wieder frei. Wenn allerdings die Request-Funktion längere Zeit läuft und dabei das Lock gesperrt bleibt, führt das zu unnötigen und ineffizienten Latenzzeiten. Aus diesem Grunde sollten verantwortungsbewusste Treiberentwickler in solchen Situationen das Lock zwischendurch freigeben, es aber vor Ende der Request-Funktion wieder sperren. Der Kernel kann ja nicht wissen, dass das Lock innerhalb der Funktion zwischenzeitlich frei wurde.

Das Lock ist übrigens der Übeltäter, der bei älterem Code zum Einfrieren des Systems sorgt. Nachdem der Treiber einen

Listing 1: Treiber für eine Ramdisk (»ramdisk.c«)

```
01 #include <linux/fs.h>
02 #include <linux/module.h>
03 #include <linux/blkdev.h>
04
05 #define SIZE_IN_KBYTES 256
06
07 static int major;
08 static struct gendisk *disk;
09 static struct request_queue *bdqueue;
10 static char *mempool;
11 DEFINE_SPINLOCK(bdlock);
12
13 static void bd_request(struct request_queue *q)
14 {
15     unsigned long start, to_copy;
16     struct request *req = blk_fetch_request(q);
17
18     while (req) {
19         printk("req: %d - ", req->cmd_type);
20         if (!blk_fs_request(req)) {
21             if (!__blk_end_request_cur(req, 0))
22                 req = blk_fetch_request(q);
23             continue;
24         }
25         start   = blk_rq_pos(req) << 9; // * 512
26         to_copy = blk_rq_cur_bytes(req);
27
28         spin_unlock_irq(q->queue_lock);
29         if ((start + to_copy) <= (SIZE_IN_KBYTES * 1024))
30             if (rq_data_dir(req) == READ) {
31                 printk("read %p %ld %lu\n",
32                        req->buffer, start, to_copy);
33                 memcpy(req->buffer, mempool + start, to_copy);
34             } else {
35                 printk("write %ld %p %lu\n",
36                        start, req->buffer, to_copy);
37                 memcpy(mempool + start, req->buffer, to_copy);
38             }
39         else
40             printk("%ld not in range ...\n", start + to_copy);
41         spin_lock_irq(q->queue_lock);
42         if (!__blk_end_request_cur(req, 0))
43             req = blk_fetch_request(q);
44         else
45             printk("followup: ");
46     }
47 }
48 static struct block_device_operations bdops =
49     { .owner = THIS_MODULE, };
50
51 static int __init mod_init(void)
52 {
53     if (!(major = register_blkdev(0,"bds"))) {
54         printk("bd: can't get majornumber\n");
55         return -EIO;
56     }
57     printk("major: %d\n", major);
58     if (!(mempool = vmalloc(SIZE_IN_KBYTES * 1024))) {
59         printk("vmalloc failed ...\n");
60         goto out_no_mem;
61     }
62     if (!(disk = alloc_disk(1))) {
63         printk("alloc_disk failed ...\n");
64         goto out;
65     }
66     disk->major       = major;
67     disk->first_minor = 0;
68     sprintf(disk->disk_name, "bds0");
69     set_capacity(disk, (SIZE_IN_KBYTES * 1024) >> 9); // 512 Byte blocks
70     disk->fops = &bdops;
71
72     if (!(bdqueue = blk_init_queue(&bd_request, &bdlock)))
73         goto out;
74     blk_queue_logical_block_size(bdqueue, 512);
75     disk->queue = bdqueue;
76
77     add_disk(disk);
78     return 0;
79 out:
80     vfree(mempool);
81 out_no_mem:
82     unregister_blkdev(major, "bds");
83     return -EIO;
84 }
85
86 static void __exit mod_exit(void)
87 {
88     unregister_blkdev(major, "bds");
89     del_gendisk(disk);
90     put_disk(disk);
91     blk_cleanup_queue(bdqueue);
92     vfree(mempool);
93 }
94
95 module_init(mod_init);
96 module_exit(mod_exit);
97 MODULE_LICENSE("GPL");
```

Request abgearbeitet hat, muss er nämlich dem Blockgeräte-Subsystem Erfolg oder Misserfolg mitteilen. Dazu dienen die Funktionen »blk_end_request()«, »blk_end_request_all()«, »blk_end_request_cur()«, »__blk_end_request()«, »__blk_end_request_cur()« und »__blk_end_request_all()«.

Die drei letzten Funktionen gehen davon aus, dass sie die als ersten Parameter übergebene Request-Queue ohne weitere Sicherungsmaßnahmen modifizieren dürfen. Andersherum formuliert: Das zur Sicherung eingesetzte Spinlock hält bereits die aufrufende Funktion (so im Fall von Listing 1). Demgegenüber reservieren die ersten drei Funktionen ohne führenden Unterstrich explizit das Spinlock, bevor sie die Quittung in der Request-Queue verarbeiten. Der Versuch jedoch, ein bereits reserviertes Spinlock ein zweites Mal zu reservieren, führt per defintionem zu aktivem Stillstand.

Erfolgskontrolle

Ob er einen Auftrag erfolgreich bearbeitet hat oder nicht, übermittelt der Treiber dem Blockgeräte-Subsystem mit dem zweiten Parameter. Eine Rückgabe von 0 steht für Erfolg, ein anderer Wert repräsentiert einen Fehlercode. Die Antwort der Quittungsfunktion »blk_end_request_cur()« informiert darüber, ob der Treiber den Request komplettiert hat. Dann liefert er den Returncode 0. Sollte es noch einen weiteren Teilauftrag geben, teilt der Treiber dies ebenfalls mit. Dann liefern »blk_rq_pos()« und »blk_rq_cur_bytes()« die nächsten Transferparameter. Die Request-Funktion holt sich also einen Request, arbeitet ihn Segment für Segment ab und nimmt sich anschließend den nächsten vor, bis alle Aufträge erledigt sind.

Listing 1 zeigt den möglichen Aufbau einer Request-Funktion. Der Code realisiert einen einfachen Ramdisk-Treiber, der die Daten per »memcpy()« in den bei der Initialisierung per »vmalloc()« reservierten Speicherbereich ablegt. Da der Code keine wirkliche Disk, sondern Hauptspeicher adressiert, benötigt er die Adressenangaben nicht in Sektoren, sondern in Byte. Um aus einer Sektor- eine Byte-Adresse zu erhalten, multipliziert der Entwickler die Sektoranzahl mit der Sektorgröße, also mit 512 Byte. In der Binärarithmetik entspricht die Multiplikation mit 512 dem Linksschieben um 9 Bit (Listing 1, Zeile 25).

Während der Treiber-Initialisierung legt das Modul durch den Aufruf der Funktion »set_capacity()« die Größe des Hintergrundspeichers fest. Sie benötigt wiederum eine Angabe in Sektoren, die im Fall der Ramdisk als Byte-Adresse vorliegt; daher das Rechtsschieben um 9 Bit.

Die im Beispielcode vorkommende Struktur »struct block_device_operations« benötigen Entwickler übrigens, um einen Entladeschutz zu realisieren. Dazu setzen sie das Element »owner« auf »THIS_MODULE« und verankern die Struktur im Disk-Objekt.

Zurück marsch, marsch!

Last but not least gilt es aufzuräumen. Die Entladefunktion »__exit mod_exit()« des Treibers macht alles wieder rückgängig: Sie gibt mit »unregister_blkdev()« die Majornummer, die beiden reservierten Objekte »struct gendisk« und »struct request_queue« durch »del_gendisk()«, »put_disk()« sowie »blk_cleanup_queue()« wieder frei. Der Aufruf von »vfree()« führt den Hauptspeicher der Wiederverwertung zu.

Um den Treiber zu testen, kompilieren Experimentierfreudige den Code aus Listing 1 mit Hilfe des Makefile aus Listing 2, laden den resultierenden Treiber, erzeugen ein Filesystem und mounten dann das Gerät: Anschließend dürfen sie die Ramdisk verwenden. Die vom Treiber automatisiert angelegte Gerätedatei lautet »/dev/bds0«. Abbildung 3 zeigt die zur Anschauung gedachte Ramdisk. In realen Einsatzszenarien sollten Entwickler jedoch besser RAM-Filesysteme verwenden, da sie eine Reihe von Vorteilen bieten [3]. *(mg)* ■

Gerätenummern im Blockgerätetreiber

Obwohl der Linux-Kernel seit Jahren auf Gerätenummern setzt, reservieren Blockgerätetreiber noch immer die aus Unix-Urtagen stammenden Major- und Minornummern mit »register_blkdev()«. Sie tauchen tatsächlich weiterhin in den Kernel-internen Datenstrukturen auf, etwa in der »struct gendisk«.

Major- und Minornummern beschränken Linux mit ihren jeweils 8 Bit auf 255 Blockgerätetreiber (0 hat eine besondere Bedeutung), die jeweils 256 Disks oder Partitionen bedienen können. Für den privaten PC zu Hause reicht das allemal, für Server im Datacenter hingegen ist das manchmal zu wenig. Gerätenummern sind 32 Bit breit und ermöglichen dadurch den Anschluss von insgesamt vier Milliarden Disks oder Partitionen. Wie viele Treiber diese bedienen, spielt dabei keine Rolle.

Eine der ersten Aufgaben der Funktion »add_disk()« ist, die in der Disk-Struktur übergebenen Major- und Minornummern auf Gerätenummern abzubilden und den entsprechenden Bereich zu reservieren (siehe Abbildung 4). Dazu ruft die Funktion intern »blk_register_region()« auf.

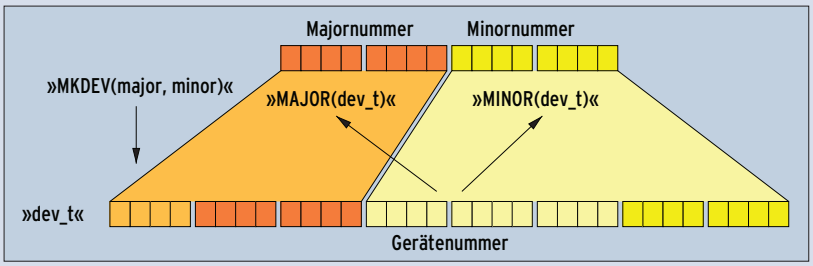


Abbildung 4: Linus Torvalds hat festgelegt, wie Entwickler die zugehörigen neuen Gerätenummern auf Grundlage der alten Major- und Minornummern finden, und bietet dazu eine Funktion an.

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 19: Linux-Magazin 03/05
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 20: Linux-Magazin 04/05
[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik": Folge 39, Linux-Magazin 05/08

Listing 2: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := ramdisk.o
03 else
04 KDIR    := /lib/modules/$(shell uname -r)/build
05 PWD     := $(shell pwd)
06
07 default:
08     $(MAKE) -C $(KDIR)  M=$(PWD) modules
09 endif
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 55

Kern-Technik

Mit ausgefeiltem Ressourcen-Management und eigenen Namensräumen organisiert der Kernel eine effiziente Container-Virtualisierung. Die Unterstützung im Userland könnte jedoch besser sein. Eva-Katharina Kunst, Jürgen Quade



Seit IT-Urzeiten ist eine der vornehmsten Aufgaben des Betriebssystems die Virtualisierung. Galt es damals vor allem, die CPU mit ihren Registern zu virtualisieren und damit das Multitasking beziehungsweise das Multithreading zu ermöglichen, ist heute die Virtualisierung des kompletten Rechners trendy. Festplatten, DVD-Laufwerke, Netzwerkschnittstellen, Grafikkarten und Hauptspeicher stehen dem Anwender scheinbar exklusiv zur Verfügung, real aber teilt er sich die Ressourcen mit den übrigen Usern.

Dank Virtualisierung lassen sich mehrere, auch unterschiedliche Betriebssysteme auf der Hardware gleichzeitig betreiben, ohne Umbooten zu müssen. Auf einer leistungsfähigen Hardware konsolidieren Unternehmen Server, was typischerweise den Wartungs- und Administrationsaufwand reduziert.

Das Hauptargument für die Virtualisierung ist aber meist eine erhöhte Sicherheit, die Systemverwalter zum einen durch die damit verbundene Isolierung der Systeme, zum anderen durch die reduzierte Komplexität erreichen. Sie handeln damit gemäß dem Motto: Eine Betriebssysteminstanz – ein Dienst.

Ins Gefängnis

Dabei gibt es verschiedene Ansätze: so genannte Container-Virtualisierung, Hypervisor-Technik und Hardware-Emulation (siehe Kasten „Virtualisierungstechniken“).

Die Variante, die dabei die geringsten Ressourcen benötigt, ist die Container-Virtualisierung (siehe Abbildung 1). Die Idee hinter dieser Technik ist, möglichst viele Ressourcen gemeinsam zu nutzen. So verwenden beispielsweise die virtualisierten Systeme denselben Betriebssystemkern und häufig auch dieselben Systemprogramme. Das spart Speicher und CPU-Zeit.

Diese Art der Virtualisierung ist – da sie sehr wenig zusätzliche Ressourcen benötigt – schon lange in Gebrauch, beispielsweise unter dem Namen Chroot. Allerdings virtualisiert Chroot im Wesentlichen nur das Root-Filesystem. Linus Torvalds hat dagegen in den letzten Jahren seinen Betriebssystemkern mit einer fast vollständigen Container-Virtualisierung ausgestattet. Dabei hat er zwei wesentliche Technologien integriert: Kontrollgruppen und Namensräume.

Mit Hilfe der Control Groups lassen sich Rechenprozesse einer virtuellen Umgebung, dem Container beziehungsweise auch nur einem Namensraum zuordnen. Sie binden aber auch die Jobs fest an einen bestimmten Prozessorkern einer Multicore-Maschine [1] oder limitieren vorhandene Speicher-Ressourcen. Aus dem Userland wird dies alles über ein virtuelles Filesystem gesteuert.

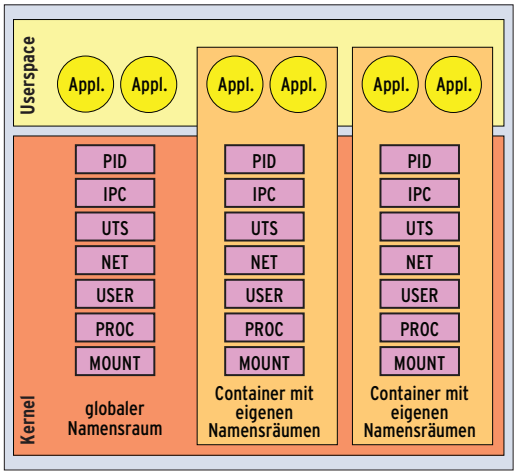
Täuschungsmanöver

Normalerweise sind die Datenstrukturen für den kompletten Kernel eindeutig, etwa sein Name oder die von ihm vergebenen Prozess-IDs. Die Namespaces isolieren die Betriebsmittel, indem sie es Anwendern erlauben, Bereiche einzurichten, in denen die Bezeichner eine neue Gültigkeit besitzen. Inzwischen sind sieben Namensräume realisiert, die Tabelle 1 auflistet.

Innerhalb eines »UTS«-Namespace vergibt der Linux-Anwender individuelle Domainnamen, Nodenamen, eine Release, eine Version und schließlich den Maschinennamen – kurzum die Informationen, die er sonst per Systemcall »uname()« ausliest. »USER« steht für eigene User-IDs, »NET« für virtuelle

Tabelle 1: Namensräume

Flag	Namensraum	Neu instanzierte Objekte
CLONE_NEWIPC	IPC	Messages, Mailboxes und Shared Memory
CLONE_NEWNET	NET	Netzwerkgeräte, Routen und Sockets
CLONE_NEWNS	MNT	Mountpoints (nur zum Lesen)
CLONE_NEWPID	PID, PROC	Prozess-IDs und zugehöriges Proc-Filesystem
CLONE_NEWUTS	UTS	Systemnamen und -informationen
CLONE_NEWUSER	USER	User-IDs



◀ **Abbildung 1:** LXC legt innerhalb von Containern eigene Namensräume an, mit denen Applikationen eine Reihe von Systemressourcen unabhängig vom restlichen Kernel benennen dürfen.

```
Datei  Bearbeiten  Ansicht  Verlauf  Lesezeichen  Einstellungen  Hilfe
quade@ezs-mobil:/tmp/clone$ make
cc -g -Wall    clone.c   -o clone
quade@ezs-mobil:/tmp/clone$ sudo ./clone
Eltern-Thread (Kind-PID 7055): eigene PID: 7054
Kind-Thread: eigene PID: 1
quade@ezs-mobil:/tmp/clone$
clone : bash
```

▲ **Abbildung 2:** Die Prozesse, die virtualisierte Jobs abarbeiten, haben zwei PIDs: eine im globalen Namensraum sowie die PID in der virtualisierten Umgebung. Der Kindprozess im Beispielprogramm hat aus Elternsicht die PID 7055. Das Kind selbst meint, es besitze die PID 1.

Netzwerkgeräte, Routen und Sockets. »PROC« erzeugt einen Container mit eigenem Proc-Filesystem. Da der Kernel Objekte der Interprozess-Kommunikation ebenfalls über Namen referenziert, stellt »IPC« einen zugehörigen Namensraum für Shared-Memory, Messages und Semaphore bereit. »MNT« schließlich instanziert unabhängige Mountpoints.

Dreh- und Angelpunkt, um Namensräume zu nutzen, ist der Systemcall »clone()«. Linux erzeugt damit neue Rechenprozesse. Den gewünschten Namensraum steuert der Code beim Erzeugen über Flags, die die Container-Virtualisierung in »linux/sched.h« bereitstellt.

Mit Root-Rechten ausgestattet ruft der Applikationsprogrammierer »clone()« auf und übergibt beispielsweise das Flag »CLONE_NEWPID«. Der Kernel erstellt daraufhin einen neuen Namensraum für den Kindprozess. Das Programm in Listing 1 verdeutlicht dies zusammen mit Abbildung 2. Der Elternprozess gibt die PID des erzeugten Kindes (7055) und die eigene (7054) aus. Der Kindprozess zeigt dagegen nur die eigene PID an.

Innenansichten

Aus Sicht des Kindprozesses ist diese 1, da der Aufruf einen neuen Namensraum erzeugt und der Kindprozess die erste freie ID bekommt. Im Hostsystem bekommt diese PID normalerweise der Init-Prozess. Das Beispiel verdeutlicht, dass der Kindprozess durch den neuen Namensraum zwei PIDs besitzt: einmal die PID im globalen Namensraum und einmal die PID im Container.

Um das selbst zu testen, kompilieren Experimentierfreudige den Code und starten das entstehende Programm als Superuser. Um zusätzlich andere Namensräume zu aktivieren, verknüpfen sie die »CLONE_NEW*Typ*«-Flags beim Aufruf von »clone()« per bitweisem ODER miteinander. Der Kernel repräsentiert die Namespaces durch jeweils eigene Datenstrukturen, die den zugehörigen Namensraum eindeutig beschreiben. Mit Einzug der Container-Virtualisierung hat Linus Torvalds den Prozess-Controllblock als beschreibende Instanz für einen Rechenprozess modifiziert (Abbildung 3).

Informationen wie beispielsweise die PID oder die UIDs sind nicht mehr direkt im Prozess-Kontrollblock (PCB) abgelegt, sondern nur noch über Zeiger auf die neuen Datenstrukturen erreichbar. Um Speicherplatz zu sparen, haben die Entwickler die Namensräume vom PCB aus jedoch nicht einzeln verlinkt, sondern über die dazwischen eingeschobene Datenstruktur »struct nsproxy« semantisch zusammengefasst. Ein einzelner Zeiger

Listing 1: Namensräume anlegen

```
01 #include <asm/unistd.h>
02 #include <stdio.h>
03 #include <sched.h>
04 #include <unistd.h>
05 #include <wait.h>
06
07 int main(int argc, char **argv, char **envp)
08 {
09     pid_t pid;
10     int status;
11
12     pid = syscall(__NR_clone, CLONE_NEWPID | SIGCHLD,
13                   0, 0, 0);
14     if (pid == 0) {
15         printf("Kind-Thread: ");
16     } else {
17         printf("Eltern-Thread (Kind-PID %d): ", pid);
18     }
19     printf("eigene PID: %d\n", getpid());
20     wait(&status);
21     return 0;
22 }
```

Virtualisierungstechniken

Auch wenn es so aussieht: Festplatten, Grafikkarten oder auch Prozessoren sind nicht immer real. Immer häufiger bekommen Betriebssysteme oder Applikationen virtuelle Hardware vorgesetzt. Die Virtualisierung unterscheidet das Host- vom Gastsystemen. Der Host kontrolliert real die Hardware. Er erzeugt die virtuelle Sicht, auf die seine Gäste bauen. Sie setzen hierbei drei Basistechnologien ein:

- Container-Virtualisierung: Das Betriebssystem schafft durch eigene Namensräume eine virtuelle Umgebung (Container). Gastsysteme nutzen den Betriebssystemkern des Hostsystems. Beispiel: Linux Container LXC, Open VZ.
- Hypervisor-Technik: Eine als Hypervisor bezeichnete Systemkomponente kontrolliert die eigentliche Hardware. Die Prozessorhersteller Intel und AMD unterstützen diese Art der Virtualisierung in modernen Prozessoren durch besondere Technologien (VT-x und Pacifica). Oft ist der Hypervisor mit dem Hostsystem verbandelt. Die Technik ermöglicht es, eine Hardware gleichzeitig von unterschiedlichen Betriebssystemen verwenden zu lassen. Beispiele: VMware, Virtualbox.
- Hardware-Emulation: Hier wird eine Hardware, ein Rechner emuliert. Sehr vorteilhaft ist, dass die Architektur des emulierten Rechners (Gastsystem) sich komplett von der Architektur des Hostsystems unterscheiden kann. Dem stehen jedoch eine niedrige Arbeitsgeschwindigkeit des Gastsystems und eine hohe Auslastung des Hostsystems gegenüber. Beispiel: Qemu.

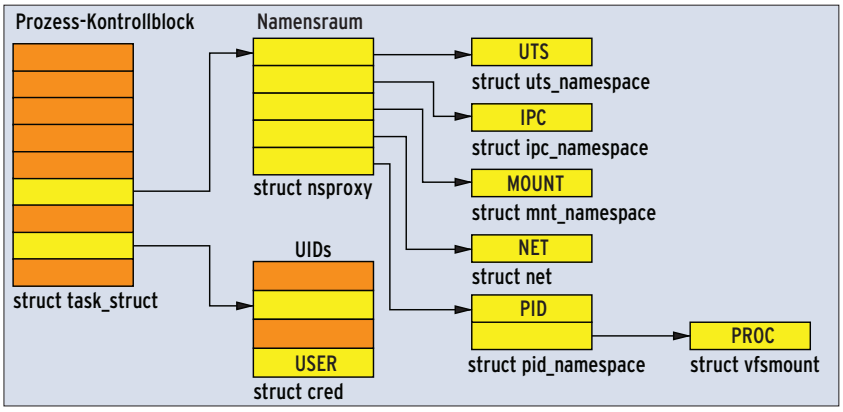


Abbildung 3: Der Prozess-Kontrollblock enthält Zeiger auf die zugehörigen Namensräume.

(8 Byte auf einem 64-Bit-System) gibt damit Zugriff auf sechs der sieben Namensräume. Neben der Speicherplatzersparnis ist das schnelle Kopieren des PCB ein weiterer Vorteil. Der Kern legt den Namensraum für eigene UIDs in der Datenstruktur »struct cred« (Credentials) ab und verlinkt sie direkt im PCB. Informationen zum virtualisierten Proc-Filesystem finden Entwickler über die »struct pid_namespace«.

Hineingeboren

Beim Systemstart instanziert der Kernel für jeden Namensraum ein Objekt. Der so geschaffene globale Namensraum ist über die Variablen »init_pid_ns«, »init_user_ns«, »init_ipc_ns« und »init_net« für jede Kernelkomponente erreichbar. Die Threads im System referenzieren ihn.

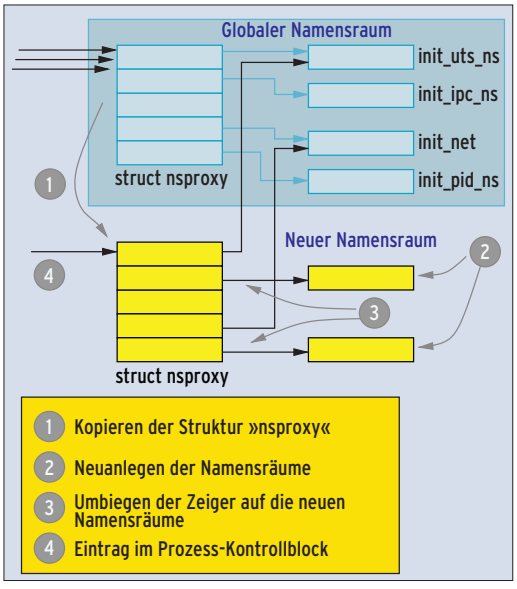


Abbildung 4: Der Kernel legt neue Namensräume effizient an, indem er vorzugsweise Zeiger kopiert.

Auch neu erzeugte Threads – erben sie doch die Umgebung vom Eltern-Thread – verwenden den globalen Namensraum. Ruft jedoch ein Thread den Systemcall »clone()« auf und setzt eines der Clone-Flags (mit Ausnahme von »CLONE_NEWUSER«), erzeugt dies ein neues Objekt vom Typ »struct nsproxy« und füllt es zunächst mit dem alten Inhalt durch Kopieren der Zeiger. Je nach Flag generiert Linux dann neue Namensräume und verbiegt die Zeiger in »nsproxy« auf sie (siehe Abbildung 4).

Torvalds hat auch die Systemaufrufe modifiziert: Sie greifen jetzt über »nsproxy« ausschließlich auf die virtuelle Sicht zu. Im Kernel zählt jedoch überwiegend der globale Namensraum, wenngleich interne Kernel-Schnittstellen – mit einem zusätzlichen Namensraum-Parameter ausgestattet – die Objekt-Adressierung innerhalb eines Containers ermöglichen.

Darüber hinaus hat der Kernel viele Detailänderungen erfahren, um die Container-Virtualisierung praxisreif zu machen. Ein Zeiger im Superblock des Proc-Filesystems etwa referenziert den PID-Namensraum, abgelegt im Element »s_sf_info«. Da es bei der Darstellung von Netzwerkinterfaces im Sys-Filesystem zu Problemen kam, haben die Entwickler mit Kernel 2.6.36 ein so genanntes Sysfs-Tagging eingeführt [2].

Um mittels Namensräumen und Cgroups zu isolieren und damit zu virtualisieren, benötigen Anwender die LXC-Userspace-Tools aus dem Debian-Paket »lxc«. Sie sorgen dafür, die Umgebung vorzubereiten und den Container so zu konfigurieren, dass darin Programme ablaufen [3]. Die wichtigsten Kommandos sind »lxc_checkconfig«, »lxc_execute«, »lxc_start« und »lxc_console«.

Die Werkzeug-Suite unterscheidet Anwendungs- und Systemcontainer. Den ersten dürfen Anwender ohne weitere Installation direkt verwenden. Er führt Kommandos in einem isolierten Container aus. Der zweite virtualisiert hingegen komplette Systeme. Wer ihn einsetzt, muss nicht nur ein rudimentäres System installieren, sondern zugleich noch dafür sorgen, dass die benötigten Dienste starten. Anders als bei der Hypervisor-Technik bootet ein Systemcontainer nämlich nicht klassisch – schließlich ist der verwendete Betriebssystemkern ja bereits aktiv. Die dazu notwendigen Administrationskenntnisse vermittelt der LXC-Workshop in diesem Heft.

Mit Sicherheit

Die LXC-Container-Virtualisierung basiert auf Linux-Namensräumen. So arbeitet der Kernel Rechenprozesse isoliert ab. Seine Schnittstellen, nämlich Zusatzflags beim Aufruf des Systemcalls »clone()« und das Cgroup-Filesystem, sind übersichtlich. Der Kernel hat damit einiges für erhöhte Sicherheit bei kaum gestiegenem Ressourcenbedarf getan. Im Userland warten Anwender dagegen noch auf feiner geschliffene Werkzeuge, um die Virtualisierung zu vereinfachen. *(mg)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 41: Linux-Magazin 09/08, S. 88

[2] Eric Biederman, „Sysfs-Tagging“: Kernel-Dokumentation: [http://lxr.free-electrons.com/source/Documentation/filesystems/sysfs-tagging.txt]

[3] Linux-Container LXC: [http://lxc.sf.net]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Demnächst erscheint die dritte Auflage ihres Buches „Linux Treiber entwickeln“.

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 56

Kern-Technik

Das Rahmenwerk DKMS ordnet für Anwender und Entwickler Module so, dass sie auch nach einem Kernupdate noch laufen. Als Bonus schnürt das Tool sogar Installationspakete für Distributionen. Eva-Katharina Kunst, Jürgen Quade



Ein Update hat das System zerschossen: Plötzlich funktionieren Drucker, Scanner und andere Geräte nicht mehr, da sie eigene Treiber und Module benötigen. Bringt ein aktualisierter Kernel sie nicht von Haus aus mit, ist die betroffene Peripherie erst einmal nutzlos. Bei Linux sind der Kernel und die Module eins zu eins aufeinander abgestimmt. Passen sie nicht zueinander, lassen sich die Treiber nicht laden, geschweige denn verwenden.

Knigge für die Struktur von Modul-Quellcodes

Als Ausweg aus diesem Schlamassel hatte der PC-Hersteller Dell bereits 2003 für eigene Kunden und als Beitrag zur Linux-Entwicklung den Dynamic Kernel Module Support (DKMS) konzipiert [1]. Einmal installiert wird DKMS bei jedem Update des Kernels aktiv. Das Werkzeug kümmert sich um außerhalb des normalen Kernel-Quellcodes gepflegte Treiber: Sobald der Admin die Treiber angemeldet hat, übersetzt und installiert DKMS sie bei jeder Aktualisierung des Kerns.

Dazu müssen jedoch die Treiber- und Kernelmodul-Programmierer DKMS explizit unterstützen. Zum Glück fällt dies Entwicklern leicht, denn es nützt ihnen, dass die Software als Beigabe automatisch Deb- und RPM-Installationspakete baut.

Um den eigenen Quellcode DKMS anzuvertrauen, muss der Entwickler einige Konventionen einhalten und eine Konfigurationsdatei verfassen. So erwartet DKMS den Quellcode eines Moduls in einem Unterverzeichnis von »/usr/src/«. Sein Name setzt sich aus Paket und Versionsnummer zusammen.

DKMS organisiert Module nämlich in Paketen, die eine eindeutige Version haben. Ein Paket darf dabei auch mehrere Module beinhalten. Enthält ein Paket nur ein Modul, legt der Entwickler seinen Quellcode direkt in dem neuen Ordner ab. Ansonsten erhält jedes Modul ein eigenes Unterverzeichnis.

Prominentes Beispiel dafür ist die Virtualisierungslösung Virtual Box: Das Paket nennt sich »vboxhost« und seine Version von Anfang Februar ist »4.0.0«. Damit lautet das Paketverzeichnis »/usr/src/vboxhost-4.0.0/«. Da Virtual Box für den Betrieb den Basistreiber »vboxdrv«, den Treiber für die virtuelle Netzwerkkarte »vboxnetadp« sowie einen für den Netzfilter »vboxnetflt« mitbringt, befinden sich unterhalb des Paketverzeichnisses drei Ordner mit diesen Namen (siehe Abbildung 1).

Pflegehinweise beachten

Auch wenn ein Paket mehrere Module hat, erwartet DKMS pro Paket genau eine Konfigurationsdatei. Sie trägt den Namen »dkms.conf« und befindet sich direkt im Paketverzeichnis. Einstellungen macht der Systemverwalter in Form von Variablenzuweisungen – das als Bash-Skript verfasste Tool verleibt sich die Datei per »source« ein. Die Modulverwaltung erfordert mindestens Angaben zum Paketnamen, zur Version und zu den Modulnamen (siehe Listing 1). Die durch »yes« aktivierte Variable »AUTO-

Name	Größe	Dateityp	Verknüpfung
vboxhost-4.0.0	4,0 KB	Ordner	
alloc	4,0 KB	Ordner	
common	4,0 KB	Ordner	
generic	4,0 KB	Ordner	
include	4,0 KB	Ordner	
math	4,0 KB	Ordner	
VBox	4,0 KB	Ordner	
vboxdrv	4,0 KB	Ordner	
alloc	4,0 KB	Ordner	../alloc
common	4,0 KB	Ordner	../common
generic	4,0 KB	Ordner	../generic
include	4,0 KB	Ordner	../include
VBox	4,0 KB	Ordner	../VBox
Makefile	8,5 KB	Makefile	
SUPDrvAgnostic.c	19,8 KB	C-Quellcodedatei	
SUPDrv.c	191,7 KB	C-Quellcodedatei	
SUPDrvIDC.h	8,6 KB	C-Headerdatei	
SUPDrvInternal.h	22,3 KB	C-Headerdatei	
SUPDrvIOC.h	36,6 KB	C-Headerdatei	
vboxnetadp	4,0 KB	Ordner	
vboxnetflt	4,0 KB	Ordner	
dkms.conf	407 B	PAK-Dateiarchiv	

Abbildung 1: Um mit DKMS ein Modul zu verwalten, legt der Admin unter »/usr/src« eine Verzeichnis- und Dateistruktur an und richtet sie in »dkms.conf« ein.

Hinter den Kulissen

Ein 4000-Zeilen-Bash-Skript bildet das Rückgrat von DKMS. Es legt Kopien der Originaldateien an, überprüft, ob die Voraussetzungen zum Generieren eines Moduls erfüllt sind, übersetzt es für mehrere Kernelversionen, kopiert das Ergebnis in die zugehörigen Verzeichnisse und lädt das Modul – falls gewünscht – in den aktuell laufenden Betriebssystemkern. DKMS erzeugt auch initiale Ramdisks [2]. Es passt die Konfigurationsdateien für das Laden der Module in »/etc/modprobe.d/« an, falls sie Parameter benötigen oder Anwender sie per Alias ansprechen sollen.

Neben den modulspezifischen Konfigurationen findet DKMS im Ordner »/etc/dkms/« in der Datei »framework.conf« seine globale Konfiguration vor. Zu ihr gehören die Pfade zum Quellcode, wohin das Werkzeug die erzeugten Module speichern soll und wo temporäre Dateien residieren. Sämtliche Konfigurationen, sowohl die modulspezifischen als auch die globalen, bindet DKMS über Shellvariablen ein. Im Verzeichnis »/etc/dkms/« liegen außerdem Template-Dateien, die die Paketierung benötigt. Typischerweise muss der Entwickler sie jedoch nicht anfassen.

Automatisch in die Aktualisierung einbinden

Neben dem eigentlichen Tool »dkms« gehört noch das Skript »dkms_autoinstaller« zum Projekt. Ein Dreizeiler im Verzeichnis »/etc/kernel/postinst.d/« startet es auf einem Ubuntu-System. Das Skript wiederum durchsucht per »find« alle unter »/var/lib/dkms/« liegenden Konfigurationen mit dem Namen »dkms.conf«. Steht in einer der Konfigurationsdateien die Variable »AUTOINSTALL« auf »yes«, ruft der automatische Installateur die Kommandos »dkms build« und »dkms install« auf.

DKMS arbeitet die skizzierten Kommandos nicht plump ab, sondern passt sie je nach Distribution an und kennt dazu RPM-basierte Systeme wie Red Hat, Fedora oder Open Suse genauso wie Debian-Derivate. Ein dediziertes Fehlermanagement sorgt für den reibungslosen Betrieb: Dazu gehört, dass das Tool beispielsweise vorhandene Module gleichen Namens sichert und später bei Bedarf wieder zurückspielt.

INSTALLER« sorgt letztlich dafür, dass DKMS beim nächsten Kernelupdate alle betroffenen Module passend übersetzt und installiert.

Ubuntu kennt das praktische Subsystem in seinem Repository als Paket »dkms«, das ein gleichnamiges Tool installiert. Es beherrscht insgesamt 13 Kommandos, im Alltag benötigen Entwickler und Admins aber nur die acht Befehle aus Abbildung 2 (siehe Kasten „Hinter den Kulissen“). Getreu dem Grundprinzip, Linux möglichst unangetastet zu lassen und Änderungen rückgängig machen zu können, übernimmt DKMS seine zu verwaltenden Module in eine eigene Datenbasis, die es unterhalb von »/var/lib/dkms/« speichert.

Der Aufruf von »dkms add« pflegt ein neues Modul in die Datenbasis ein. Dazu legt das Tool ein Directory für das neue Paket und anschließend ein Unterverzeichnis für die zugehörige Version an. Ein symbolischer Link verweist auf das Quellcode-Verzeichnis mit der Datei »dkms.conf«. Weil es Paketname und Version aufspaltet, verwaltet DKMS übersichtlich mehrere Releases der gleichen Komponente.

Eigene Module hinzufügen

Wer unterhalb des Ordners »/usr/src/« ein neues Verzeichnis mit beispielsweise dem Namen »linuxmagazin-20110113« anlegt und darin den Quellcode eines Moduls (Listing 1), ein passendes Makefile (Listing 2) sowie schließlich die DKMS-Konfiguration (siehe Listing 3 und Kasten „DKMS konfigurieren“) ablegt, kann die Wirkungsweise leicht verfolgen (Abbildung 3). Der Aufruf

```
dkms add -m linuxmagazin -v 20110113
```

erzeugt den neuen Ordner »/var/lib/dkms/linuxmagazin/« mit dem Namen des Pakets und der Version als Unterverzeichnis »20110113«.

Um das Modul zu erzeugen, ruft der Entwickler das Werkzeug mit »dkms build« und den Optionen »-m« für Modulname sowie »-v« für Version auf:

```
dkms build -m linuxmagazin -v 20110113
```

Das Skript kopiert den unter »/usr/src/linuxmagazin-20110113/« liegenden Quellcode ins Verzeichnis »/var/lib/dkms/linuxmagazin/20110113/build/« und ruft dort »make« auf, um die Module zu übersetzen. Der ursprüngliche Quellcode

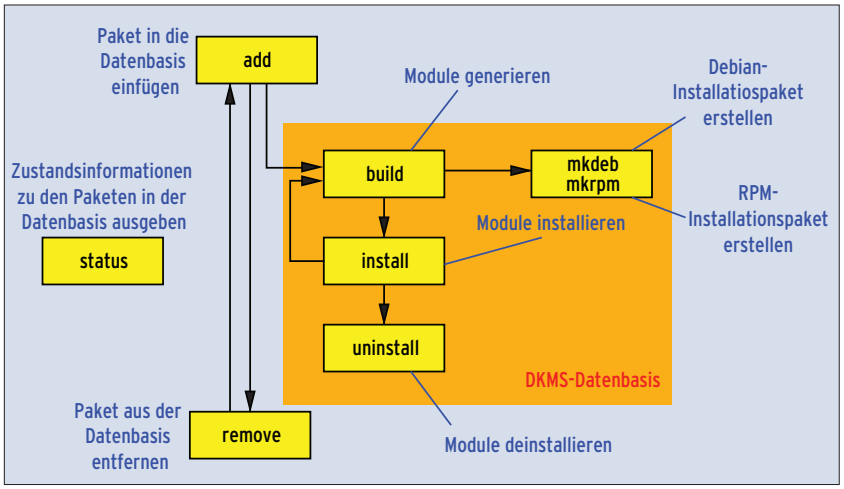


Abbildung 2: DKMS kennt viele Subkommandos. Mit »add« macht der Admin ein Paket bekannt, mit »build« baut er es für den lokalen Gebrauch und aktiviert es mit »install«. Weitere Befehle machen diese Schritte rückgängig, informieren über den Status oder bauen als Bonus Installationspakete für andere Rechner.

Listing 1: Demo-Modul für DKMS

```
01 /* Dummy-Modul erklärt Funktionsweise von DKMS: */
02 #include <linux/module.h>
03
04 static int __init lm_init(void)
05 {
06     printk("Linux-Magazin Modulinitialisierung\n");
07     return 0;
08 }
09
10 static void __exit lm_exit(void)
11 {
12     printk("Linux-Magazin Deinitialisierung\n");
13     return;
14 }
15
16 module_init(lm_init);
17 module_exit(lm_exit);
18 MODULE_LICENSE("GPL");
```

bleibt dabei unverändert. DKMS protokolliert alle Ausgaben des Buildvorgangs in »make.log«, sodass ein Admin später die Ursache von Fehlern erkunden kann. Das fertige Produkt legt DKMS in

```
/var/lib/dkms/linuxmagazin/20110113/➥
Kernelversion/Kernelarchitektur/module/
```

ab – repräsentiert durch die Variable »BUILT_MODULE_NAME« in der Konfigurationsdatei »dkms.conf«. Die Installation an die üblichen Stellen für den Produktivbetrieb gibt der Systemverantwortliche durch den Befehl

```
dkms install -m linuxmagazin -v 20110113
```

in Auftrag. Das Werkzeug prüft zunächst, ob es bereits ein Modul mit gleichem Namen gibt. Ist das der Fall, überschreibt es das gleichnamige Modul nicht einfach, sondern sichert es zuvor. Entfernt der Systemverwalter ein Modul wieder, kopiert »dkms« den Originalcode zurück. Die Software legt die generierten Module übrigens im Verzeichnis »/lib/modules/*Kernelversion*/updates/« ab.
Hat er das Modul erfolgreich installiert, lädt es der Anwender mit

```
modprobe linuxmagazin
```

in den Kernel. Entwickler freut, dass sie auch Module für einen anderen als den gerade laufenden Kernel erzeugen dürfen. Dazu reicht es, mit der Option »-k« die gewünschte Kernelversion anzugeben – die zugehörigen Header und Konfigurationen müssen auf dem System unter »/usr/src/linux-2.6.*x*« und »/usr/src/linux-headers-2.6.*x-y*« installiert sein.
Nach dem erstmaligen Einpflegen übernimmt das Skript »dkms_autoinstaller« die Kontrolle über DKMS. Der distributionsspezifische Updatemechanismus ruft es auf. Der Auto-Installer führt fortan die Subkommandos »build« und »install« in der beschriebenen Weise mit jedem neuen Kernelupdate automatisiert und ohne Benutzerinteraktion aus.

Listing 2: Zugehöriges Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m    := linuxmagazin.o
03 else
04 KDIR     := /lib/modules/$(shell uname -r)/build
05 PWD      := $(shell pwd)
06
07 default:
08         $(MAKE) -C $(KDIR)       M=$(PWD) modules
09 endif
10
11 clean:
12         rm -f *.o *.ko  .*.d .*.cmd *.mod.c
13         rm -f modules.order Module.symvers
14         rm -rf .tmp_versions
```

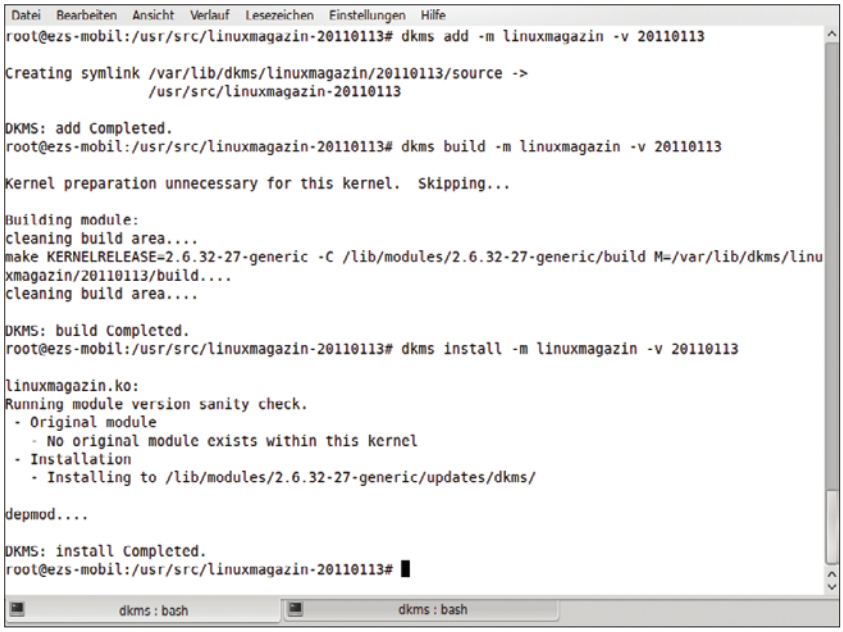


Abbildung 3: Damit DKMS ein Treiberpaket unter seine Fittiche nimmt, fügt es der Admin erst mit »add« dem lokalen Repository hinzu und baut es mit »build«. Die notwendigen Teilschritte erledigt das Werkzeug selbstständig.

Änderungsschneiderei

Das klappt natürlich nur, solange es keine großen Modifikationen an den verwendeten Kernelschnittstellen gibt. Überarbeitet Linus Torvalds sie jedoch, benötigen Anwender zwangsläufig eine im Quellcode angepasste Version des Moduls. Diese automatisiert im Internet zu suchen und einzuspielen, beherrscht DKMS zurzeit allerdings nicht.
Wer in Handarbeit die aktualisierte und zum neuen Kernel passende Version besorgt hat, deinstalliert oder entfernt zunächst die bisher verwendete Variante. Dazu bietet »dkms« die Kommandos »uninstall« und »remove« an. Remove ist das

Tabelle 1: Konfigurationsvariablen

Variable	Bedeutung
PACKAGE_NAME, PACKAGE_VERSION	Name und Version des Pakets sind eine Pflichtangabe. Sie ergeben sich aus dem Pfad »/usr/src/*Paket-Version*/«.
BUILT_MODULE_NAME[*n*]	Name des Moduls. Soll ein Paket mehrere Module enthalten, nummeriert der Entwickler die einzelnen Treiber beginnend mit 0.
BUILT_MODULE_LOCATION[*n*]	Ordner, in dem DKMS das Modul nach dem Übersetzen ablegt. Verzeichnisse geben Admins relativ zum Wurzelverzeichnis des Quellcode an.
REMAKE_INITRD	Wer die Variable mit »yes« aktiviert, lässt DKMS die zum Kernel gehörende initiale Ramdisk neu generieren.
PATCH[*p*]	Entwickler belegen die Variable mit einer Liste von Dateinamen, die eine Reihe von Patches widerspiegelt. Jedes Patch wendet DKMS per »patch -p1 *Patchdatei*« an. Die zugehörigen Patchdateien liegen im Unterverzeichnis »patches«.
AUTOINSTALL	Wer die Variable mit »yes« aktiviert, lässt DKMS ein generiertes Modul automatisch bei einem Kernelupdate installieren.
PRE_BUILD	Name eines Skripts, das DKMS vor dem Kommando »build« abarbeitet.
PRE_INSTALL	Name eines Skripts, das DKMS vor dem Kommando »install« abarbeitet.
POST_ADD	Name eines Skripts, das DKMS nach dem Kommando »add« abarbeitet.
POST_BUILD	Name eines Skripts, das DKMS nach dem Kommando »build« abarbeitet.
POST_INSTALL	Name eines Skripts, das DKMS nach dem Kommando »install« abarbeitet.
POST_REMOVE	Name eines Skripts, das DKMS nach dem Kommando »remove« abarbeitet.

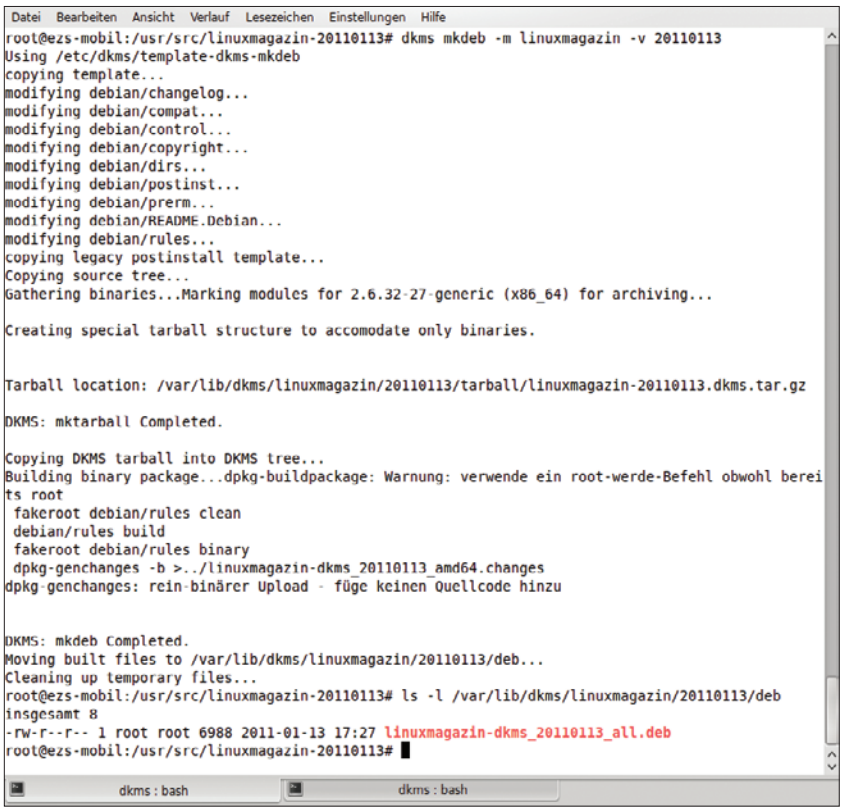


Abbildung 4: DKMS reicht ein einziges Kommando aus, um beispielsweise ein Debian-Paket zu schnüren. Eine weitere Funktion baut auch RPM-Pakete – praktisch für Modulanbieter, um viele Anwender anzusprechen.

Gegenstück zu »add«. DKMS löscht damit den kompletten Dateibaum für das Modul unterhalb von »/var/lib/dkms/«, der Code unter »/usr/src/« bleibt erhalten:

```
dkms remove -m linuxmagazin -v 20110113
  -k 2.6.32-26-generic
```

Danach gibt der Admin per »add« die neue Version in die Obhut von DKMS. Die folgenden »build« und »install« generieren und installieren das Modul.

DKMS ist ziemlich flexibel und beherrscht noch deutlich mehr als die dargestellten Basisfunktionen: So lassen sich Modulparameter in den Konfigurationsdateien unterhalb von »/etc/modprobe.d/« ebenso einrichten wie Aliase. Das Werkzeug generiert Module für ganz bestimmte Kernelversionen und legt auf Wunsch eine initiale Ramdisk an. Sind Patches in der Konfiguration aufgeführt, wendet DKMS sie auf den Quellcode an. An wichtigen Stellen im Buildprozess lassen sich außerdem externe Kommandos ausführen.

DKMS konfigurieren

Um ihre Module besser zu ordnen, konfigurieren Systemverwalter DKMS-Pakete durch das Zuweisen von Shellvariablen. Die Datei »dkms.conf« im Hauptverzeichnis des betroffenen Pakets nimmt dank Variablenfeldern auf Wunsch auch Angaben für mehrere Module auf einmal auf, wenn sie alle zu einem Paket gehören. Dann versehen Admins die vorbestimmten Variablen – beginnend bei 0 – mit einer laufenden Nummer. Tabelle 1 stellt die wichtigsten Attribute vor.

Alles einpacken

Die Unterkommandos »mkdeb«, »mkdsc« und »mkrpm« erleichtern dem Entwickler die Arbeit erheblich, denn sie schnüren Installationspakete für Debian- und Red-Hat-basierte Linux-Varianten. Dazu braucht der Entwickler im Idealfall nicht einmal etwas zu konfigurieren. Auf dem Rechner benötigt er natürlich die notwendige Software für das Paketebauen, bei Ubuntu beispielsweise das Paket »deb-helper«. Der Aufruf

```
dkms mkdeb -m linuxmagazin -v 20110113
 --source-only
```

erzeugt ein Debian-Paket mit dem Namen »linuxmagazin-dkms_20110113_all.deb« und legt es im Verzeichnis »/var/lib/dkms/linuxmagazin/20110113/deb/« ab (siehe Abbildung 4). Das »all« im Namen bedeutet, dass alle Kernelversionen das Paket akzeptieren. Die Option »source-only« ist nicht unbedingt notwendig, stellt aber sicher, dass nur der Modul-Quellcode, nicht aber das bereits generierte Modul in das Debian-Paket kommt. Dadurch generiert DKMS das Modul auf jeder Plattform neu. Natürlich muss auf jedem Zielsystem ebenfalls das Paket »dkms« installiert sein. Glücklicherweise ist jedes erzeugte Paket bereits von »dkms« abhängig, sodass der Paketmanager automatisch alles Notwendige auf dem Zielsystem installiert.

Module ordentlich im Griff

Der Dynamic Kernel Module Support löst das Problem, den Kernel und extern gepflegte Module auf gleichem Stand zu halten. Modul-Anbieter für Nvidia, Virtualbox oder Batman, das Routingprotokoll der Freifunk-Szene, nutzen das Verfahren bereits. Trotzdem ist es bei vielen Entwicklern noch unbekannt. Admins, Anwender und Entwickler profitieren davon, wenn sich das ändert. (*mg*)

Infos

[1] Dynamic Kernel Module Support (DKMS): [http://linux.dell.com/dkms/]
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik: Initramfs“, Folge 39: Linux-Magazin 05/08, S. 98

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Demnächst erscheint die dritte Auflage ihres Buches „Linux Treiber entwickeln“.

Listing 3: Konfigurationsdatei »dkms.conf«

```
01 # dkms.conf
02 PACKAGE_NAME="linuxmagazin"
03 PACKAGE_VERSION=20110113
04 AUTOINSTALL=yes
05
06 BUILT_MODULE_NAME[0]="linuxmagazin"
07
08 # Distributionen werten die nachfolgende Variable
09 # in aktuellen Versionen nicht mehr aus. Ubuntu
10 # installiert das Modul unter
11 # /lib/modules/Kernelversion/updates/
11 DEST_MODULE_LOCATION[0]="/updates"
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 57

Kern-Technik

Texte mit AES verschlüsseln, Daten komprimieren oder Hashsummen mit MD5 berechnen: Mit dem Crypto-API des Linux-Kernels und etwas Know-how kein Problem! Jürgen Quade, Eva-Katharina Kunst



Die umfassende Internetnutzung und das Bedürfnis nach Sicherheit sorgen für stärkere Verbreitung der Kryptographie. Hardwarehersteller bauen aus diesem Grund seit einigen Jahren entsprechende Erweiterungen in die hauseigenen Chips ein, beispielsweise VIA mit Padlock [1] oder Intel mit dem AES-NI-Befehlssatz [2]. Linux wiederum stellt dem Programmierer mit dem Crypto-API Kryptographie-Routinen zur Verfügung, die entweder rein in Software oder aber Hardware-unterstützt ablaufen. Ziel der Kernelentwickler ist es, dies für den Entwickler transparent zu halten.

Das Crypto-API ist nicht nur für Verschlüsselung zuständig, sondern auch für das Bilden von Hash- und Checksummen, für Kompression oder die Generierung von Zufallszahlen. Durch den modularen Aufbau des Crypto-Layers lassen sich auf der unteren Seite die unterschiedlichsten Algorithmen andocken (siehe Tabelle 1), die sich auf der oberen Seite von verschiedenen Subsystemen wie beispielsweise der Festplattenverschlüsselung (DM-Crypt) oder der geschützten VPN-Kommunikation (IPsec) nutzen lassen (siehe Abbildung 1).

Transform-Objekte

Basis des Crypto-API im Kernel sind die Transform-Objekte. Für die Bildung von Hashsummen sowie Kompressions- und Verschlüsselungsverfahren gibt es jeweils eigene Objekte. Wenn es um Verschlüsselung geht, hat der Entwickler sogar die Auswahl zwischen vier unterschiedlichen Transforms: Cipher, Blockcipher, Asynchronous Blockcipher und AEAD (Authenticated Encryption with Associated Data). Diese unterscheiden sich bezüglich der Angabe von Quell- und Zielspeicherbereichen und der Funktionalität.

Verstreute Daten

Block-orientierte Transform-Objekte, wozu neben Blockcipher, Asynchronous Blockcipher und AEAD auch Hashsummen gehören, spezifizieren Quelle und Ziel der Operationen auf Basis von Scatter-Gather-Listen (siehe Kasten „Scatter-Gather-Listen“). Stream-orientierte Transforms (Cipher, Compress) verwenden direkt virtuelle Adressen.

Generell arbeiten die Algorithmen synchron, mit dem »ablkcipher«-Transform ist auch eine asynchrone Ver- und Entschlüsselung möglich: Der Crypto-Layer übernimmt Aufträge und übergibt das Ergebnis später per Callback-Funktion

Scatter-Gather-Listen

Scatter-Gather-Listen tauchen im Kernel immer dann auf, wenn größere Mengen an Daten zu transferieren sind. Größere Datenmengen – also typischerweise mehr Daten, als in eine so genannte Page (4096 Byte) passen – verteilen sich nämlich im Hauptspeicher häufig über verschiedene physische Adressen.

Scatter-Gather-Listen fassen diese physischen Adressen in einer Datenstruktur zusammen, sodass die Transferfunktionen – etwa das Schreiben der Daten auf die Festplatte – in einem Zug alle notwendigen Informationen besitzen und den Transfer auch per DMA durchführen können. Eigentlich wäre Scatter-Gather-Feld die korrekte Bezeichnung, denn die grundlegende Datenstruktur ist ein Array vom Typ »struct scatterlist«. Jedes Element eines solchen Felds repräsentiert ein zusammenhängendes Fragment im Speicher, das durch eine Page-Adresse, den Startoffset innerhalb der Speicherseite sowie die Länge der dort abgelegten Daten (in Bytes) gekennzeichnet ist.

Ein derartiges Feld kann der Programmierer entweder dynamisch während der Laufzeit allozieren und initialisieren oder dies statisch durch den Compiler erledigen lassen. Im letzteren Fall muss der Code zur Initialisierung »sg_init_table()« aufrufen. Um die einzelnen Feldelemente zu belegen, kommt entweder »sg_set_buf()« oder »sg_set_page()« zum Aufruf – je nachdem, ob die Adresse der Daten als virtuelle oder als Page-Adresse vorliegt. Wurde das Scatter-Gather-Feld per »sg_alloc_table()« angelegt, gibt es »sg_free_table()« nach Gebrauch wieder frei.

Neben den in Tabelle 2 aufgeführten Funktionen gibt es noch eine Reihe weiterer, die insbesondere die Abarbeitung einer Scatter-Gather-Liste vereinfachen.

dem Auftraggeber. Das AEAD-Transform schließlich kombiniert Verschlüsselung und Hashing, wie es bei IPsec benötigt wird. Vor der Nutzung einer Kryptofunktion im Kernel steht das Anlegen eines geeigneten Transform-Objekts. Dabei ist gleichzeitig der gewünschte Algorithmus anzugeben, beispielsweise »md5«. Vereinfacht ausgedrückt ist hiernach die gewünschte Kryptofunktion mit Übergabe der Quell- und Zielspeicheradressen aufzurufen.

Dass sich der Programmierer im Detail jedoch noch mit Scatter-Gather-Listen, einem Descriptor-Objekt und so etwas wie »digestsize« herumschlagen muss, verdeutlicht dieser Artikel an den Beispielen MD5-Hashsummenbildung und AES-Verschlüsselung.

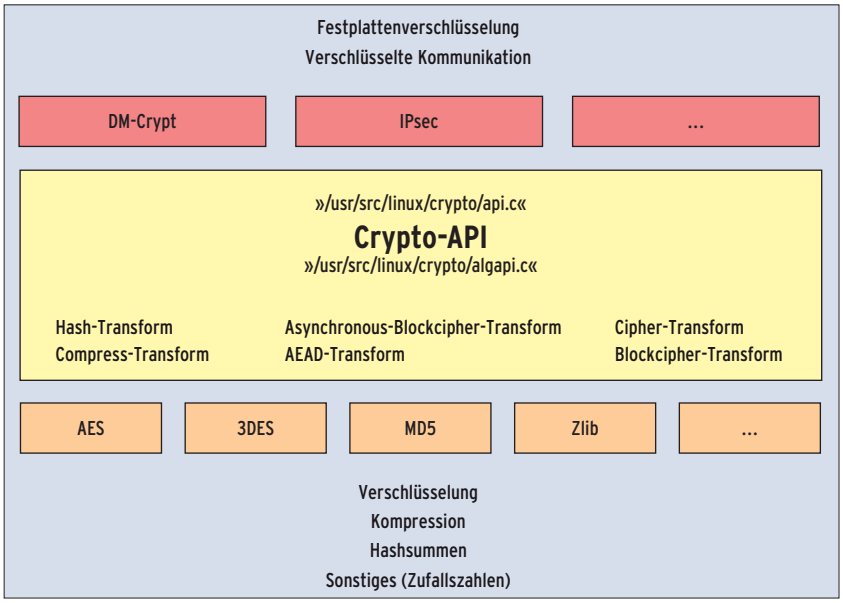


Abbildung 1: Der Crypto-Layer des Linux-Kernels abstrahiert Verschlüsselungsalgorithmen.

Hash berechnen

Mit »digestsize« dimensioniert der Entwickler die Größe des Speicherbereichs, der das Ergebnis der Hashsummenbildung aufnimmt, den Hashwert. Dieser ist – abhängig vom gewählten Hashalgorithmus – unterschiedlich groß und lässt sich über die Inline-Funktion »crypto_hash_digestsize()« abfragen. Das erwähnte Descriptor-Objekt dient als Container für die verschiedenen Kryptofunktionen. Der Programmierer initialisiert es mit der Adresse des Transform-Objekts.

Die Scatter-Gather-Listen schließlich enthalten die Adressen der Speicherbereiche. Sie bilden den aus Sicht einer Applikation zusammenhängenden Speicherbereich auf die physischen, realen Speicherorte ab, also die meist verstreut liegenden zugehörigen Pages.

Tabelle 1: Unterstützte Krypto-Algorithmen

Funktion	Algorithmen
Hashsummen	md4, md5, michael.mic, ripemd-128, ripemd-160, ripemd-256, ripemd-320, sha1, sha256, sha512, tiger, vmac, whirlpool
Cipher	3des, aead, aes, anubis, arc4, blowfish, camellia, cast5, cast6, des, fcrypt, khazad, salsa20, seed, serpent, tea, xtea, xeta, twofish
Kompression	deflate, lzo, zlib
Sonstiges	ansi_cprng (Pseudo Random Nummer Generator), kernel random number generator, random number generator, crc32c (Checksummen)

Listing 1 zeigt die Bildung der MD5-Summe des kurzen Datensatzes „Linux-Magazin“ im Kernel. Zeile 27 reserviert das Transform-Objekt unter Angabe des gewünschten Hashsummen-Algorithmus (»md5«), Zeile 33 initialisiert das Descriptor-Objekt mit der Adresse des Transform. Die Vorbereitung der Scatter-Gather-Listen ist in den Zeilen 24 und 25 zu sehen, und schließlich wird in Zeile 36 der Hashwert gebildet.

Dass dieser korrekt ist, lässt sich leicht – wie in Abbildung 2 dargestellt – überprüfen, indem man den Hashwert mit dem »md5sum«-Kommando erzeugt und mit der Ausgabe des Kernels im Syslog vergleicht. Der negative Rückgabewert (Zeile 49) signalisiert dem Kernel übrigens, dass er das Modul direkt wieder entladen soll. Das erspart beim Testen das ansonsten notwendige manuelle Entladen.

Verschlüsseln

Daneben bietet das Crypto-API des Kernels auch Verfahren zur Verschlüsselung an. Hier ist ein kleiner Ausflug in die Kryptographie [3] angebracht: Bei der Verschlüsselung wird eine Klartextnachricht mit Hilfe von Verschlüsselungsalgorithmen – so genannte Cipher – und unter Zuhilfenahme eines geheimen Schlüssels in den Ciphertext gewandelt. Unter der Voraussetzung, dass sowohl Algorithmus als auch der eingesetzte Schlüssel als

```
Datei  Bearbeiten  Ansicht  Verlauf  Lesezeichen  Einstellungen  Hilfe
root@ezs-mobil:/tmp/treiber# make
make    -C /lib/modules/2.6.32-30-generic/build M=/tmp/treiber modules
make[1]: Betrete Verzeichnis '/usr/src/linux-headers-2.6.32-30-generic'
  CC [M]  /tmp/treiber/lm.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /tmp/treiber/lm.mod.o
  LD [M]  /tmp/treiber/lm.ko
make[1]: Verlasse Verzeichnis '/usr/src/linux-headers-2.6.32-30-generic'
root@ezs-mobil:/tmp/treiber# insmod lm.ko
insmod: error inserting 'lm.ko': -1 Input/output error
root@ezs-mobil:/tmp/treiber# tail -2 /var/log/messages
Apr  2 18:14:40 ezs-mobil kernel: [32550.825944] Ergebnis md5sum:
Apr  2 18:14:40 ezs-mobil kernel: [32550.825951] 88b1f16f5bb4b923b5d352f09da9eb0d
root@ezs-mobil:/tmp/treiber# echo -n "Linux-Magazin" | md5sum
88b1f16f5bb4b923b5d352f09da9eb0d  -
root@ezs-mobil:/tmp/treiber#
quade : bash
```

Abbildung 2: Die Gegenprobe mit »md5sum« beweist: Der Code aus Listing 1 – zum Kernelmodul »lm.ko« kompiliert – berechnet beim Laden den MD5-Hashwert.

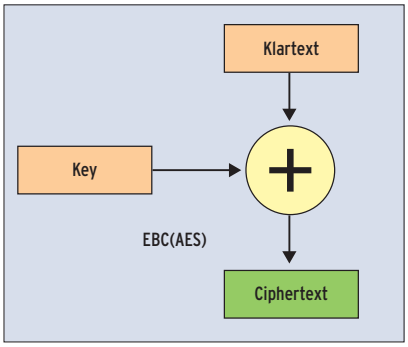


Abbildung 3a: Das Verfahren Electronic Code Book Mode (EBC) gilt als nur beschränkt sicher.

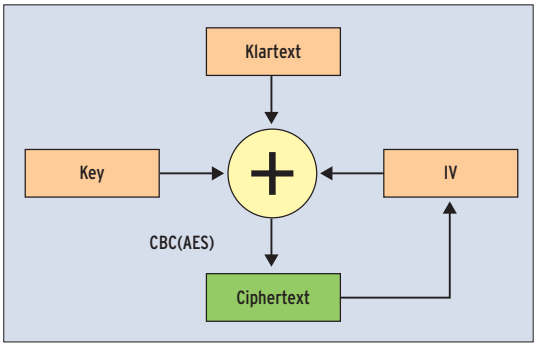


Abbildung 3b: Verkettung: Cipher Block Chaining (CBC) ist sicher, aber nicht für Multicore-Computer geeignet.

sicher gelten, kann nur derjenige den Ciphertext zurückwandeln (entschlüsseln) und interpretieren, der im Besitz des passenden Schlüssels ist.

Prinzipiell sind zwei Arten der Verschlüsselung zu unterscheiden: die symmetrische Verschlüsselung von der asymmetrischen. Die asymmetrische (Public-Key-Verfahren) setzt zur Ver- und Entschlüsselung unterschiedliche Schlüssel ein. Vorteil: Eine Verschlüsselung ist gezielt für einen dedizierten Empfänger beziehungsweise Empfängerkreis möglich. Die symmetrische Verschlüsselung demgegenüber setzt für beide Vorgänge denselben Key ein.

Die sichere Übergabe des geheimen Schlüssels und der Umstand, dass mehrere Personen oder Systeme den (geheimen) Schlüssel kennen, ist aus Sicht der Kryptographen bedenklich und erfordert besondere Sicherungsmaßnahmen. Andererseits ermöglichen symmetrische Verfahren aber eine sehr schnelle Kodierung und Dekodierung.

Die Cipher selbst lassen sich nach ihrer internen Arbeitsweise unterscheiden: Streamcipher verschlüsseln Nachrichten beliebiger Länge und typischerweise Zeichen für Zeichen, während Blockcipher Nachrichten blockweise verarbeiten. Nachrichten, die nicht dem Vielfachen einer Blöckgröße entsprechen, werden mit Nullen aufgefüllt.

Sicher oder nicht sicher

Falls der Algorithmus jeden Block unabhängig bearbeitet, spricht man vom Electronic Code Book Mode (EBC). Da dieses Verfahren für mehrfach vorkommende identische Blöcke den gleichen Ciphertext erstellt, gilt das Verfahren nur als eingeschränkt sicher (siehe Abbildung 3a). Das Cipher Block Chaining (CBC) vermengt den jeweils vorangehenden Block mit dem zu verschlüsselnden, zum Beispiel per XOR, wobei ein als Initialisierungsvektor (IV) bezeichneter Block für die Verknüpfung mit dem allerersten Block zuständig ist (Abbildung 3b).

Der IV wird typischerweise mit Zufallszahlen vorbelegt und braucht nicht geheim gehalten zu werden. CBC hat aber gerade in Zeiten von Multicore-Systemen einen entscheidenden Nachteil: Die Verschlüsselung ist nur sequenziell, also Block für Block möglich, nicht parallel.

CTR macht Tempo

Diesen Nachteil überwindet der Counter Mode (CTR): Er verschlüsselt jeden Block mit einem Initialisierungsvektor (Abbildung 3c). Der Inhalt des für jeden Block unterschiedlichen IV ergibt sich aus einem einmal festgelegten (Zufalls-)Wert und der Blocknummer durch sinnvolle mathematische Verknüpfung

Listing 1: MD5-Summe des Textes „Linux-Magazin“ (»lm.c«)

```
01 #include <linux/crypto.h>
02 #include <linux/err.h>
03 #include <linux/scatterlist.h>
04
05 static struct scatterlist sg_text;
06 static char result[128];
07 static struct crypto_hash *tfm;
08 static struct hash_desc desc;
09
10 static int __init mod_init( void )
11 {
12         struct page *pg;
13         char *buf;
14         int i;
15
16         pg = alloc_pages( GFP_USER, 0 );
17         if (pg==NULL) {
18                 printk("alloc_pages failed\n");
19                 return -EIO;
20         }
21         buf = page_address( pg );
22         memcpy( buf, "Linux-Magazin", 13 );
23
24         sg_init_table( &sg_text, 1 );
25         sg_set_page( &sg_text, pg, 13, 0 );
26
27         tfm = crypto_alloc_hash("md5", 0,
   CRYPTO_ALG_ASYNC);
28         if (IS_ERR(tfm)) {
29                 printk("crypto_alloc_hash
   failed\n");
30                 goto out_free_pages;
31         }
32
33         desc.tfm = tfm;
34         desc.flags = 0;
35
36         if (crypto_hash_digest(&desc,
    &sg_text, 13, result)) {
37                 printk("crypto_hash_digest
   failed\n");
38         } else {
39                 printk("Ergebnis md5sum:\n");
40                 for (i=0; i <
   crypto_hash_digestsize(tfm); i++ ) {
41                         printk("%02x",
   (unsigned char) result[i]);
42                 }
43                 printk("\n");
44         }
45
46         crypto_free_hash(tfm);
47 out_free_pages:
48         free_pages( (unsigned long)
    page_address(pg), 0 );
49         return -EIO;
50 }
51
52 static void __exit mod_exit( void )
53 {
54         return;
55 }
56
57 module_init( mod_init );
58 module_exit( mod_exit );
59 MODULE_LICENSE("GPL");
```

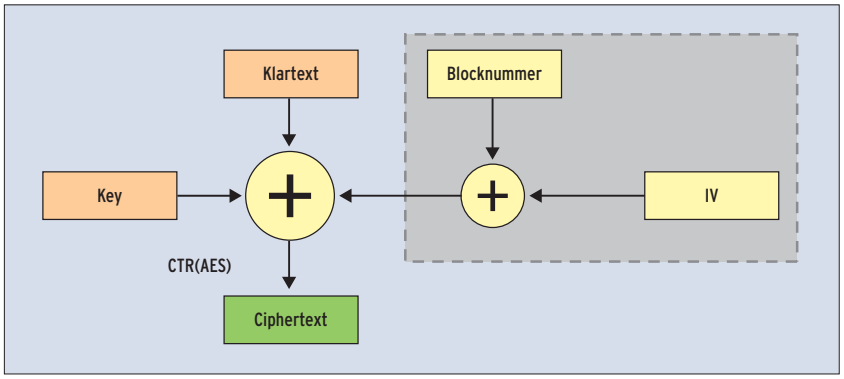


Abbildung 3c: Das Beste zweier Welten: Der Counter Mode (CTR) schafft es, durch die Verknüpfung jedes Blocks mit einem eigenen IV zugleich sicher und parallelisierbar zu sein.

(zum Beispiel XOR). Das Verschlüsseln von Daten mit dem Linux-Kernel verwendet eines der Cipher-Transforms aus dem Crypto-API. Für die weitverbreitete AES-Verschlüsselung eignet sich beispielsweise das Blockcipher-Transform. Die Verschlüsselung transformiert eine Klartextnachricht mit Hilfe des Schlüssels und mit einem Initialisierungsvektor (bei CBC und CTR) in den Ciphertext.

Erforderliche Parameter

Den Speicherbereich, der den Schlüssel enthält, übergibt der Programmierer zusammen mit der zugehörigen Schlüssellänge der Funktion »crypto_blkcipher_setkey()«. Den Speicherbereich für den Initialisierungsvektor stellt das Blockcipher-Transform-Objekt selbst zur Verfügung (siehe Abbildung 4). Er muss damit nur noch belegt werden.

In Form einer Scatter-Gather-List werden noch zwei weitere Speicherbereiche benötigt: Der erste enthält die Klartextnachricht, der zweite nimmt die verschlüsselte Nachricht auf, den Ciphertext. Mit Klartextnachricht, Key, Initialisierungsvektor und Speicher für das Ergebnis startet durch Aufruf der Funktion »crypto_blkcipher_encrypt()« die Verschlüsselung. Am Ende ist der Ergebnispuffer mit dem Ciphertext gefüllt. Das Entschlüsseln funktioniert genauso: Der Programmierer muss Ciphertext, Key, IV und Speicher für das Ergebnis – in diesem Fall die Klartextnachricht – bereitstellen.

Listing 2 zeigt die Ver- und Entschlüsselung des Textes „Linux Magazin“ mit AES im Blockchaining-Modus (»cbc«). Der Einfachheit halber kommt als Key und ebenso als IV eine Folge von Nullen zum Einsatz. AES ist ein Blockcipher, verschlüsselt also immer nur Daten, die ein Vielfaches eines Blocks (256 Bit) lang sind. Ist die Nachricht etwas kürzer, füllt man sie einfach mit Nullen auf.

Der Code zeigt im Syslog (auf vielen Systemen die Datei »/var/log/messages«) die Klartextnachricht (»mesg«), den Ciphertext (»encr«) und die entschlüsselte Nachricht (»decr«) an (siehe Abbildung 5). Es versteht sich von selbst, dass ein Entwickler beim Praxiseinsatz den Key intelligenter wählt, besonders schützt und nicht für jedermann sichtbar im Quellcode hinterlegt.

Keine asymmetrische Verschlüsselung

Die übrigen vier, hier nicht weiter vorgestellten Transform-Objekte arbeiten ähnlich. Leider gibt es insgesamt nur sehr wenig Dokumentation. Im Wesentlichen existiert nur die Textdatei in [4]. Der Programmierer ist praktisch gezwungen, sich die zugehörigen Funktionsnamen

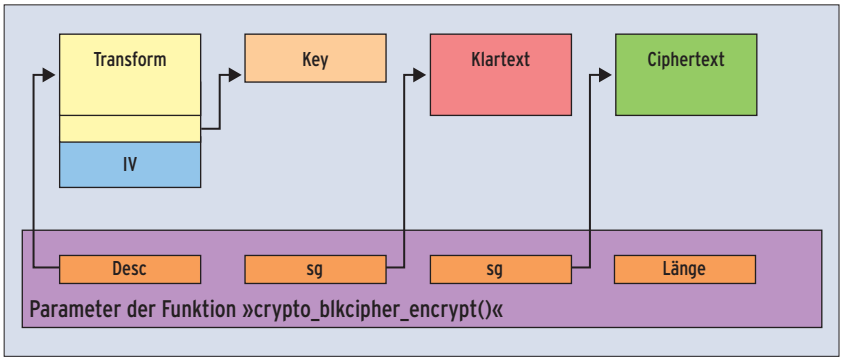


Abbildung 4: Zur Verschlüsselung notwendige Datenstrukturen: der Name des Transform-Objekts, der Schlüssel, die Speicherorte von Klar- und Ciphertext als Scatter-Gather-Listen (»sg«) sowie die Länge.

Tabelle 2: Funktionen zum Aufbau von Scatter-Gather-Listen

Funktionsprototyp	Kurzbeschreibung
int sg_alloc_table(struct sg_table *table, unsigned int nents, gfp_t gfp_mask)	Reserviert und initialisiert eine Scatter-Gather-Liste mit »nents« Feldern. »gfp_mask« gibt den Kontext an, in dem die Funktion abläuft (User-, Kernel- oder Interrupt-Kontext), »table« enthält nach dem Aufruf die Adresse der Liste.
void sg_init_table(struct scatterlist *sg, unsigned int nents)	Initialisiert die Liste »sg«, die »nents« Einträge hat.
void sg_init_one(struct scatterlist *sg, const void *buf, unsigned int length)	Initialisiert die Liste »sg« (die aus einem einzigen Listenelement besteht) und belegt sie mit der Adresse des Speicherbereichs »buf« und der Länge »length« vor.
static inline void sg_set_page(struct scatterlist *sg, struct page *page, unsigned int len, unsigned int offset)	Initialisiert das Listenelement »sg« mit der Seitenadresse »page«, der Länge »len« und dem Offset »offset« innerhalb der Seite.
static inline void sg_set_buf(struct scatterlist *sg, const void *buf, unsigned int buflen)	Initialisiert das Listenelement »sg« mit der virtuellen Adresse »buf« und der Länge »buflen«.
static inline void *sg_virt(struct scatterlist *sg)	Gibt die virtuelle Adresse des Listenelements »sg« zurück.

inklusive der erforderlichen Parameter aus dem Quelltext der Crypto-Headerdatei [5] rauszusuchen.

Hinzu kommt, dass das Crypto-API des Linux-Kernels Gegenstand aktiver Weiterentwicklung ist. Das Ziel von Maintainer Herbert Xu ist es, eine universelle Programmierschnittstelle zu schaffen, die – analog zu »memcpy« – Daten transferiert und dabei falls notwendig auch transformiert. Dank des zurzeit leider sehr spärlich genutzten asynchronen Interface lässt sich dabei auch Parallelverarbeitung auf Multicore-Maschinen nutzen.

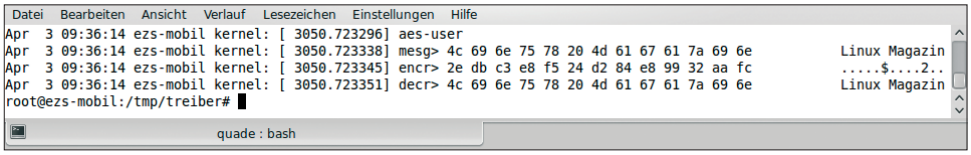


Abbildung 5: Blick ins Syslog: Die Ausgabe des Moduls »aes_user.ko« – aus dem Code in Listing 2 kompiliert – zeigt den Klartext, den Ciphertext und schließlich die entschlüsselte Nachricht.

Zukunftsvision

Außerdem befindet sich eine Progamm-ierschnittstelle zum Userspace in Entwicklung, sodass auch normale Linux-Programme bequem und transparent Verschlüsselung einsetzen können, und das nach Möglichkeit mit Hardware-Unterstützung. Sobald auch noch die asymmetrischen Verschlüsselungsverfahren hinreichend Umsetzung finden, ist Linux im Bereich Kryptographie bestens ausgestattet. *(mhu)*

Infos

[1] Wikipedia, Padlock Security Engine: [http://de.wikipedia.org/wiki/PadLock]
[2] Wikipedia, AES instruction set: [http://en.wikipedia.org/wiki/AES_instruction_set]
[3] Wikipedia, Kryptographie: [http://de.wikipedia.org/wiki/Kryptografie]
[4] Herbert Xu, „Scatterlist Cryptographic API“: [http://lxr.free-electrons.com/source/Documentation/crypto/api-intro.txt]
[5] Headerdatei »linux/crypto.h«: [http://lxr.free-electrons.com/source/include/linux/crypto.h]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Mittlerweile ist die dritte Auflage ihres Buches „Linux Treiber entwickeln“ erschienen.

Listing 2: Verschlüsseln und Entschlüsseln im Kernel (»aes_user.c«)

```
01 #include <linux/crypto.h>
02 #include <linux/scatterlist.h>
03
04 #define DATA_SIZE  32
05
06 static int __init aes_user_init(void)
07 {
08         char key[16], *iv;
09         struct crypto_blkcipher *tfm_cbc;
10         struct scatterlist sg[3];
11         int ret;
12         char *msg, *encrypted, *decrypted;
13         struct blkcipher_desc desc;
14         size_t ivsize;
15
16         printk("aes-user\n");
17         memset(key, 0, sizeof(key));
18
19         tfm_cbc = crypto_alloc_blkcipher
   ("cbc(aes)",0,CRYPTO_ALG_ASYNC);
20         if ( IS_ERR(tfm_cbc) )
21                 return -EIO;
22
23         desc.tfm = tfm_cbc;
24         desc.flags = 0;
25
26         ret = crypto_blkcipher_setkey
   (tfm_cbc, key, sizeof(key));
27         if ( ret<0 )
28                 goto out;
29
30         msg = kzalloc(DATA_SIZE, GFP_KERNEL);
31         if ( !msg )
32                 goto out;
33
34         encrypted = kzalloc(DATA_SIZE,
    GFP_KERNEL);
35         if ( !encrypted )
36                 goto alloc_encrypted_failed;
37
38         decrypted = kzalloc(DATA_SIZE,
    GFP_KERNEL);
39         if ( !decrypted )
40                 goto alloc_decrypted_failed;
41
42         memcpy(msg, "Linux Magazin", 13 );
43
44         sg_init_table( sg, ARRAY_SIZE(sg) );
45         sg_set_buf( &sg[0], msg, DATA_SIZE );
46         sg_set_buf( &sg[1], encrypted,
    DATA_SIZE );
47         sg_set_buf( &sg[2], decrypted,
    DATA_SIZE );
48
49         iv = crypto_blkcipher_crt(tfm_cbc)
   ->iv;
50         ivsize = crypto_blkcipher_ivsize
   (tfm_cbc);
51         memset(iv, 0, ivsize);
52
53         ret = crypto_blkcipher_encrypt(&desc,
   &sg[1], &sg[0], DATA_SIZE);
54         if ( ret<0 )
55                 goto crypt_failed;
56
57         iv = crypto_blkcipher_crt(tfm_cbc)
   ->iv;
58         ivsize = crypto_blkcipher_ivsize
   (tfm_cbc);
59         memset(iv, 0, ivsize);
60
61         ret = crypto_blkcipher_decrypt(&desc,
   &sg[2], &sg[1], DATA_SIZE);
62         if ( ret<0 )
63                 goto crypt_failed;
64
65         print_hex_dump(KERN_INFO,"mesg> ",
   DUMP_PREFIX_NONE,16,1,msg,13,1);
66         print_hex_dump(KERN_INFO,"encr> ",
   DUMP_PREFIX_NONE,16,1,encrypted,13,1);
67         print_hex_dump(KERN_INFO,"decr> ",
   DUMP_PREFIX_NONE,16,1,decrypted,13,1);
68
69 crypt_failed:
70         kfree(decrypted);
71 alloc_decrypted_failed:
72         kfree(encrypted);
73 alloc_encrypted_failed:
74         kfree(msg);
75 out:
76         crypto_free_blkcipher(tfm_cbc);
77         return -EIO;
78 }
79
80 static void __exit aes_user_exit(void)
81 {
82 }
83
84 module_init(aes_user_init);
85 module_exit(aes_user_exit);
86 MODULE_LICENSE("GPL");
```

Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 58

Kern-Technik

Der Linux-Kernel bietet seine Verschlüsselungsfunktionen auch zum asynchronen Zugriff an. Ineinander geschachtelte Datenstrukturen und undokumentierte Funktionen machen dem Programmierer allerdings das Leben schwer. Diese Kern-Technik schafft Abhilfe. Jürgen Quade, Eva-Katharina Kunst



Wenn die Kernelentwickler von Datentransformation sprechen, meinen sie damit das Ver- und Entschlüsseln, Komprimieren oder das Bilden von Hashsummen. Geeignete Interfaces stellen den Zugriff auf die dafür implementierten Algorithmen sicher, die ihrerseits über definierte Schnittstellen an das Crypto-Subsystem des Linux-Kernels andocken.

Synchron – asynchron

Bisher finden die Transformationen vorwiegend synchron statt, wie es die vorige Ausgabe der Kern-Technik beschrieben hat [1]: Ein User, beispielsweise die Festplattenverschlüsselung, übergibt einen Verschlüsselungsauftrag an das Crypto-Subsystem und wartet bis zum Abschluss der Transformation. Erst dann erteilt er den nächsten Auftrag. Hardware-Verschlüsselung oder Parallelverarbeitung kommen dabei aber nicht zum Zug. Performanter geht es mit dem asynchronen Zugriff, wie ihn auch die Diplomarbeit von Sebastian Siewior [2] vorstellt.

Asynchron bedeutet, dass der User dem Crypto-Subsystem einen oder mehrere Aufträge zum Bearbeiten übergibt und die erfolgreiche oder vergebliche Auftragsbearbeitung zu einem späteren Zeitpunkt prüft (Abbildung 1). Der angeforderte Algorithmus wiederum muss die Aufträge in die Warteschlange einreihen und starten. Die Nachricht über Ge- oder Misslingen teilt das Crypto-Subsystem dem User später mit, indem es eine Callback-Funktion aufruft.

In der Theorie mag das recht einfach erscheinen, in der Praxis erweisen sich ineinander geschachtelte Datenstrukturen und nicht dokumentierte Funktionen als Stolpersteine. Grundwissen und Quellcodestudium helfen weiter.

Dabei ist für jene, die performant verschlüsseln möchten, der erste Schritt noch einfach: Er spezifiziert den gewünschten Algorithmus und reserviert mit dieser Information das Transform-Objekt vom Typ »ablkcipher_tfm«. Existiert für den Algorithmus keine asynchrone Implementierung, gibt die Reservierungsfunktion einen Wert ungleich 0 zurück. Im anderen Fall wird der zu verwendende Schlüssel an das Transform-Objekt gebunden (Abbildung 2, (1)).

Der nachfolgende Schritt bereitet die Transformations-Aufträge und Übergabe an das Crypto-Subsystem vor. Dazu reserviert der User ein Request-Objekt und initialisiert es mit den Adressen der Quell- und Zielspeicherbereiche (als Scatter-Gather-Listen, siehe dazu [1]) und einer Callback-Funktion (Abbildung 2, (2)). Diese spielt eine zentrale Rolle für den dritten Block, der das Durchführen des Auftrags überwacht (3).

Hierbei fällt auf, dass das Crypto-Subsystem einen Auftrag manchmal auch synchron abarbeitet. Ob der Auftrag direkt abgearbeitet (synchron) oder zur Abarbeitung in die Queue gestellt wurde, ist am Rückgabewert der Funktion zur Auftragsvergabe ersichtlich. Im ersten Fall gibt die Funktion 0 zurück, sonst »-EINPROGRESS« oder »-EBUSY«.

Im Fall der asynchronen Abarbeitung ruft das Crypto-Subsystem nach dem Bearbeiten die Callback-Funktion auf und übergibt ihr die Adresse des Auftragsobjekts. Liegen die Ergebnisse sämtlicher Aufträge vor, beginnt im vierten Schritt das Aufräumen: die Freigabe aller Auftragsobjekte, aber auch des Transform-Objekts (Abbildung 2, (4)).

Entschlüsseln

Listing 1 zeigt den Einsatz der asynchronen Schnittstelle. Das Beispiel dekodiert eine Geheimnachricht (Ciphertext), die per XOR-Verfahren mit dem Key »Mein Schluessel.« verschlüsselt wurde. Um den Code möglichst übersichtlich zu halten, beschränkt sich die normalerweise aufwändige Verwaltung der Request-Objekte (»struct ablkcipher_request«) auf die Reservierung und Freigabe eines einzelnen Objekts. Zum Entschlüsseln einer Nachricht, die kürzer als 16 Byte ist, reicht das aus.

Das Ende der Auftragsbearbeitung lässt sich am besten mit Hilfe des Completion-Objekts verfolgen. Nachdem er die Aufträge dem Crypto-Subsystem zum

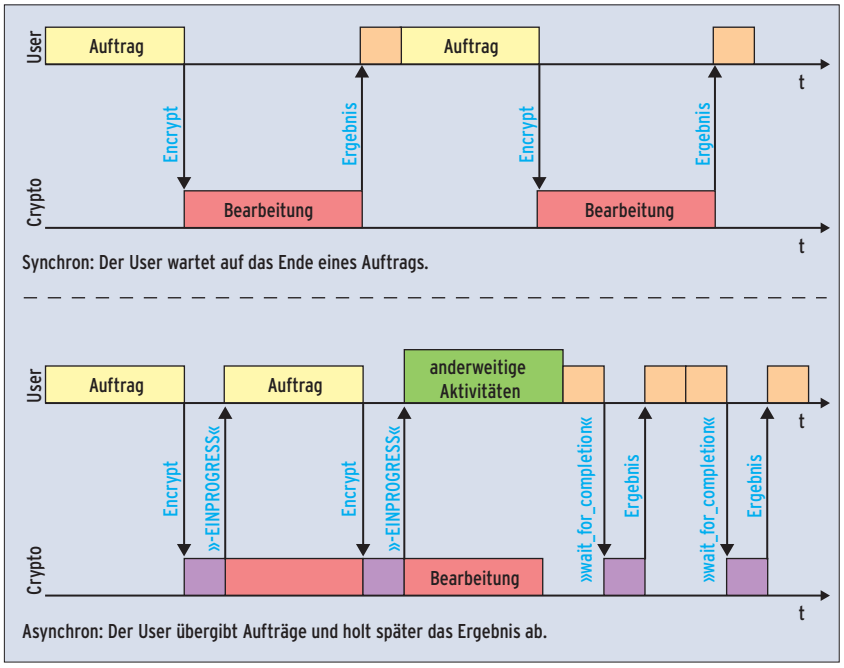


Abbildung 1: Der synchrone Zugriff (oben) auf das Crypto-API des Kernels bringt Wartezeiten mit sich. Bessere Performance verspricht die asynchrone Variante (unten).

Bearbeiten übergeben hat, ruft der User für jeden Auftrag einmal die Funktion »wait_for-completion()« auf (Abbildung 1). Die legt die Instanz schlafen, bis das korrespondierende »complete()« erfolgt, bis also die Callback-Funktion zum Ende des Auftrags aktiviert wurde.

Da sein Code neben dem Completion-Objekt typischerweise auch noch den Fehlerstatus eines Auftrags verwalten muss, deklariert der Programmierer für all diese Daten eine eigene Datenstruktur (Listing 1, Zeile 10) und instanziert sie für jeden Auftrag. Der Beispielcode reserviert zur Vereinfachung Speicher für diese Datenstruktur auf dem Stack (Zeile 33).

Callback

Etwas komplexer gerät der Code dadurch, dass ein asynchroner Auftrag auch synchron, also direkt bearbeitet werden kann. In diesem Fall kommt es nie zum Aufruf der Callback-Funktion durch das Crypto-Subsystem; daher wird die Callback-Funktion stellvertretend aktiviert (Listing 1, Zeile 73). Innerhalb der Callback-Funktion selbst werden der Fehlercode gesetzt und das Completion-Objekt aktiviert. Die Callback-Funktion muss im Übrigen so programmiert sein, dass sie sich in jedem Kontext aufrufen lässt. Innerhalb dieser Funktion ist es daher beispielsweise nicht möglich, den gerade aktiven Rechenprozess in den Schlafzustand zu versetzen.

Listing 2 zeigt, wie der Programmierer eigene Algorithmen an das Crypto-Subsystem des Kernels andockt. Im Kern beschreibt es ein Objekt vom Typ »struct crypto_alg«. Dieses Objekt gibt es für jede Transformation (Cipher, Blockcipher, Asynchronous Blockcipher, AEAD) in einer eigenen Ausprägung. Welche vorliegt, spezifiziert ein Flag im Attribut »cra_flags«. Die je nach Ausprägung unterschiedlichen Attribute realisiert der Programmierer über das Konstrukt des Union »cra_u«.

Obligatorisch ist es, für jeden Algorithmus einen Namen und eine Priorität zu vergeben. Namen dürfen im Übrigen mehrfach vorkommen, wenn sie auch real den gleichen Algorithmus umsetzen. Das Crypto-Subsystem ermöglicht nämlich die Koexistenz verschiedener Implementierungen. Ist ein Algorithmus mehrfach umgesetzt, wählt das Subsystem die Implementierung mit der höchsten Priorität. Normalerweise besitzen Algorithmen die Priorität 100, besonders optimierte Algorithmen – etwa durch eine Assembler-Kodierung – haben die Priorität 200, Hardware-unterstützte Implementierungen 300 oder 400 (Abbildung 3).

Objekte und Parameter

Sobald ein User sich beim Crypto-Subsystem unter Angabe des Algorithmen-Namens anmeldet, wird vom Algorithmen-Objekt ein Transform-Objekt abgeleitet. Dieses Objekt enthält nicht nur die Adressen der eigentlichen Transformierungsfunktionen (wie beispielsweise Verschlüsseln), sondern auch die notwendigen instanzenspezifischen Parameter. Möchte ein User beispielsweise Daten ver- oder entschlüsseln, speichert das Objekt den zugehörigen Schlüssel als so genannten Kontext ab.

Weil der Kontext nicht nur algorithmenspezifisch, sondern auch implementie-

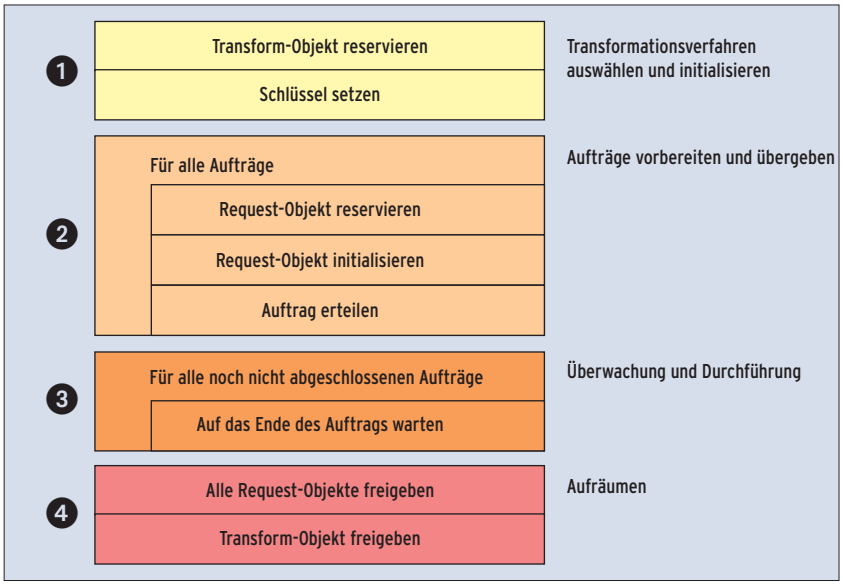


Abbildung 2: Softwarestruktur eines im Kernel asynchron zugreifenden Users.

```
quade@ezs-mobil:/tmp/treiber$ cat /proc/crypto
name         : ablk
driver       : ablk-generic
module       : kernel
priority     : 100
refcnt       : 1
selftest     : passed
type         : ablkcipher
async        : yes
blocksize    : 16
min keysize  : 0
max keysize  : 16
ivsize       : 0
geniv        : <default>

name         : cbc(aes)
driver       : cbc(aes-asm)
module       : kernel
priority     : 200
refcnt       : 752
selftest     : passed
type         : blkcipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : <default>

name         : ecb(aes)
driver       : ecb(aes-asm)
module       : kernel
priority     : 200
refcnt       : 2
selftest     : passed
type         : blkcipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 0
geniv        : <default>

name         : aes
driver       : aes-asm
module       : aes_x86_64
priority     : 200
```

Abbildung 3: Das Crypto-API des Linux-Kernels gibt bereitwillig Auskunft über seine Fähigkeiten: Die virtuelle Datei »/proc/crypto« listet auf, welche Transformationen und Attribute das Betriebssystem bereitstellt.

rungsspezifisch ist, muss der Programmierer den Speicherbedarf hierfür dem Algorithmenobjekt in dem Attribut »cra_ctxsize« noch mitgeben. Der Speicher für das Transform-Objekt wird dann so groß bemessen, dass sich der Kontext im selben Speicherbereich ablegen lässt. Typischerweise deklariert der Programmierer für den Kontext eine eigene Datenstruktur, wie dies in Listing 2 (Zeile 8) erkennbar ist.

Bitte warten

Im Fall eines asynchronen Blockcipher bekommen die Methoden des Algorithmen-Objekts zum Ver- und zum Entschlüsseln jeweils die Adresse des Request-Objekts übergeben. Hier finden sie insbesondere die Speicheradressen für die Transformation und einen eventuell vorhandenen Initialisierungsvektor. Wird der Auftrag direkt behandelt und erfolgreich abgeschlossen, geben die Methoden 0 zurück, wird der Auftrag asynchron bearbeitet, »-EINPROGRESS«.

Listing 2 empfindet die asynchrone Verarbeitung durch Einsatz eines Tasklet nach. Ein Tasklet ist eine Routine, die das System nach dem nächsten Interrupt abarbeitet und die im Interruptkontext abläuft. Eine produktive Implementierung würde die Funktionalität wohl in einer Interrupt-Service-Routine oder in einem Kernelthread umsetzen. Die Funktion, die das Ende der Bearbeitung identifiziert – im Beispiel das Tasklet –, ruft dann die im Request abgelegte Callback-Funktion (Listing 2, Zeile 25) auf und übergibt ihr den Fehlerstatus, bei erfolgreichem Abschluss des Auftrags also eine 0.

Kompliziert wird die Implementierung eines Transform-Algorithmus durch die komplexen, ineinander geschachtelten Datenstrukturen (Abbildung 4). Diese sind zudem oft unterschiedlich ausgeprägt: Neben globalen Elementen finden sich dort instanzenspezifische Methoden und Attribute. Beispielsweise gibt es das Objekt »struct crypto_tfm« unter anderem in einer Ausprägung für den synchronen (»struct crypto_blkcipher«) und einer für den asynchronen Zugriff (»struct crypto_ablkcipher«).

Header-Studium

Um sich hier zum gewünschten Element durchzuhangeln, hilft nur, die zugehörigen Deklarationen in der Headerdatei »<linux/crypto.h>« gründlich zu studieren [3]. Diese Headerdatei stellt auch einige Inline-Funktionen wie beispielsweise »crypto_tfm_ctx()« (Listing 2, Zeile 31)

Listing 1: Asynchrone Verschlüsselung (»ablk_user.c«, Teil 1)

```
001 #Include <linux/crypto.h>
002 #include "internal.h"
003 #include <linux/scatterlist.h>
004
005 #define CIPHERTEXT "\x01\x0c\x07\x1b\x58\
    x73\x2e\x09"\
006     "\x0b\x14\x1f\x1a\x1d\x65\x6c\x2e"
007
008 static char *key = "Mein Schluessel.";
009
010 struct crypt_result {
011   struct completion completion;
012   int err;
013 };
014
015 static void request_done(struct
     crypto_async_request *req, int err)
016 {
017   struct crypt_result *res = req->data;
018
019   printk("request_done()\n");
020   if (err == -EINPROGRESS)
021     return;
022   res->err = err;
023   complete(&res->completion);
024 }
025
026 static int __init ablkuser_init( void )
027 {
028   int ret;
029   struct scatterlist sg[2];
030   char *encrypted, *decrypted;
031   struct crypto_ablkcipher *tfm;
032   struct ablkcipher_request *req;
033   struct crypt_result result;
034
035   printk("ablkuser_init\n");
036   encrypted = kzalloc(16, GFP_KERNEL);
037   if (!encrypted)
038     goto alloc_encrypted_failed;
039   decrypted = kzalloc(16, GFP_KERNEL);
040   if (!decrypted)
041     goto free_encrypted;
042
043   init_completion(&result.completion);
044
045   memcpy(encrypted, CIPHERTEXT, 16);
046
047   sg_init_table( sg, ARRAY_SIZE(sg) );
048   sg_set_buf( &sg[0], encrypted, 16 );
049   sg_set_buf( &sg[1], decrypted, 16 );
050
051   tfm = crypto_alloc_ablkcipher("ablk", 0, 0);
052   if (IS_ERR(tfm)) {
053     printk("crypto_alloc_ablkcipher failed
    %ld\n", (long)tfm);
054     goto free_decrypted;
055   }
056   ret = crypto_ablkcipher_setkey(tfm, key,
    strlen(key));
057   if (ret<0) {
058     printk("crypto_ablkcipher_setkey
    failed\n");
059     goto free_cipher;
060   }
061   /* Auftragsvergabe */
062   req = ablkcipher_request_alloc(tfm,
     GFP_KERNEL);
063   if (!req) {
064     printk("ablkcipher_request_alloc
    failed\n");
065     goto free_cipher;
066   }
067   ablkcipher_request_set_callback(req,
    CRYPTO_TFM_REQ_MAY_BACKLOG,
068       request_done, &result);
069   crypto_ablkcipher_clear_flags(tfm, ~0);
```

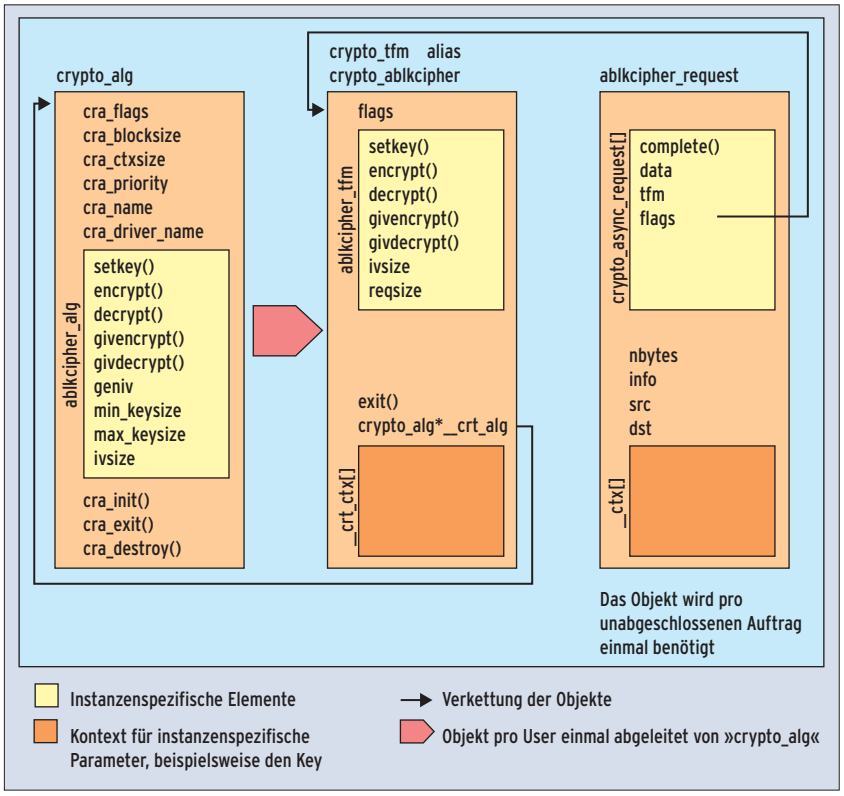


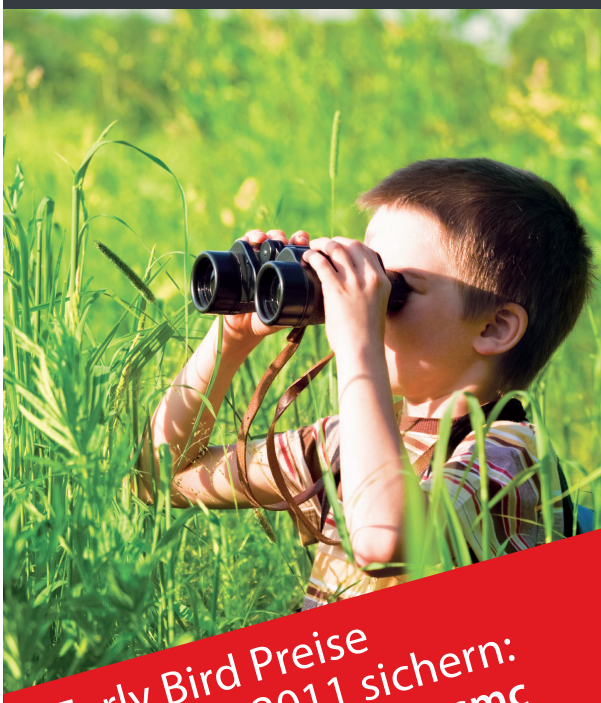
Abbildung 4: Die Datenstrukturen des Crypto-Subsystems sind unübersichtlich und verschachtelt.

zur Verfügung, die den Zugriff auf die Elemente abstrahieren. Abbildung 5 zeigt, wie »make« aus den beiden Codebeispielen zwei Kernelmodule generiert. Nacheinander geladen – erst der Verschlüsselungsalgorithmus, danach das zugreifende Testmodul – entschlüsseln sie den Cyphertext. An dieser Stelle noch einmal der Hinweis: Der vorgestellte Verschlüsselungsalgorithmus dient aus-

Listing 1: Asynchrone Verschlüsselung (»ablk_user.c«, Teil 2)

```
070   ablkcipher_request_set_crypt(req,
      &sg[0], &sg[1], 16, NULL);
071   ret = crypto_ablkcipher_decrypt(req);
072   if (ret == 0) {
073     request_done( &req->base, 0 );
074   } else if (ret!=-EINPROGRESS &&
      ret!=-EBUSY) {
075     printk("crypte_ablkcipher_decrypt
      failed %d\n", ret);
076     goto free_all;
077   }
078   ret = wait_for_completion_
      interruptible(&result.completion);
079   if (ret || (ret = result.err)) {
080     printk("decryption failed %d\n",
      ret);
081     goto free_all;
082   }
083
084   print_hex_dump(KERN_INFO,"encr> ",
      DUMP_PREFIX_NONE,16,1,encrypted,16,1);
085   printk("\n");
086   print_hex_dump(KERN_INFO,"decr> ",
      DUMP_PREFIX_NONE,16,1,decrypted,16,1);
087   printk("\n");
088
089 free_all:
090   ablkcipher_request_free(req);
091 free_cipher:
092   crypto_free_ablkcipher(tfm);
093 free_decrypted:
094   kfree(decrypted);
095 free_encrypted:
096   kfree(encrypted);
097 alloc_encrypted_failed:
098   return -EIO; /* macht das Testen
      einfacher */
099 }
100
101 static void __exit ablkuser_exit
    ( void ) {
102   return;
103 }
104
105 module_init(ablkuser_init);
106 module_exit(ablkuser_exit);
107
108 MODULE_LICENSE("GPL");
```

THE 6TH CONFERENCE ON NAGIOS
6. OPEN SOURCE MONITORING CONFERENCE
OSMC ON NAGIOS



Early Bird Preise bis 15.08.2011 sichern: www.netways.de/osmc

CALL FOR PAPERS BIS 30.7

Die **OSMC on Nagios** (früher "Nagios Konferenz") findet in diesem Jahr zum 6. Mal statt.

Die Veranstaltung bildet eine Plattform, zum Austausch und zur Weiterbildung rund um das Thema Open Source Monitoring mit Nagios.

NÜRNBERG
29.-30. November 2011
[WWW.NETWAYS.DE/OSMC]

```
root@ezs-mobil:/tmp/treiber# cat Makefile
ifneq ($(KERNELRELEASE),)
obj-m   := ablk.o ablk_user.o

else
KDIR    := /lib/modules/$(shell uname -r)/build
PWD     := $(shell pwd)

default:
        $(MAKE) -C $(KDIR)      M=$(PWD) modules
endif
root@ezs-mobil:/tmp/treiber# make
make    -C /lib/modules/2.6.38-8-generic/build  M=/tmp/treiber modules
make[1]: Betrete Verzeichnis '/usr/src/linux-headers-2.6.38-8-generic'
  CC [M]  /tmp/treiber/ablk.o
  CC [M]  /tmp/treiber/ablk_user.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /tmp/treiber/ablk.mod.o
  LD [M]  /tmp/treiber/ablk.ko
  CC      /tmp/treiber/ablk_user.mod.o
  LD [M]  /tmp/treiber/ablk_user.ko
make[1]: Verlasse Verzeichnis '/usr/src/linux-headers-2.6.38-8-generic'
root@ezs-mobil:/tmp/treiber# insmod ablk.ko
root@ezs-mobil:/tmp/treiber# insmod ablk_user.ko
insmod: error inserting 'ablk_user.ko': -1 Input/output error
root@ezs-mobil:/tmp/treiber# tail -n 10 /var/log/kern.log
Jun  2 16:44:12 ezs-mobil kernel: [25942.143617] crypto_ablk_mod_init()
Jun  2 16:44:18 ezs-mobil kernel: [25947.773866] ablkuser_init
Jun  2 16:44:18 ezs-mobil kernel: [25947.773885] ablk_setkey( ffffffffa0066064, 16 )
Jun  2 16:44:18 ezs-mobil kernel: [25947.773891] ablk_decrypt( ffff8800b144bd20 )
Jun  2 16:44:18 ezs-mobil kernel: [25947.773907] ablk_doit(ffff8800b144bd20)
Jun  2 16:44:18 ezs-mobil kernel: [25947.773911] request_done()
Jun  2 16:44:18 ezs-mobil kernel: [25947.773922] encr> 01 0c 07 1b 58 73 2e 09 0b 14 1f
 1a 1d 65 6c 2e  ....Xs.....el.
Jun  2 16:44:18 ezs-mobil kernel: [25947.773927]
Jun  2 16:44:18 ezs-mobil kernel: [25947.773932] decr> 4c 69 6e 75 78 20 4d 61 67 61 7a
 69 6e 00 00 00  Linux Magazin...
Jun  2 16:44:18 ezs-mobil kernel: [25947.773936]
root@ezs-mobil:/tmp/treiber#
```

Abbildung 5: Zwei Module entschlüsseln die Geheimnachricht.

schließlich zu Demonstrationszwecken und gewährleistet in der vorgestellten Form keinen Schutz! Die implementierte XOR-Verschlüsselung ist nur dann sicher, wenn der Schlüssel mindestens ebenso lang wie die Nachricht ist und außerdem die Schlüsselwerte zufällig verteilt sind. Im Ernstfall greift der Programmierer also lieber zu etwas Bewährtem wie AES mit einem ausreichend langen Schlüssel.

Das Crypto-Subsystem des Linux-Kernels bietet noch weit mehr Funktionalitäten als diese und die vorige Kern-Technik beschrieben haben. Dazu gehören beispielsweise der Umgang mit Initialisierungsvektoren über so genannte Templates oder das automatische Laden eines Transformations-Moduls. Offene Fragen kann der Programmierer aber leider fast ausschließlich durch Quellcodestudium klären. *(mhu)*

Infos:

[1] Jürgen Quade und Eva-Katharina Kunst, „Kern-Technik“, Folge 57: Linux-Magazin 06/11, S. 86

[2] Sebastian Siewior, „Acceleration of encrypted communication using co-processors“: [http://diploma-thesis.siewior.net/html/diplomarbeitch4.html]

[3] Headerdatei »linux/crypto.h«: [http://lxr.free-electrons.com/source/include/linux/crypto.h]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Zwischenzeitlich ist die dritte Auflage ihres Buches „Linux Treiber entwickeln“ erschienen.

Listing 2: Asynchroner Blockcipher (»ablk.c«)

```
01 #include <linux/module.h>
02 #include <linux/interrupt.h>
03 #include <linux/scatterlist.h>
04 #include <crypto/algapi.h>
05
06 static struct tasklet_struct ablk_tasklet;
07
08 struct ablk_ctx {
09   char key[16];
10 };
11
12 static void ablk_doit( unsigned long data )
13 {
14   int error = 0, i;
15   struct ablkcipher_request *req = (struct
    ablkcipher_request *) data;
16   char *src, *dst;
17   struct ablk_ctx *ctx;
18
19   printk("ablk_doit(%p)\n", req );
20   src = sg_virt( req->src );
21   dst = sg_virt( req->dst );
22   ctx = crypto_tfm_ctx(req->base.tfm);
23   for( i=0; i<req->nbytes; i++ )
24     dst[i] = src[i] ^ ctx->key[i];
25   req->base.complete(&req->base, error);
26 }
27
28 static int ablk_setkey(struct
    crypto_ablkcipher *tfm,
29     const u8 *key, unsigned int keylen)
30 {
31   struct ablk_ctx *ctx =
    crypto_tfm_ctx(&tfm->base);
32
33   printk("ablk_setkey( %p, %d )\n", key,
    keylen );
34   if (keylen == 16) {
35     memcpy(ctx->key, key, keylen);
36     return 0;
37   }
38   return -EINVAL;
39 }
40
41 int ablk_decrypt(struct ablkcipher_request
    *req )
42 {
43   printk("ablk_decrypt( %p )\n", req);
44   tasklet_init( &ablk_tasklet, ablk_doit,
    (unsigned long)req );
45   tasklet_schedule( &ablk_tasklet );
46   return -EINPROGRESS;
47 }
48
49 int ablk_encrypt(struct ablkcipher_request
    *req )
50 {
51   printk("ablk_encrypt( %p )\n", req);
52   tasklet_init( &ablk_tasklet, ablk_doit,
    (unsigned long)req );
53   tasklet_schedule( &ablk_tasklet );
54   return -EINPROGRESS;
55 }
56
57 static struct crypto_alg ablkcipher_ablk = {
58   .cra_list       =
    LIST_HEAD_INIT(ablkcipher_ablk.cra_list),
59   .cra_flags      =
    CRYPTO_ALG_TYPE_ABLKCIPHER|CRYPTO_ALG_ASYNC,
60   .cra_blocksize  = 16,
61   .cra_ctxsize    = sizeof(struct ablk_ctx),
62   .cra_priority   = 100,
63   .cra_name       = "ablk",
64   .cra_type       = &crypto_ablkcipher_type,
65   .cra_u          = { .ablkcipher = {
66     .setkey       = ablk_setkey,
67     .encrypt      = ablk_encrypt,
68     .decrypt      = ablk_decrypt,
69     .geniv        = NULL,
70     .min_keysize  = 0,
71     .max_keysize  = 16,
72     .ivsize       = 0,
73     },
74   }
75 };
76
77 static int __init crypto_ablk_mod_init(void)
78 {
79   printk("crypto_ablk_mod_init()\n");
80   return crypto_register_alg(
    &ablkcipher_ablk);
81 }
82
83 static void __exit crypto_ablk_mod_exit(void)
84 {
85   printk("crypto_ablk_mod_exit()\n");
86   tasklet_kill( &ablk_tasklet );
87   crypto_unregister_alg(&ablkcipher_ablk);
88 }
89
90 module_init(crypto_ablk_mod_init);
91 module_exit(crypto_ablk_mod_exit);
92 MODULE_LICENSE("GPL");
```

Für die Zukunft gerüstet – Linux-Kernel 3.0

Tragfähige Architektur

Was steckt im Linux-Kernel 3.0? Ein Rundgang durch Taskverwaltung, Memory-Management und weitere wichtige Architekturmerkmale eines modernen Betriebssystemkerns. Jürgen Quade, Eva-Katharina Kunst



„Der Punkt ist, dass 3.0 nichts weiter als eine Neunummerierung bedeutet“, schreibt Linus Torvalds trocken in seiner Freigabemail zur Vorabversion des neuen Kernels **[1]**. Im Vergleich zum letzten Kernel der 2.6er Serie – Kernel 2.6.39 – hat Torvalds recht: „Keine Änderungen am Systemcall-Interface, keine Änderungen an der Programmierschnittstelle oder sonstige neue, magische Features, nur der übliche Fortschritt, im Wesentlichen Treiberupdates.“

Stille Revolution

Doch es kommt auf die Perspektive an. Der Vergleich mit Kernel 2.6.0 offenbart durchaus eine Revolution: Aus dem Kernel für ein Standardbetriebssystem ist in acht Jahren ein Kernel mit ausgefuchsten Echtzeiteigenschaften geworden. Seine Grundwerte hat er trotzdem behalten: Skalierbarkeit, Portierbarkeit, Cutting Edge und Open Source. Linux erschließt sich neue Einsatzfelder, ohne alte aufzugeben.

Linux ist, wie auch schon die ersten Unix-Systeme, monolithisch aufgebaut. Alle zentralen Dienste sind in einem großen Ganzen, dem Kernel, untergebracht. Das erhöht zwar – wie die Befürworter einer Mikrokernel-Architektur behaupten – das Risiko für Instabilitäten, in der Praxis aber überwiegen die Performance-Vorteile bei Weitem, die sich durch den Wegfall von Kontext-Wechselzeiten ergeben. Der Linux-Kern gilt trotzdem von jeher als ausgesprochen stabil.

Abbildung 1 zeigt grob den Aufbau des Systems. Das Systemcall-Interface legt Art und Anzahl der Dienste fest, die eine Applikation vom Betriebssystem anfordern kann: Jobs starten oder beenden, Daten speichern, lesen oder per TCP/IP an andere Rechner übertragen oder einfach nur die Uhrzeit auslesen. Kernel 2.6.0 offerierte bei einem 32-Bit-Linux 274 Dienste, bei 3.0 sind es schon 347. Die Dienste erbringen die darunter liegenden Blöcke I/O-, Task- und Memory-Management, die auf die Hardware über die Gerätetreiber zugreifen.

Bei feinerer Unterteilung werden die Subsysteme des Kernels sichtbar, etwa für den Zugriff auf USB- oder PCI-Geräte, für Multimedia und Netzwerk. Die Schnittstellen zu den Subsystemen sind im Linux-Kernel meist so flexibel, dass sich verschiedenste Implementierungen für einzelne Funktionen – etwa den I/O-Scheduler – andocken lassen.

Innerhalb des Kernels ist die Taskverwaltung von zentraler Bedeutung, sie ist für das Multitasking – die gleichzeitige Abarbeitung mehrerer Programme – zuständig. Dabei mehrere CPU-Kerne zu nutzen ist für Linux seit vielen Jahren selbstverständlich.

Grundsätzlich ist das Scheduling – also das Festlegen, wann welche Task wie lange auf welchem CPU-Kern arbeitet – zweistufig aufgebaut: Der Multicore-Scheduler fasst einzelne Prozessoren beziehungsweise Prozessorkerne einer Mehrkernmaschine zu Gruppen zusammen. Innerhalb der Gruppe verteilt Linux die Tasks auf die CPUs. In der zweiten Stufe, dem Singlecore-Scheduling, wählt jeder einzelne Prozessorkern aus den zugeteilten Tasks jene aus, die laufen darf.

Verteilungskämpfe

Bevor der Multicore-Scheduler aktiv wird, hat der Kernel bereits beim Booten ein Abbild der vorhandenen Hardware angefertigt und diese aufgrund der Multicore-Architektur (Hyperthreading, Symmetric Multiprocessing, Non Uniform Memory Architecture) hierarchisch in so genannte Scheduling-Domains eingeteilt. Sie bestehen aus Scheduling-Gruppen, diese

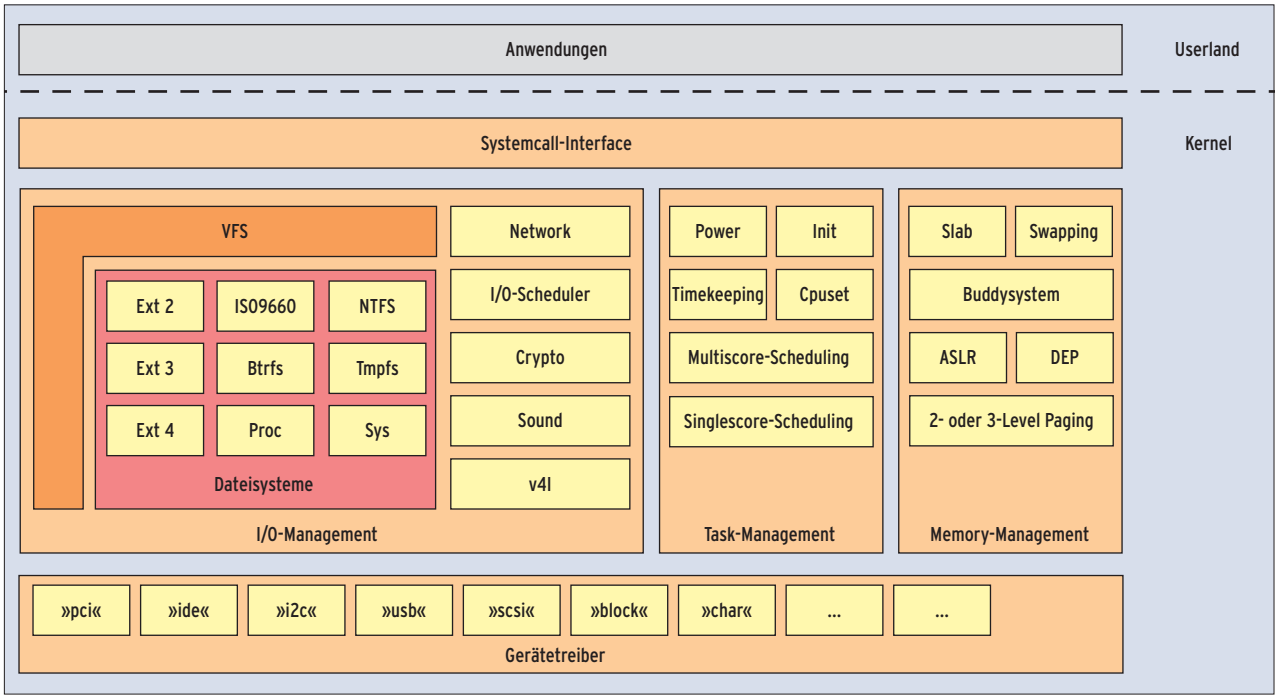


Abbildung 1: Systemarchitektur: Kernel 3.0 gehört zwar zu den monolithischen Betriebssystemkernen, ist aber modular aufgebaut.

wiederum aus CPU-Kernen oder anderen Scheduling-Domains. Jeder Gruppe beziehungsweise jedem CPU-Kernel ist eine Liste der zugehörigen Jobs zugeordnet.

Lastverteilung

Der Scheduler sorgt durch Prozessmigration, also das Verschieben eines Jobs auf einen anderen Kern oder in eine andere Gruppe, innerhalb seiner Domain für eine ausbalancierte Lastverteilung. Multicore-Scheduling ist immer dann aktiv, wenn sich ein Job beendet, ein neuer Job gestartet wird, sich schlafen legt oder wenn ein starkes Ungleichgewicht bei der Auslastung der einzelnen Rechnerkerne festgestellt wird. Ob in letzterem Fall eine Prozessmigration Vorteile bringt, rechnet Linux in jedem Einzelfall aus, zudem kann der Sysadmin dies per Kommandozeilen-Option beeinflussen.

Fürs Singlecore-Scheduling hat Linux mit Kernel 2.6.23 ein erweiterbares Framework zur einfachen Implementierung von Scheduling-Algorithmen verpasst bekommen. Dieses Framework führt so genannte Scheduling-Klassen ein, die per implementiertem Algorithmus aus den zugeordneten Jobs den nächsten zu bearbeitenden auswählen. Liefert eine Klasse auf Nachfrage keinen Job, fragt der Scheduler bei der nächsten nach.

Im Standardkernel sind drei Klassen implementiert. Die Klasse »rt_sched_class« ist für Echtzeitprozesse zuständig. Der in dieser Klasse implementierte Algorithmus realisiert ein prioritätengesteuertes Scheduling. Besitzen mehrere Jobs die gleiche Priorität, wird entweder per FCFS (First Come First Served) oder per Zeitscheibenverfahren (Round Robin) ausgewählt. Insgesamt bietet Linux in dieser Klasse 99 Prioritätsebenen.

Normal gestartete Jobs sind der Klasse »fair_sched_class« zugeordnet, die den Completly Fair Scheduler (CFS) realisiert. Dieser Algorithmus sortiert lauffähige Rechenprozesse nicht mehr – wie in den ersten 2.6-Versionen – in Listen, sondern in Red-Black-Trees. Das ermöglicht ihm, mit wenig Rechenaufwand eine faire Verteilung der Rechenzeit vorzunehmen, die nicht mehr auf Heuristiken, sondern allein auf Mathematik beruht.

Im Userland sind dabei weiterhin Nice-Level (Kommando »nice«) sichtbar, die schon die allerersten Unix-Systeme kannten und eine Priorisierung von Jobs im Bereich von -20 bis +19 ermöglichen. Das Kommando »ps -ce« stellt diese als Linux-Prioritäten von 0 bis 39 dar.

Die Echtzeitprioritäten (1 bis 99) werden bei dieser Ausgabe übrigens als Linux-Prioritäten 41 bis 139 ausgegeben, sodass zur Umrechnung von Linux-Priorität in eine Posix-Realzeitpriorität immer 40 zu subtrahieren ist (Abbildung 2). Die dritte Klasse heißt »idle_sched_class« und tritt nur in Aktion, wenn es keinen anderen lauffähigen Job gibt.

Bitte nicht stören!

Die Unterbrechbarkeit (Preemption) durch den Scheduler bezieht sich im Linux-Kernel auch auf Funktionen, die im Kernel- oder im Prozesskontext abgearbeitet werden (siehe Kasten „Unterbrechungsmodell“), also Systemcalls oder Kernelthreads. Klassischerweise bringt

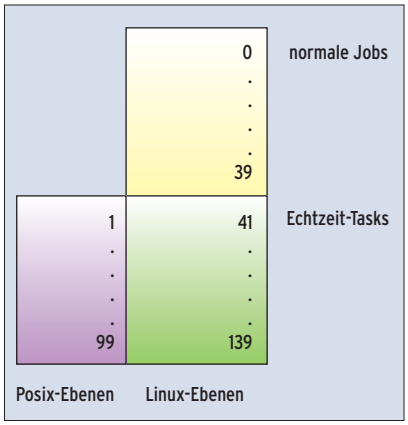


Abbildung 2: Aus logischer Sicht unterstützt Linux 140 Prioritätsebenen, die in einen Bereich für normale Jobs und in einen für Echtzeit-Tasks unterteilt sind. Die Posix-Spezifikation zählt dabei anders.

Linux einmal im Kernel begonnene Funktionen erst bis zum nächsten planmäßig unterbrechbaren Punkt aus, bevor es andere Funktionen auf Kernel- oder Prozessebene in Angriff nimmt. Das gilt auch für den Fall, dass – etwa per Hardware-Interrupt signalisiert – eine sehr wichtige Funktion abzuarbeiten ist.

In einem modernen Linux-Kernel ist das anders: Höher priorisierte Kernelfunktionen unterbrechen niedriger priorisierte. Dadurch reduzieren sich die Latenzzeiten deutlich. Einziger Nachteil: Auf Single-core-Maschinen gibt es mehr kritische Abschnitte, die das Betriebssystem schützen muss.

Eine sinnvolle Ergänzung zur Kernel-Preemption sind Threaded Interrupts (Abbildung 3). Startet ein Kernel 3.0 mit der Option »threadirqs«, arbeitet das Betriebssystem die zentralen Teile der Interrupt-Serviceroutinen (ISRs) im Kernelkontext ab, also als Kernelthreads. Damit beeinflusst der Admin oder der Systemarchitekt die Priorisierung nicht nur der Interrupts untereinander, sondern sogar in Beziehung zu sonstigen Aufgaben.

Jiffies

Damit der Scheduler überhaupt aktiv werden kann, sind Interrupts notwendig. Daher erzeugt klassischerweise ein Timer-Baustein periodisch (zum Beispiel alle 10 Millisekunden) einen Interrupt. Die Anzahl der Interrupts zählt Linux in der Variablen »jiffies« mit, sie sind so etwas wie der Herzschlag des Kernels. In Kernel 2.6 hatte Linus Torvalds den zeitlichen Abstand auf 4 Millisekunden reduziert und so das Interaktivitätsverhalten spürbar verbessert. Das geht allerdings zu Lasten der Effizienz, schließlich wird auf diese Art der Kernel häufiger aktiv.

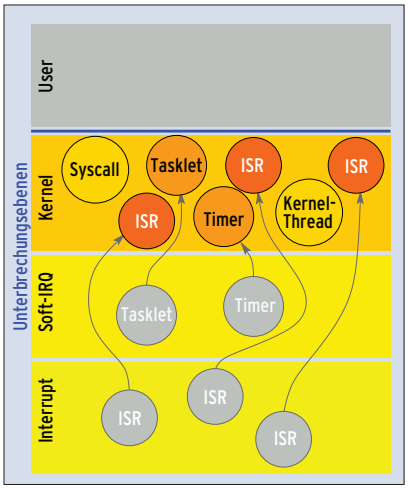


Abbildung 3: Interrupt-Serviceroutinen und Soft-IRQs mutieren zu Threads, was sie unterbrechbar macht: Ein Echtzeitprozess, der auf der Userebene läuft, kann sie verdrängen.

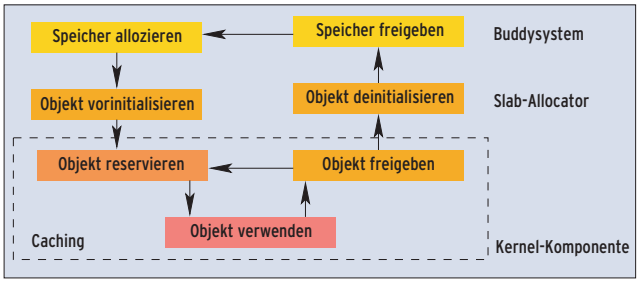


Abbildung 4: Der Lebenszyklus typisierter Kernelobjekte: Indem der Linux-Kernel massiv Objekte wiederverwendet, spart er sich das andauernde Reservieren und Initialisieren von Speicher und steigert so seine Performance.

Herz ohne Schlag

Kernel 3.0 ist tickless: Er erzeugt Interrupts nicht mehr periodisch, sondern zu den Zeitpunkten, zu denen sie auch tatsächlich benötigt werden. Die Liste der Vorteile ist beeindruckend: Systemlast reduziert, Effizienz gesteigert, gerade auf mobilen Endgeräten wird durch längere Zeiten der Inaktivität Energie eingespart und die Genauigkeit zeitgesteuerter Aktionen erhöht, da Zeitaufträge nicht mehr zum Zeitpunkt des nächsten periodisch auftretenden Interrupts abgearbeitet werden, sondern punktgenau.

Natürlich gibt es auch Kosten: Mit jedem Interrupt muss der Kernel den nächsten Zeitpunkt berechnen, den Timer-Baustein neu programmieren und die interne Zeitbasis aktualisieren. Jiffies gibt es zwar weiterhin, aber unabhängig davon haben die Linux-Entwickler das Timekeeping mit dem Tickless-System auf Nanosekunden-Genauigkeit umgebaut.

Die zweite wesentliche Komponente des Kernels ist die Speicherverwaltung (Memory-Management). Eine ihrer Aufgaben ist die Adressenumsetzung. Der Kernel sorgt dafür, dass jede Applikation auf einen Hauptspeicher zugreifen kann, der an Adresse 0 startet und dann – je nach Konfiguration – 3 oder 4 oder noch mehr GByte umfasst.

Die Methode, die dabei vor allem zum Einsatz kommt, ist das so genannte Paging. Es teilt den Hauptspeicher in Seiten gleicher Größe (Pages) ein. Typischerweise hat eine Page eine Größe von 4 KByte. Einen 4 GByte großen Hauptspeicher teilt das Betriebssystem somit in eine Millionen Speicherseiten ein. Um eine Millionen Pages zu adressieren, benötigt man 20 Bit; um innerhalb der Page eine Speicherzelle auszusuchen 12 Bit. Auf einem 32-Bit-System (PC-Plattform) ist das normalerweise über eine zweistufige Speicherverwaltung (Two-Level-Paging) realisiert, auf einem 64-Bit-Rechner über eine dreistufige.

Aus Sicherheitsgründen variiert Linux die Adressenlagen der einzelnen Segmente einer Applikation (Heap, Stack, Shared Libraries) um jeweils einen zufälligen Wert. Dass Linux bei dieser Adress Space Layout Randomisation (ASLR) genannten Technik stärker variiert als andere Betriebssysteme, ist ein wichtiges Sicherheitsmerkmal.

Speicherschutz

In dieselbe Kerbe schlägt der Speicherschutz, eine weitere Aufgabe des Memory-Managements. Applikationen dürfen nicht auf die Speicherbereiche einer anderen Applikation oder auf die des Kernels zugreifen und mal eben die im Hauptspeicher abgelegten Passwörter auslesen. Zur Realisierung von Speicherschutz benutzt Linux die in Hardware festgelegten Mechanismen.

Das gilt auch für den Schutz vor Ausführung von Code auf dem Stack (Data Execution Prevention, DEP) – ebenfalls eine wesentliche Eigenschaft zum Schutz vor Hackerangriffen. Unglücklicherweise aktivieren die Linux-Distributoren in ihren 32-Bit-Versionen dieses Feature des Kernels meist nicht.

Die Speicherverwaltung hat noch mehr Aufgaben: Dank Swapping macht sich ein Applikationsprogrammierer keine Gedanken über die Menge des physisch zur Verfügung stehenden Hauptspeichers. Und sollten auf einem 32-Bit-System

mehr als die adressierbaren 4 GByte zur Verfügung stehen, ist dank Memory Management auch dieser erweiterte Speicher nutzbar.

Darüber hinaus setzt Linux auf typisierte Objekte und nutzt mit dem Slab-Allocator die Vorteile der Wiederverwertung [3]. Hierbei stellt der Kernel eine Reihe bereits vorinitialisierter Objekte zur Verfügung. Das hat vor allem Performance-Vorteile: Das gleichzeitige Initialisieren mehrerer gleichartiger Objekte nutzt den Hauptspeicher und die Wirkung des Prozessor-Cache besser aus.

Zudem fällt Code weg, weil sich die Operationen „Speicher allozieren“, „Objekt vorinitialisieren“, „Objekt deinitialisieren“ und „Speicher freigeben“ einsparen lassen (Abbildung 4). Der Sysadmin bekommt Informationen zu den Objekten, indem er das Kommando »cat /proc/slabinfo« aufruft.

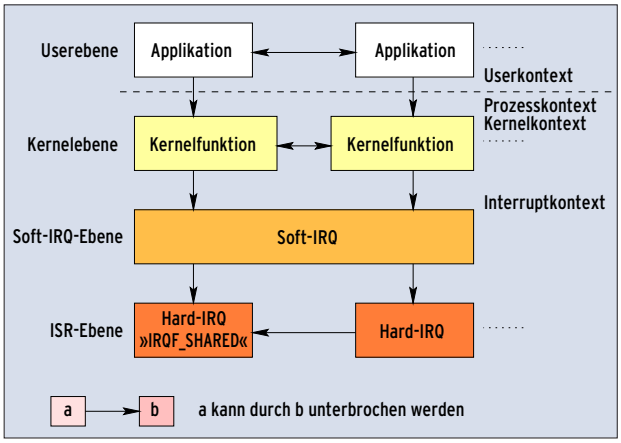


Abbildung 5: Kernel-Programmierer müssen das Unterbrechungsmodell kennen, um kritische Abschnitte erkennen und richtig schützen zu können.

Virtual Filesystem Switch

Das I/O-Management ist für den Zugriff auf Dateien und auf Peripherie zuständig. Zentrale Verteilstation ist hierfür der Virtual Filesystem Switch (VFS). Dieser verteilt die Zugriffe auf zeichen- oder blockorientierte Geräte, setzt Dateinamen auf Sektornummern um, speichert die

Unterbrechungsmodell

Linux unterscheidet vier Ebenen, auf denen es Code abarbeitet (Abbildung 5): Die Userebene ist dem so genannten Userland und damit Applikationen vorbehalten. Die hier laufenden Tasks können unterbrochen (preempted) werden und sich schlafen legen. Auf Multicore-Maschinen allerdings können die Funktionen real mehrfach parallel arbeiten.

Auf der Kernelebene sind Systemcalls und Kernelthreads angesiedelt, die ähnliche Eigenschaften haben: Sie sind unterbrechbar, laufen dadurch auf Singlecore-Maschinen scheinbar parallel, auf Multicore-Maschinen real parallel und sie können schlafen.

Wenn Schlafen verboten ist

Auf der Soft-IRQ-Ebene ist Schlafen dagegen ein absolutes Tabu. Hier arbeiten Timer und Tasklets, also kurze Codesequenzen, die eine Unterbrechung durch Interrupts zulassen und früher auch „Bottom Half“ genannt wurden. Auf der untersten Ebene schließlich laufen Interrupt-Serviceroutinen (ISRs). Sie können alle darüber liegenden Codesequenzen unterbrechen. Selbst sind sie typischerweise nicht unterbrechbar (Interrupts gesperrt), es sei denn, sie erlauben dies explizit.

Das Unterbrechungsmodell zeigt nicht nur, dass die jeweils niedrigeren Ebenen darüber liegende Ebenen unterbrechen können, sondern auch, in welchem Kontext eine Ebene abgearbeitet wird. ISRs und Soft-IRQs laufen demnach im Interruptkontext.

Auf Kernelebene wird der Prozesskontext vom Kernelkontext unterschieden. Im Prozesskontext laufen all jene Jobs, die zu einer Userland-Applikation gehören. Das sind die Systemcalls, die die Applikationen aufgerufen haben wie etwa »open()«, »close()«, »read()« oder »write()«. Diese Codesequenzen können Daten zwischen dem Kernel und der zugehörigen Applikation austauschen.

Für den Kernelkontext bleiben nur die Kernelthreads übrig. Für jeden Kontext gibt es in den Kernel-Headerdateien ein Define (»GFP_ATOMIC« für den Interruptkontext, »GFP_KERNEL« für den Kernelkontext und »GFP_USER« für den Prozesskontext). Allgemeine Funktionen, beispielsweise »kmalloc()« (»malloc()« im Kernel), lassen sich aus unterschiedlichen Kontexten aufrufen. Ihnen muss der Kernelprogrammierer über das Define den aktuellen Kontext mitgeben, etwa damit die Funktion weiß, ob sie schlafen darf oder nicht.

Tel. 0 64 32 / 91 39-749
Fax 0 64 32 / 91 39-711
vertrieb@ico.de
www.ico.de/linux

ICO INNOVATIVE COMPUTER SEIT 1982

Innovative Computer GmbH • Zuckmayerstr. 15 • 65582 Diez

BALIOS 191 1HE GREEN SERVER Art.Nr. BTO-259446-23



1HE SiEnergieeffizienter 1HE Server mit 8 GB Arbeitsspeicher und 2 Festplatten.

- Intel® Xeon® E3110 Prozessor mit 3,0 GHz
- 8 GB (2 x 4 GB) DDR3 RAM
- 4 TB (2 x 2 TB) WD HDD
- 250W Green Netzteil
- Systemtest mit SuSE Linux 11.3 64bit

www.ico.de/linux

inkl. Mwst 938,91 exkl. Mwst 789,-

Art.Nr. BTO-259447-23

XANTHOS 357 3HE STORAGE SERVER



Skalierbare 3HE Storage-Lösung mit 16 Festplatten-Einschüben und neuestem Intel® Xeon® Quad-Core Prozessor.

- 2x Intel® Xeon® E5620 Prozessor mit 2,4 GHz
- 24 GB (6x 4 GB) DDR3 RAM
- 32 TB (16x 2 TB) x WD HDDs
- Adaptec RAID-Controller 51645 inkl. Kabel BBU
- 500 W Netzteil (2U+3U)
- Systemtest mit SuSE Linux 11.3 64bit

www.ico.de/linux

inkl. Mwst 5829,81 exkl. Mwst 4899,-

Hier geht's direkt zu den Produkten



Art.Nr. BTO-259448-23

BALIOS 553 5HE STORAGE SERVER



Leistungsstarke Storage-Lösung mit Intel® Xeon® Prozessor und redundantem Netzteil.

- Intel® Xeon® X3430 Prozessor mit 2,4 GHz
- 16 GB (4x 4 GB) DDR3 RAM
- 46 TB (24x 2TB) WD HDDs
- Adaptec RAID-Controller 52445 inkl. Kabel BBU
- Intel® Remote Management Modul 3
- 1350W+450W red. Netzteil 5U
- Systemtest mit SuSE Linux 11.3 64bit

www.ico.de/linux

inkl. Mwst 8328,81 exkl. Mwst 6999,-

Alle Preise in Euro

Intel, das Intel Logo, Xeon, und Xeon Inside sind Marken der Intel Corporation in den USA und anderen Ländern.

wir liefern auch nach Österreich u. in die Schweiz

Daten zwischen und gibt Aufträge für die Festplatten an den I/O-Scheduler weiter. Der I/O-Scheduler ist für eine geschickte Sortierung zuständig, sodass der Lesekopf einer Festplatte möglichst wenige Rückwärtsbewegungen macht. Da Solid State Disks (Flashspeicher) keine Köpfe zu bewegen haben, reicht Linux in diesem Szenario bei entsprechender Konfiguration die Daten direkt an die Platte weiter.

Der VFS implementiert auch das hierarchische Modell, gemäß dem Linux Daten auf Festplatten abspeichert oder wieder einliest. Jede Datei ist in diesem Modell über einen Inode repräsentiert [4]. Der Inode, der über eine eindeutige Inodenummer bestimmt ist, speichert die wesentlichen Meta-Informationen: Dateigröße, Zeitstempel bezüglich des Anlegens, des letzten Zugriffs oder der letzten Modifikation, Zugriffsrechte, Besitzverhältnisse und so weiter. Ein einzelner Name gehört allerdings nicht zu diesen Meta-Informationen. Inodenummern lassen sich im Übrigen durch Eingabe von »ls -i« in einer Konsole listen.

Ein Dentry genanntes Objekt verknüpft schließlich einen Dateinamen mit dem Inode. Um per Namen auf eine Datei zugreifen zu können, sind zunächst die Dentries zu durchsuchen. Diesen zeitlich teuren Vorgang beschleunigt Linux durch einen Zwischenspeicher, den Dcache.

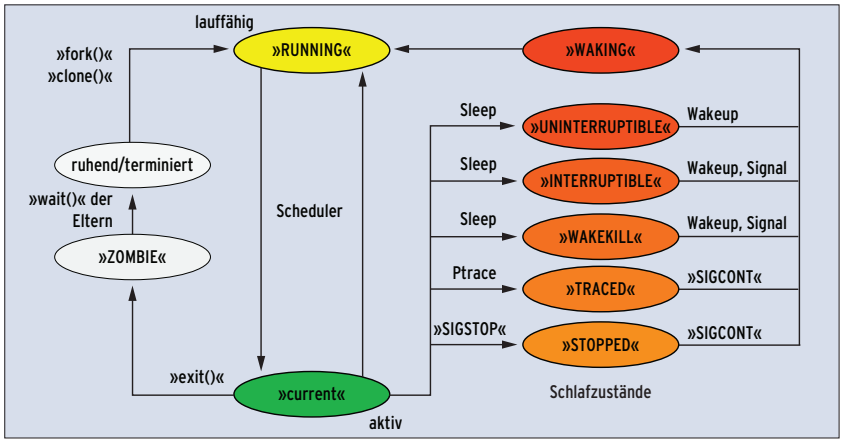


Abbildung 6: Applikationen und Kernelthreads durchlaufen verschiedene Zustände.

Mit dem Inode-Objekt ist eine Reihe von Methoden verknüpft, die – verkürzt ausgedrückt – die Abbildung auf die Festplatte und die dort vorhandenen Datenstrukturen, das eigentlichen Dateisystem, vornehmen.

Dateisysteme

Aktuell verwenden die meisten Distributionen ein Ext-4-Dateisystem, das nicht nur performant arbeitet, sondern auch besonders große Dateien ermöglicht. Mittelfristig wird aber wohl Btrfs die Rolle des Standardfilesystems übernehmen. Dann sollen auch Filesystemchecks bei einem gemounteten und in Verwendung befindlichen Dateisystem möglich sein.

Die universelle Dateisystemschnittstelle ermöglicht es Linux, auf unterschiedlichste Dateisysteme zuzugreifen, wie etwa FAT, NTFS oder ISO9660.

Außerdem lassen sich Daten auch in virtuellen Dateisystemen ablegen: Aus Sicht der Applikation gibt es Dateien, die real nur im Hauptspeicher und nicht auf der Festplatte zu finden sind. Mehr noch, die Daten entstehen erst beim Zugriff.

Drei virtuelle Dateisysteme kennen die meisten Admins: Das Proc-Filesystem enthält Informationen über sämtliche Rechenprozesse, das Sys-Filesystem über die Hardwarestruktur inklusive der zugehörigen Treiber. Das Temp-Filesystem schließlich ermöglicht die Ablage von Daten im Hauptspeicher, ohne dass überhaupt ein klassisches Dateisystem zum Zuge kommt. Das nutzt Linux, um per Initram-FS quasi ohne Treiber zu booten [5]. Außerdem ermöglicht ein VFS auch das Schichten von Dateisystemen, um damit beispielsweise einfach eine Verschlüsselung zu realisieren.

In enger Verbindung zum VFS stehen die Gerätetreiber, die den weitaus größten Teil des Linux-Quellcode stellen. Historisch wird ein Treiber über eine Major-Nummer identifiziert, die 8 Bit breit ist. Die sich daraus ergebende Anzahl von gerade einmal 256 Treibern ist für ein modernes Betriebssystem jedoch viel zu wenig. Linux arbeitet daher mit Gerätenummern, die insgesamt 32 Bit breit sind. Mit einer Kodierung von rund 4000 Treibern mit insgesamt 4 Milliarden anzusprechenden Geräten ist das ausreichend dimensioniert.

Den größten Sprung von 2.6.0 zu 3.0 hat der Linux-Kernel im Bereich Echtzeitver-

Taskzustände

Tasks können sich in unterschiedlichen Zuständen befinden. Grundsätzlich erzeugt der Systemcall »clone()« eine neue Task als exakte Kopie ihres Elternprozesses. Der neue Job befindet sich im lauffähigen Zustand (»TASK_RUNNING«). Der Scheduler wählt aus allen lauffähigen Jobs jenen aus, den die CPU als nächsten abarbeitet. Der so genannte Context-Switch aktiviert die Task, die sich damit im aktiven Zustand befindet. Es gibt diverse Gründe, warum sich eine Task selbst in den schlafenden Zustand versetzt: Sie wartet auf Daten, sie wartet auf ein Betriebsmittel oder will einfach nur Zeit verstreichen lassen.

Linux unterscheidet verschiedene Arten des Schlafens. In der einfachsten Variante schläft der Job exakt so lange, bis das Ende der Schlafensbedingung erreicht ist und niemand im Kernel die »wake_up«-Funktion aufruft. In diesem Zustand weckt den Job nichts anderes. Nicht gar so tief schläft der Job im Killable-Zustand. Hier kann ihn immerhin das Signal 9 (»SIGKILL«) aufwecken. Den leichten Schlaf repräsentiert der Interruptible-Zustand. Hier wird der Job durch jedes Signal geweckt.

Selbstmord im Kernel

In Abbildung 6 ist gut zu erkennen, dass Jobs sich grundsätzlich selbst beenden. Die Aufforderung zum „Selbstmord" kann jedoch per Signal von anderen Jobs aus (per Systemcall »kill«) erfolgen. Bevor aber sämtliche Spuren des Jobs erloschen sind, nimmt er noch den Zombie-Status an.

In diesem Zustand hat der Kernel zwar bereits die zum Prozess gehörenden Speicherbereiche (Code-, Daten- und Stacksegmente) freigegeben, der Task-Kontrollblock aber, die Datenstruktur, die den Job im Kernel repräsentiert, existiert noch. Bevor diese gelöscht wird, muss die Elterntask den Exitcode (Returnwert der Funktion »main«) abholen. Ist die Elterntask bereits selbst beendet, übernimmt der Job mit der PID 1 (der Init-Prozess) dieses Abholen.

halten gemacht. Nicht zuletzt dank des deutschen Kernelentwicklers Thomas Gleixner, der seine Realtime-Patches in „schmackhafte, Trojanische Pferde“ verpackt hat [6], stehen dem Architekten eines Echtzeitsystems mit Linux viele Möglichkeiten offen: Hochauflösende Timer (Hrtimer) erlauben sehr genaue Zeitsteuerungen, Realtime-Mutexe bieten Prioritätsvererbung und die bisher nicht erwähnten CPU-Sets fixieren Rechenprozesse exklusiv auf spezifischen Rechnerkernen eines Multicoresystems.

Hinzu kommen die erwähnten Techniken Threaded Interrupts, Kernel-Preemption und Tickless Kernel. Das unter Entwicklern berüchtigte Big Kernel Lock (BKL), das in den alten Versionen mal eben Interrupts auf sämtlichen Prozessoren gesperrt hat, ist ebenfalls rechtzeitig vor der Veröffentlichung von 3.0 aus dem Kernel verschwunden.

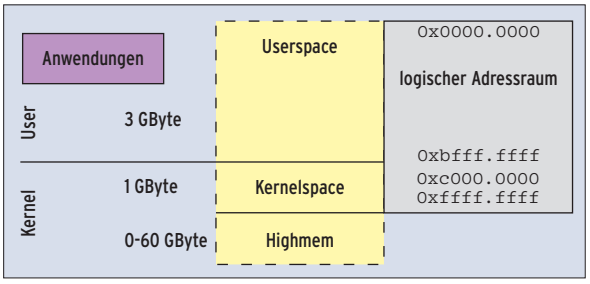


Abbildung 7: Da alle Page-Directories in den höchsten Adressen die gleichen Einträge aufweisen, ist beim Übergang in den Kernel kein Umschalten der Speicherbereiche notwendig.

Code zum Anfassen

Der Linux-Kernel 3.0 präsentiert sich aus vielen Gründen als spannendes Stück produktiver Software. Nicht nur, weil er objektorientiert aufgebaut ist, ohne eine objektorientierte Programmiersprache zu verwenden, und weil er eine ungewöhnliche Skalierbarkeit und Portierbarkeit aufweist oder weil er die Ideen und die Programmierfähigkeiten unterschiedlichster, weltweit verstreuter Entwickler vereint, sondern auch, weil er dank seiner Open-Source-Lizenz ein Stück Hightech zum Anfassen ist. *(mhu)* ■

4G/4G-Patch

Auf einem 32-Bit-System sind Adressenregister 32 Bit breit und geben damit Zugriff auf 4 GByte virtuellen Hauptspeicher. Der Versuch einer Applikation, diesen Adressraum komplett zu nutzen und auf eine Adresse oberhalb von 3 GByte zuzugreifen, scheitert jedoch kläglich mit einem Segmentation Fault. Im Adressraum der Applikation haben die Kernelentwickler das oberste Gigabyte für den Kernel reserviert. Konkret: In dem Page-Directory einer jeden Task stehen in den höchsten Adressen exakt die gleichen Einträge, während die übrigen Einträge Task-spezifisch sind (Abbildung 7).

Dadurch kann der Kernel mit dem Page-Directory jeder gerade aktiven Task auf seine dort referenzierten Speicherbereiche – den Kernelspace – zugreifen. Ohne diesen Trick müsste Linux bei jedem Systemcall und bei jedem Interrupt das Page-Directory, also den Einsprung in die Speicherverwaltung, austauschen. Dazu gehörte dann auch das Leeren des Translation Lookaside Buffer (TLB), des Cache für die Speicherverwaltung. Das ist zeitlich betrachtet ein teurer Vorgang.

Sollte eine speicherhungrige Task doch den Bedarf nach 4 GByte virtuellem Speicher haben, kann der Admin das 4G/4G-Patch aktivieren, das dem Kernel ein eigenes Page-Directory zuweist. In einem solchen Fall sollte er allerdings vorzugsweise den Umstieg auf ein 64-Bit-System erwägen.

Infos:

[1] Linus Torvalds, „Linux 3.0-rc1“: [http://thread.gmane.org/gmane.linux.kernel/1147415]

[2] Quade, Kunst, „Kern-Technik“, Folge 33 (Multicore): Linux-Magazin 05/07, S. 52

[3] Quade, Kunst, „Linux-Treiber entwickeln“, 3. Auflage: Dpunkt-Verlag, 2011, S 232 ff.

[4] Quade, Kunst, „Kern-Technik“, Folge 23 (VFS): Linux-Magazin 10/05, S. 90

[5] Quade, Kunst, „Kern-Technik“, Folge 39 (Tempfs): Linux-Magazin 05/08, S. 98

[6] Thomas Gleixner, „Forced threaded interrupt handlers“: [https://lkml.org/lkml/2011/2/23/510]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Mittlerweile ist die dritte Auflage ihres Buches „Linux Treiber entwickeln“ erschienen.

opensourcepress.de

open source PRESS

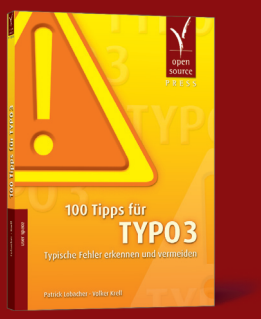
Bücher



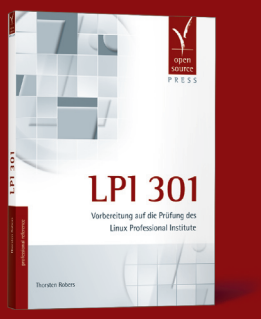
ISBN 978-3-941841-42-0
328 Seiten · brosch. · € 29,90 [D]



ISBN 978-3-941841-44-4
ca. 300 Seiten · brosch. · € 24,90 [D]



ISBN 978-3-941841-26-0
248 Seiten · brosch. · € 24,90 [D]



ISBN 978-3-941841-36-9
343 Seiten · brosch. · € 34,90 [D]

open source SCHOOL

Kurse



05.–09. September Vorbereitung auf LPIC-2
10. – 14. Oktober Vorbereitung auf LPIC-1

Diese Kurse bereiten Sie auf die Prüfungen 101 und 102 bzw. 201 und 202 des Linux Professional Institute vor. Am letzten Kurstag können Sie die Prüfungen bei uns ablegen!

Dozent: Michael Gisbers

Auswahl Kurstermine August – Oktober 2011

TCP/IP-Analyse mit Wireshark	31.08 – 02.09.
Python	31.08 – 02.09.
OTRS::ITSM	01. – 02.09.
Qt für Fortgeschrittene	08. – 09.09.
Vorbereitung auf LPI 301	12. – 16.09.
Nagios Advanced	15. – 16.09.
Novell 3102	19. – 23.09.
C++: STL und Boost	26. – 27.09.
Performance-Analyse mit collectd	26. – 28.09.
Samba in heterogenen Netzen	26. – 28.09.
Virtualisierung mit KVM	04. – 07.10.
IPv6	10. – 12.10.

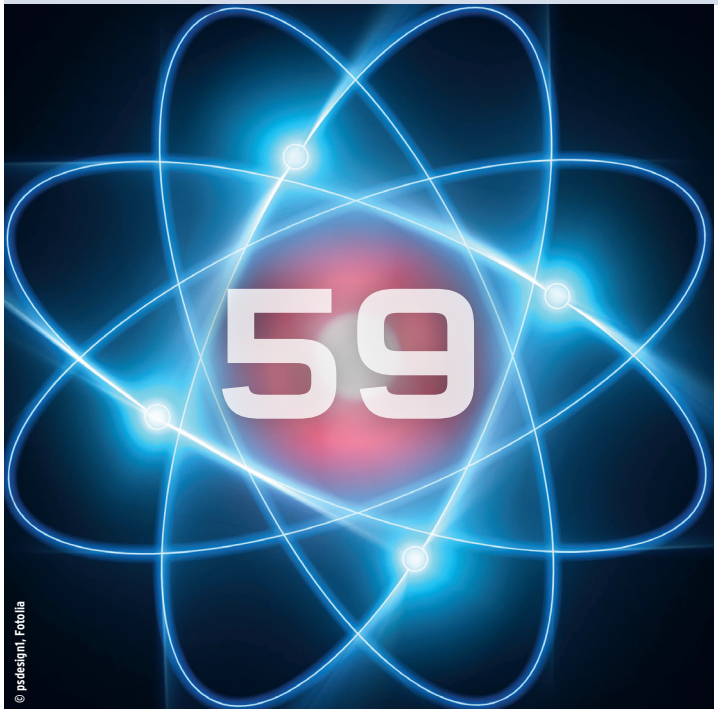
opensourceschool.de

Sonderpreise für Studierende

Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 59

Kern-Technik

Linux plattformübergreifend: Die praktische Skriptsammlung Buildroot erzeugt im Handumdrehen einen Kernel für Embedded-Systeme mit ARM-CPU. Als Nebenprodukt fällt dabei ein komplettes, lauffähiges Linux-System ab, das sich im Emulator Qemu testen lässt. Jürgen Quade, Eva-Katharina Kunst



© psdesignt, Fotolia

In den Anfangstagen von Linux war es nicht abzusehen, dass der ursprünglich ausschließlich für i386-CPUs entwickelte Betriebssystemkern einmal mehr als 20 verschiedene Prozessoren unterstützen würde. Heute gehen dank Smartphone- und Tablet-Boom täglich mehr Linux-Systeme auf einer ARM-basierten Plattform in Betrieb als auf klassischen, x86-basierten Desktop- oder Serversystemen.

Host und Target

Der herkömmliche PC dient jedoch weiterhin als Entwicklungsrechner (Host), um darauf die Systemsoftware, einen Kernel und ein Root-Dateisystem für das Ziel (Target), etwa ein Android-Smartphone, zu generieren. Für diese so genannte Cross-Entwicklung braucht man Compiler, Linker und Bibliotheken, die Code für den anderen Prozessortyp erzeugen können.

Verantwortlich für das Bauen eines Kernels ist das zum Linux-Quellcode gehörende Kernel-Build-System. Es benötigt die üblichen Entwicklungswerkzeuge (unter Ubuntu die Pakete »ncurses-dev« und »build-essentials«) sowie Cross-Compiler und Cross-Linker (in der Regel GCC und den GNU-Linker »ld«) auf dem Entwicklungsrechner.

Dann kann das System mit Hilfe der Environment-Variablen beziehungsweise Aufrufparameter »ARCH«, »CROSS_COMPILE« und »O« den Code für die Zielplattform generieren. »ARCH« spezifiziert die Zielplattform – bei Kernel 3.0 stehen die in Abbildung 1 gelisteten Architekturen zur Auswahl. »CROSS_COMPILE« spezifiziert den Teilstring, den das Kernel-Build-System vor den Namen des Compilers und des Linkers einfügt.

Es liegt im Namen

Typischerweise haben die Cross-Entwicklungswerkzeuge einen Namen, der unter anderem auf die Zielplattform hinweist: »arm-linux-gcc« beispielsweise steht für einen Compiler (»gcc«), der Code für einen ARM-Prozessor generiert. Der Namensbestandteil »CROSS_COMPILE = arm-linux-« etwa weist das Kernel-Build-System dazu an, eine Cross-Generierung mit diesem Werkzeug durchzuführen – vorausgesetzt die Pfadvariable schließt das Verzeichnis ein, in dem sich der gewünschte Compiler befindet.

Nicht unbedingt notwendig, aber hilfreich ist schließlich die Variable »O«, die das Ausgabeverzeichnis bestimmt. Damit ist es möglich, Konfigurations- und Generierungs-Informationen vom eigentlichen Quellcode des Kernels zu trennen. Liegen die Kernelquellen beispielsweise im Verzeichnis »/usr/src/linux-3.0/« und gibt es ein Verzeichnis »/usr/src/linux-3.0-arm/«, lässt sich der Kernel mit dem folgenden Befehl konfigurieren, wobei das GUI-Tool Menuconfig zum Einsatz kommt (siehe Abbildung 2):

```
cd /usr/src/linux-3.0/
make ARCH=arm ➥
CROSS_COMPILE=arm-linux- ➥
O=/usr/src/linux-3.0-arm menuconfig
```

Das nächste Kommando veranlasst anschließend die Übersetzung:

```
cd /usr/src/linux-3.0/
make ARCH=arm ➥
CROSS_COMPILE=arm-linux- ➥
O=/usr/src/linux-3.0-arm bzImage modules
```

Das Kernel-Build-System legt den übersetzten Kernel in »/usr/src/linux-3.0-arm/arch/arm/boot/zImage« ab.

Leider liegen auf dem Weg zum funktionierenden Kernel noch ein paar Stolpersteine aus, zunächst die korrekte Konfiguration: ARM-basierte Plattformen sind deutlich vielgestaltiger als die bekannte x86-Plattform (siehe Kasten „Advanced Risc Machine (ARM)“).

Das Kernel-Build-System bietet zum Ausgleich immerhin eine Reihe von Default-Konfigurationen für verschiedenste ARM-

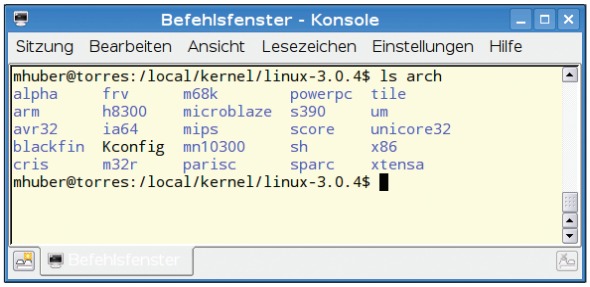


Abbildung 1: Im Verzeichnis »arch« des Kernel-Quellcodes finden sich die von Linux unterstützten Architekturen.

Systeme an. Welche das sind, lässt sich mit Hilfe des Aufrufs

```
cd /usr/src/linux-3.0/
make help ARCH=arm
```

anzeigen und durch »make *Default-Konfiguration*« aktivieren. Die zweite Stolperfalle sind die Cross-Entwicklungswerkzeuge selbst. Auf eventuell vorhandene vorkompilierte Pakete der gerade verwendeten Distribution zurückzugreifen ist frustrierend. Ebenso wenig Erfolg versprechend ist es, Pakete für Compiler und Linker gemäß diverser Anleitungen aus dem Internet selbst zu generieren.

Lust statt Frust

Zum Glück gibt es mit dem Tool »buildroot« jedoch einen nahezu narrensicheren Weg, an eine funktionierende Werkzeugkette zu kommen. Mehr noch, quasi als Geschenk generiert Buildroot gleich den Kernel und erzeugt zudem noch ein Root-Dateisystem. Buildroot ist nämlich eine Sammlung von Skripten und Programmen, mit denen man auf sehr einfache Weise ein eigenes Linux-System für verschiedenste Zielplattformen konfigurieren und erzeugen kann.

Nach der Konfiguration lädt Buildroot die aktuellen Quellcode-Pakete sämtlicher Werkzeuge von deren Ursprungsserver, patcht diese wenn notwendig, generiert sie und benutzt sie, um damit das komplette Linux – einschließlich Bootloader, Kernel und Root-Dateisystem – anzufertigen (siehe Abbildung 3).

Buildroot wird vom Uclib-Projekt gehostet. Dieses Projekt ist für den Quellcode einer C-Bibliothek verantwortlich, die für den Einsatz in eingebetteten Systemen optimiert ist und auch in Buildroot zum Einsatz kommt. Von der Projektwebseite lässt sich das Buildroot-Paket herunterladen [1] und in einem Verzeichnis, beispielsweise dem Home Directory auspacken. Das alles und auch die weiteren Aktionen können unter einer normalen Benutzer-ID erfolgen; Superuser-Rechte sind nicht erforderlich.

Ein »make help« – im Hauptverzeichnis von »buildroot/« aufgerufen – listet unter anderem die unterstützten Plattformen auf, immerhin 16 in der Version »buildroot-2011.08«. Für die ersten Versuche bietet sich die Plattform »qemu_arm_versatile« an, denn damit lässt sich das erzeugte System mit Hilfe des Emulators Qemu gleich auf dem Entwicklungsrechner testen. Die passende Konfiguration erzeugt der Befehl »make qemu_arm_versatile_defconfig«.

Konfigurieren

Per »make menuconfig« lässt sich die voreingestellte Version anpassen. Interessant ist es, unter dem Punkt »Kernel | Kernel version« an Stelle von »2.6.38.8« als aktuelleren Kernel »3.0« einzustellen. Außerdem sollte man unter dem Menüpunkt »System Configuration | Port to run a getty (login prompt) on« an Stelle von »ttyAMA0« lieber »tty1« einstellen. Unter »Build options | Number of jobs to run simultaneously« lässt sich die Anzahl der Prozessoren angeben, die die Entwicklungsmaschine zur Verfügung stellt. Wer also beispielsweise einen Hexacore sein Eigen nennt, trägt hier »6« ein. Nach dem Speichern der Konfiguration startet das Kommando »make« den Vorgang der Generierung.

Als Erstes lädt Buildroot über das Internet die Quellcodepakete der benötigten Komponenten herunter. Diese landen im Buildroot-Unterverzeichnis »dl/«. Anschließend generiert Buildroot die Toolchain und mit deren Hilfe sowohl

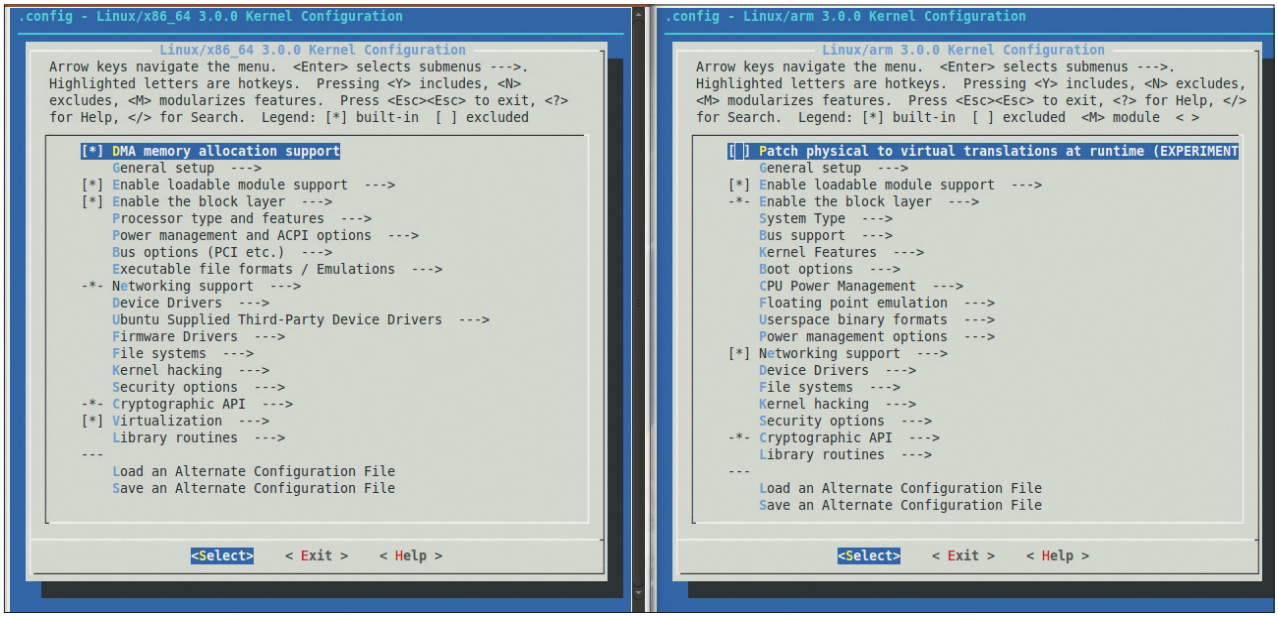


Abbildung 2: Je nach Plattform zeigt Menuconfig bereits im Hauptmenü unterschiedliche Konfigurationsoptionen (links x86_64, rechts ARM).

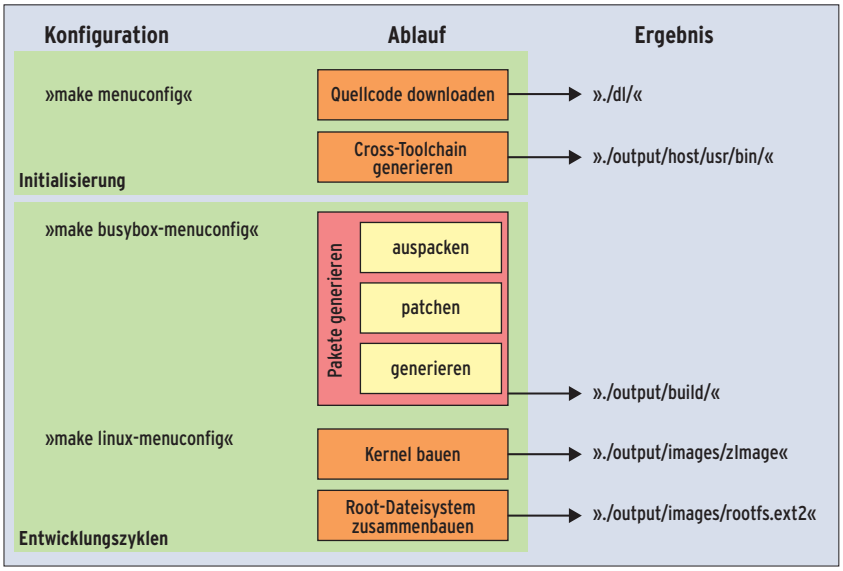


Abbildung 3: In mehreren Schritten erstellt Buildroot ein lauffähiges Linux-System.

das Root-Filesystem als auch den Kernel. Jetzt ist Zeit für eine ausgiebige Kaffeepause, denn das Übersetzen aller Komponenten kann abhängig von der Leistungsstärke der eingesetzten Hardware ohne Weiteres eine Stunde in Anspruch nehmen. Alle Ergebnisse des Prozesses landen im Unterverzeichnis »output« – unter »output/images« erscheinen das Root-Filesystem (»rootfs.ext2«) und der Kernel (»zImage«).

Für einen Test, ob Kernel und Root-Dateisystem funktionstüchtig sind, ist ein Emulator nützlich, der mit Qemu aber nicht für alle Plattformen bereitsteht. Wichtig ist es, mindestens die Qemu-Version 14.0 zu verwenden, denn ältere Versionen verhindern in einigen Fällen einen korrekten Bootvorgang.

Das folgendem Kommando bootet das neue System in Qemu:

```
cd ~/buildroot-2011.08/
qemu-system-arm -M versatilepb ➥
-m 128 -kernel output/images/zImage ➥
output/images/rootfs.ext2 ➥
-append "root=/dev/sda"
```

Die Option »-M« spezifiziert die zu emulierende Hardwareplattform, »-m« die Größe des zur Verfügung stehenden Hauptspeichers (hier 128 MByte), die Angabe »-kernel« bestimmt den Kernel, »-append« die dem Kernel zu übergebenden Parameter. Hier ist es notwendig, dem Kernel zumindest das Gerät mitzuteilen, über das er auf die Rootpartition zugreifen soll.

Geht alles gut, erscheint ein neues Fenster mit einem Login. Als User »root« eingeloggt (siehe Abbildung 5), steht dem Anwender ein rudimentäres System im Konsolenmodus zur Verfügung. Dieses kann er per Buildroot-Konfiguration und Neugenerierung umfassend an die eigenen Bedürfnisse anpassen. Um bei-

Advanced Risc Machine (ARM)

Eine der zurzeit wichtigsten Prozessorarchitekturen firmiert unter dem Namen Advanced Risc Machine (ARM). Von den im Jahr 2010 nach Schätzungen rund 13 Milliarden verkauften (Mikro-)Prozessoren verarbeiteten wohl beinahe 50 Prozent den ARM-Befehlssatz ([2], [3]) Fast alle Android-Smartphones und Tablets oder auch Navigationsgeräte sind typischerweise mit mehr als einer ARM-CPU realisiert, ebenso Apples Mobilgeräte. Das liegt daran, dass diese Architektur hohe Leistung bei niedrigem Stromverbrauch bietet.

Die Entwicklung begann bereits 1983, der erste Prozessor stand 1987 mit dem ARM 2 zur Verfügung. Es handelte sich schon damals um einen 32-Bit-Prozessor, der sich mit erstaunlich wenigen Transistoren und damit Silizium bauen lässt. Bei der implementierten Risc-Architektur handelt es sich um eine so genannte 3-Register-Maschine, die das Verknüpfungsergebnis der Inhalte zweier Register in einem dritten Register ablegt.

Im Laufe der Zeit hat der Hersteller verschiedene Befehlssätze spezifiziert und implementiert, unter anderem mit Jazelle auch einen, der auf die Verarbeitung von Javacode spezialisiert ist. ARM-Prozessoren haben allerdings in Normalausführung keine Floating Point Unit.

Bei der Namensgebung der Prozessoren hat ARM den großen Vorbildern Intel und AMD nachgeeifert, sie ist – mit einem Wort gesagt – unübersichtlich. Es werden zunächst vier Architekturen (Befehlssätze) unterschieden:

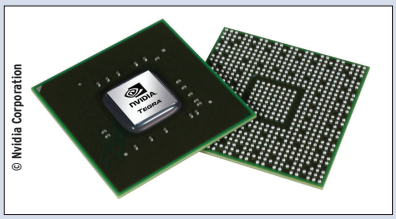


Abbildung 4: ARM-CPUs wie Nvidias Tegra 2 finden sich in Smartphones und Tablets.

ARMv4, ARMv5, ARMv6 und ARMv7. Letztere ist aktuell. Darüber hinaus gibt es etwa sechs Familien: Arm7, Arm9, Arm11, Cortex R, Cortex M und Cortex A. Der Namensvorsatz Cortex scheint reines Marketing zu sein, M steht dagegen für Mikrocontroller, also eine CPU samt notwendigster Peripherie, R für Realtime und A für Application.

Außerdem werden noch Implementierungsvariationen an den Architekturnamen gehängt. T etwa steht für einen zusätzlichen, Speicherplatz sparenden Befehlssatz (Thumb), D für Hardware-technische Debugmöglichkeiten, M für Multiply-Enhanced und I für eine eingebettete ICE-Makrozelle.

In den aktuellen Topmodellen der Smartphones befindet sich als High-End-Prozessor ein Cortex A9. Diese Version ist mehrkernfähig. Typischerweise bauen die Hersteller damit Dualcore-Prozessoren auf, beispielsweise den in den aktuellen Tablets vorwiegend eingesetzten Tegra von Nvidia (Abbildung 4).

Besonders interessant ist, dass ARM selbst keine Prozessoren herstellt, sondern nur das Design erledigt. Die Firma verdient ihr Geld allein auf Basis von Lizenzen. Die Lizenznehmer, Qualcomm, Nvidia, Apple und diverse andere, nehmen das Basisdesign, erweitern oder modifizieren es und bauen dann die hauseigenen Prozessoren daraus. Ob ARM seine Vormachtstellung im Segment der Mobilgeräte halten kann, ist noch fraglich. Denn der Hunger nach Rechenleistung und Speicherausbau wird immer größer und ARM hat noch keine 64-Bit-Versionen angekündigt.

Unübersichtlich wie die Welt der ARM-Prozessoren ist auch der architekturspezifische Quellcode im Linux-Kernel. Linus Torvalds hat daher zu einer Aufräumaktion aufgerufen. Erste Patches sind in Kernelversion 3.0 eingeflossen.

spielsweise den Kernel zu konfigurieren, gibt der Benutzer im Hauptverzeichnis von Buildroot das Kommando »make linux-menuconfig« ein. Viele weitere Möglichkeiten sind in der Buildroot-Dokumentation aufgeführt, die im HTML-Format im Unterverzeichnis »docs« von Buildroot liegt.

Wer Buildroot lediglich verwendet, um damit an Cross-Entwicklungswerkzeuge zu kommen, muss wissen, dass sich die gewünschten Programme nach einem Generierungslauf im Buildroot-Unterverzeichnis »output/host/usr/bin/« befinden. Entweder erweitert der Benutzer seine »PATH«-Variable um diesen Pfad oder er befüllt »CROSS_COMPILE« mit »/home/user/buildroot-2011.08/output/host/usr/bin/arm-linux-«. Um das Eingeben der Parameter »CROSS_COMPILE« und »ARCH« bei jedem Make-Aufruf zu vermeiden, kann man diese auch als Environment-Variable ablegen:

```
cd /usr/src/linux-3.0/
export CROSS_COMPILE=arm-linux-
export ARCH=arm
```

Achtung: Diese Variablen sind für nachfolgend aufgerufene Programme nur in jener Shell sichtbar, in der sie auch exportiert wurden.

Weniger ist mehr

Um einen möglichst kompakten Kernel zu bekommen, ist es im Übrigen sinnvoll, von einer minimalen Konfiguration auszugehen. Dazu dient der Aufruf »make allnoconfig«. Anschließend lassen sich per »make menuconfig« genau jene Treiber und Subsysteme aktivieren, die benötigt beziehungsweise von der Hardware unterstützt werden. Ein weiterer Vorteil bei diesem Vorgehen ist die kurze Generierungszeit: Auf einem schnellen Rechner vergeht kaum eine Minute, bis ein solch kompakter Kernel entstanden ist, und selbst auf einem Netbook bleibt die Wartezeit erträglich.

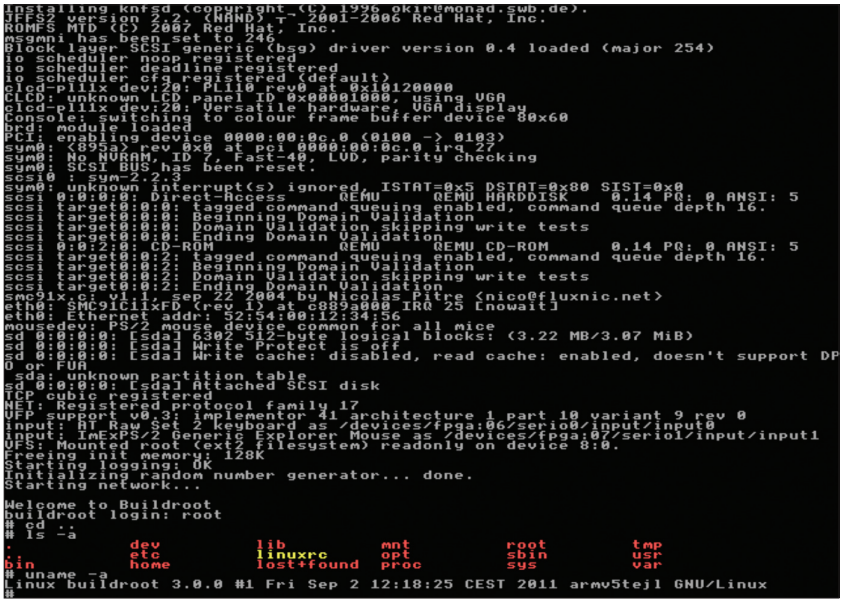


Abbildung 5: Das von Buildroot erzeugte Linux-System benötigt nur wenige Sekunden zum Booten.

Mein Linux

Dank Buildroot und der Emulatorsoftware Qemu lässt sich in kurzer Zeit ein eigenes Linux-System aufbauen, und das nicht nur für die bekannte x86-Plattform, sondern auch für ganz andere Architekturen. Das selbst kompilierte System bleibt überschaubar – wie in den Anfangstagen von Linux ist es möglich, jede installierte Komponente zu verstehen. Genau das Richtige also für den technikinteressierten Linux-Fan. *(mhu)* ■

Infos

[1] Download-Adresse von Buildroot: [http://buildroot.uclibc.org/download.html]
[2] Automotive Microcontrollers Market Report, August 2010: [http://www.electronics.ca/publications/products/Automotive-Microcontrollers-Market-Report.html]
[3] Heise Online, „ARM freut sich über kräftig sprudelnde Lizenzeinnahmen", Oktober 2010: [http://www.heise.de/ct/meldung/ARM-freut-sich-ueber-kraeftig-sprudelnde-Lizenzeinnahmen-1125958.html]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux-Fans von Open Source. Mittlerweile ist die dritte Auflage ihres Buches „Linux Treiber entwickeln" erschienen.

Root-Dateisystem

Für ein funktionstüchtiges Linux ist neben dem Kernel noch ein Root-Dateisystem erforderlich. Es enthält die zum Betrieb notwendigen Programme, Bibliotheken und Konfigurationsdateien. Buildroot erstellt das Root-Filesystem als eine Datei, die per Loop-Device gemountet wird. Dadurch lässt sie sich auch nachträglich beliebig modifizieren.

Die zentrale Komponente des von Buildroot erstellten Dateisystems bildet das Multicall-Binary Busybox. Es vereint in einem einzigen Executable eine Shell mit allen wichtigen Systemkommandos, Editor, Logging und – falls konfiguriert – sogar Web- und FTP-Server. Der Name Multicall-Binary rührt daher, dass dieses einzelne Executable mit Hilfe vieler symbolischer Links unter allen benötigten Programm- und Kommandonamen ansprechbar ist. Busybox erkennt beim Aufruf, unter welchem Namen es gestartet wurde, und nimmt dann wie ein Chamäleon die zugehörige Funktionalität an.

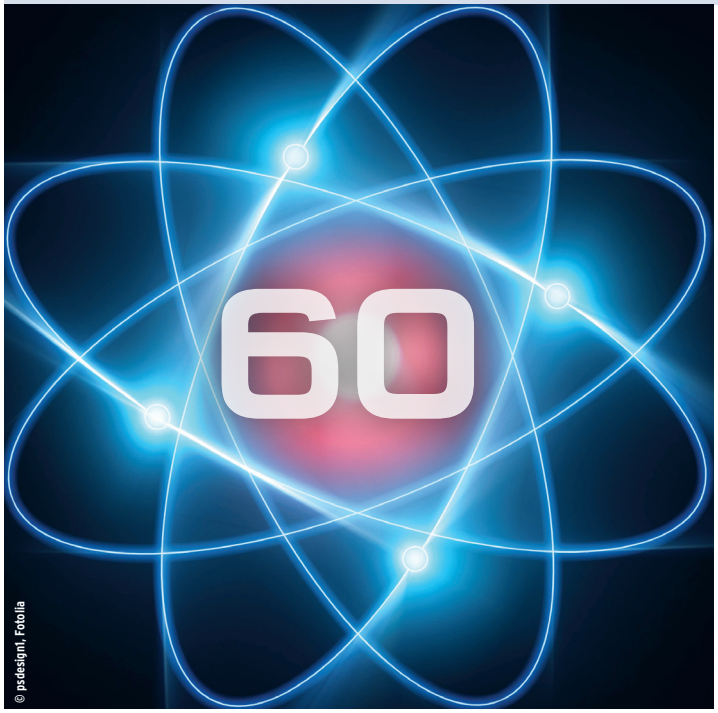
Kein Wunder, dass dieses Schweizer Taschenmesser auch das Init-Programm (System-V-Init), den ersten vom Kernel beim Booten gestarteten Prozess, im generierten Linux stellt. Die Konfiguration von Init befindet sich im Root-Dateisystem unter »/etc/inittab«, die zugehörigen Skripte, die zum Beispiel das Netzwerk konfigurieren, unter »/etc/init.d/«.

Als Gerätedateien finden sich typischerweise »/dev/console«, »/dev/null« und »/dev/zero« im Dateisystem, weitere Einträge lassen sich nach Bedarf anlegen. Außerdem braucht das Root-Filesystem noch ein temporäres Verzeichnis (»/tmp/«) und ein Logverzeichnis (»/var/log/«). Sinnvollerweise legt man zudem Mountpoints für die virtuellen Dateisysteme »/proc/« und »/sys/« an. Das Mounten selbst erledigt eines der per Init gestarteten Skripte.

Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 60

Kern-Technik

Den Kern eines laufenden Betriebssystems zu debuggen war und ist problematisch. Doch der Emulator Qemu ermöglicht plattformunabhängiges Kernel- und Modul-Debugging auf Hochsprachen-Niveau. Dank der virtuellen Maschine braucht der Kernelhacker noch nicht einmal einen zweiten Rechner. Jürgen Quade, Eva-Katharina Kunst



© psdesignt, Fotolia

Zu den Basisoperationen eines Debuggers zählt das Einfrieren von Codesequenzen und das nachfolgende Analysieren von Speicherinhalten. Gehören die Codesequenzen zu einer Applikation, ist Debugging vergleichsweise unproblematisch. Wird aber der Kernel selbst eingefroren, steht keine Ablaufumgebung mehr zur Verfügung, die Tastatureingaben entgegennimmt, Daten auf den Monitor zaubert, auf Speicherinhalte zugreift oder die den Kernel später weiterarbeiten lässt. Kernel-Debugging hat etwas von einem Arzt, der sich selbst operieren möchte.

Technisch ist das Problem einfach zu lösen, indem man komplexe Funktionen auf ein zweites System auslagert. Das hat ja üblicherweise ein funktionierendes Speicher- und Dateimanagement und übernimmt das Durchwühlen des Quellcodes nach Variablen, Datenstrukturen, Funktionen und Codezeilen. Für den zu debuggenden Kernel wird nur noch ein so genannter Debugserver benötigt, der nur einfache Kommandos wie etwa Speicherzellen lesen und schreiben oder Breakpoints setzen in dem zu untersuchenden System ausführt.

Der in der vorigen Kern-Technik [1] vorgestellte Emulator Qemu enthält einen solchen Debugserver (siehe Kasten „Varianten des Kernel-Debugging“). Kommt zudem wieder die Systemgenerierungs-Software Buildroot [2] zum Einsatz, ist Kernel-Debugging vergleichsweise leicht umzusetzen – Voraussetzung ist ein mit Symbolinformationen ausgestatteter Kernel.

Dank Buildroot ist das aber kein Problem: Binnen Kurzem stellt das Tool ein übersichtliches Userland und einen schlanken Kernel zur Verfügung, der sich rasch umkonfigurieren und bearbeiten lässt. Alle Schritte sind in dem Kasten „Kurzanleitung“ zusammengefasst. Zunächst lädt der Anwender Buildroot herunter und entpackt das Archiv. Danach erstellt er die Default-Konfiguration – diesmal am besten für ein x86-System – mit »make qemu_x86_defconfig«.

Vier Optionen sind nach dem anschließenden Aufruf von »make menuconfig« anzupassen: Unter »Toolchain« aktiviert der Anwender die Option »Build gdb for the Host«, unter »Kernel | Kernel version« gibt er »3.1« ein, unter »System Configuration | Port to run getty (login prompt) on« trägt er »tty1« ein und unter »Build options | Number of jobs to run simultaneously«, die Anzahl der Prozessorkerne der Generierungsmaschine.

Ein weiteres »make« stößt den ersten Generierungslauf an, er installiert unter anderem die Kernelquellen, deren Konfiguration der Benutzer allerdings später noch einmal bezüglich Kernel-Debugging anpassen muss.

Debugging-Kernel

Dazu ruft der Anwender im Buildroot-Hauptverzeichnis »make linux-menuconfig« auf. Die relevanten Einstellungen im daraufhin angezeigten Menü »Optionen« befinden sich unterhalb des Punktes »Kernel hacking« (siehe Abbildung 1). Erforderlich sind die Optionen »Kernel debugging« und »Compile the kernel with debug info«. Ein weiteres »make« generiert einen Kernel mit der angepassten Konfiguration.

Das Ergebnis sind im Wesentlichen zwei Dateien: Im Hauptverzeichnis findet sich die Datei »vmlinux«, die nicht nur den Code, sondern auch die zugehörigen Debuginfos enthält. Im Architektur-Unterverzeichnis – für die x86-Plattform ist das beispielsweise »arch/x86/boot/« – liegt der komprimierte Kernel als »bzImage«. Auf anderen Plattformen heißt der Kernel schon mal »zImage«.

Bootloader wie beispielsweise Grub brauchen den komprimierten Kernel (»bzImage«), der Debugger selbst benötigt zwar ebenfalls das Image des Kernels, greift jedoch auf das unkomprimierte und

mit Debuginfo versehene Pendant »vmlinux« zurück. Selbstverständlich benötigt der Debugger auch Zugriff auf den Quellcode.

Wer mit Buildroot Kernel und Root-Filesystem generiert hat, sollte beides zunächst ohne Debugging ausprobieren:

```
qemu -kernel output/images/bzImage➥
 -hda output/images/rootfs.ext2➥
 -append "root=/dev/sda rw"
```

Funktioniert alles, kann das Debuggen durch Anhängen von »-s« und »-S« beginnen. Die Option »-s« startet den Debugserver (»gdbserver«), »-S« hält den Kernel gleich zu Beginn an:

```
qemu -kernel output/images/bzImage➥
 -hda output/images/rootfs.ext2 -append➥
 "root=/dev/sda rw" -s -S
```

Damit der GNU-Debugger (GDB) die zum Kernel gehörenden C- und Headerdateien findet, startet der Anwender das Tool aus dem Quellcodeverzeichnis des Linux-Kernels heraus (siehe Abbildung 2). Wer ein x86-System gebaut hat, kann den auf dem Entwicklungsrechner vorinstallierten GNU-Debugger einsetzen, ansonsten kommt der für das Hostsystem gebaute und unterhalb des Buildroot-Verzeichnisses »output/host/usr/bin/« abgelegte GDB zur Verwendung:

```
cd output/build/linux-3.1/
gdb
```

Das Kommando »gdb« startet die Debugger-Sitzung. Als Erstes lädt der Benutzer Code und Symbole des Kernels mit Hilfe des Befehls »file vmlinux«. Wer nun die Meldung »no debug-symbols found« zu sehen bekommt, muss die Kernelkonfiguration bezüglich der Debug-Optionen überprüfen und gegebenenfalls den Linux-Kernel neu kompilieren. Inklusive der Symbole ist »vmlinux« mehr als 40 MByte groß.

Stopp und Start

Die Verbindung zum Debugserver stellt das Kommando »target remote :1234« her (Abbildung 2). Danach kontrolliert »gdb« die weitere Ausführung. Ein »continue« aktiviert das Linux-Gastsystem, das Tastenkürzel [Strg] + [C] unterbricht

Kurzanleitung

Kernelquellen und GCC sowie Buildroot, Qemu und GDB sind die Zutaten für erfolgreiches Kernel-Debugging. Die folgende Anleitung fasst wie ein Kochrezept alle Arbeitsschritte für das richtige Setup zusammen. Die Feinheiten beschreibt der Artikel.

1. Buildroot herunterladen und auspacken

```
wget http://buildroot.uclibc.org/download/buildroot-2011.08.tar.bz2
tar xvfj buildroot-2011.08.tar.bz2
```

2. Buildroot konfigurieren und System generieren

```
cd buildroot-2011.08
make qemu_x86_defconfig
make menuconfig
  [build-option, Kernel-Version 3.1, gdb, tty1]

make
make linux-menuconfig
 [debug info]
make
```

3. Modul kompilieren und ins Rootverzeichnis kopieren

```
cd driver
export CROSS_COMPILE=...
export ARCH=...
make
cp module.ko .../output/target/root/
```

4. Root-Dateisystem neu generieren

```
cd buildroot-2011.08
make
```

5. System mit Qemu und Debugserver starten

```
qemu -kernel output/images/bzImage -hda output/images/rootfs.ext2 -append "root=/dev/
sda rw" -s -S &
```

6. Debugger starten

```
gdb

file vmlinux
target remote :1234
continue
```

7. Einloggen, Treiber laden und Speicheradressen identifizieren

```
insmod Modul.ko
cat /sys/module/hello/sections/.text
cat /sys/module/hello/sections/.data
cat /sys/module/hello/sections/.bss
```

8. Modul-Symbolinformationen laden

```
add-symbol-file Modul.ko Addr_text
    -s .data Addr_data
    -s .bss Addr_bss
```

9. Debuggen: Breakpoints setzen etc.

Tabelle 1: Die wichtigsten GDB-Kommandos

Kommando	Bedeutung
add-symbol-file	Zusätzliche Symboldatei laden
break	Breakpoint setzen
bt	Aufrufreihenfolge der Funktionen ausgeben
continue	Programm fortsetzen
del	Breakpoint löschen
detach	Debugger vom Debugserver abmelden
file	Code (Programm) laden
help	Informationen zu den implementierten Funktionen ausgeben
info	Diverse Informationen anzeigen
next	Codezeile abarbeiten
print	Ausgabe von Variablen, Datenstrukturen und Speicherzellen
quit	GDB beenden
step	Bei der Abarbeitung Unterfunktionen mit debuggen
target	Verbindung zum Debugserver herstellen

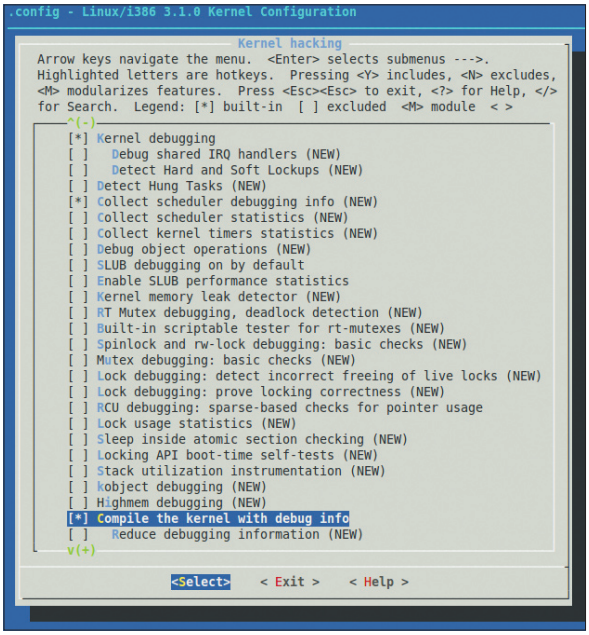


Abbildung 1: Die zum Debugging notwendigen Kerneloptionen finden sich unter dem Menüpunkt »Kernel hacking«.

die Abarbeitung. Abbildung 2 zeigt, wie das GDB-Kommando »break vfs_mknod« einen Breakpoint auf die Funktion »vfs_mknod« setzt.

Ausgang

Sobald der Anwender auf dem Linux-System das Kommando »mknod /dev/hello 254 0« aufruft, stoppt die Ausführung und Variablen lassen sich untersuchen. Um den Programmablauf fortzusetzen, gibt er das Kommando »continue« ein. Um den Debugger vom Linux-System zu trennen, drückt er zunächst [Strg] + [C]. Anschließend löst das GDB-Kommando »detach« die Verbindung zum Server, »quit« beendet den Debugger (weitere Kommandos siehe Tabelle 1).

Listing 1: Makefile

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m := hello.o
03
04 else
05 PWD   := $(shell pwd)
06 KDIR  := ~/buildroot-2011.08/output/build/linux-3.1/
07
08 default:
09         $(MAKE) -C $(KDIR) M=$(PWD) modules
10 endif
11
12 clean:
13         rm -rf *.ko *.o *.mod.c *.mod.o modules.order
14         rm -rf Module.symvers .*.cmd .tmp_versions
```

Auch Kernelmodule lassen sich auf Hochsprachen-Niveau mit Qemu debuggen. Doch muss dafür die Unterstützung für Module in der Kernelkonfiguration aktiviert sein (»Enable loadable module support« mit dem Unterpunkt »Module unloading«). Bei dem von Buildroot erzeugten Kernel ist hier also eine Neukonfiguration und Neugenerierung erforderlich.

Da zudem die Adresse des Modulcodes im Hauptspeicher nicht im Vorhinein bekannt ist, muss der Anwender sie nach dem Laden des Moduls identifizieren und dem Debugger mitteilen. Er findet sie in den Einträgen des »/sys«-Dateisystems.

Vor dem Debuggen steht zunächst noch das Übersetzen des Moduls für den von Buildroot erzeugten Kernel. Am einfachsten geschieht das mit einem angepassten Makefile. Dabei muss die Variable »KDIR« auf den Pfad der zu verwendenden Kernelquellen zeigen, der sich im vorliegenden Fall unterhalb des Buildroot-Verzeichnisses befindet.

Sollte das zu debuggende Linux-System im Qemu nicht für eine x86-Architektur sein, sind wie in [1] beschrieben zusätzlich die Environment-Variablen »CROSS _COMPILE« und »ARCH« zu setzen.

Zum Debuggen des Moduls muss der Inhalt von Listing 1 als »Makefile«, der von Listing 2 als »hello.c« in einem eigenen Ordner unterhalb des Buildroot-Verzeichnisses liegen. Eventuell ist noch der Pfad zu den in der Variablen »KDIR« abgelegten Linux-Quellen zu korrigieren. Das angepasste Makefile baut das Modul »hello.ko«, das der Anwender anschließend in das Root-Filesystem kopiert, etwa mit dem Kommando:

```
cp hello.ko ../output/target/root/
```

Der Aufruf von »make« im Buildroot-Verzeichnis erzeugt das Root-Filesytem neu. Damit steht das Modul im System nach dem Booten im Homeverzeichnis des Superusers zur Verfügung. Am einfachsten ist es, bei dem Aufruf von Qemu die Option »-S« wegzulassen, »-s« aber zu verwenden. Dadurch ist zwar der Debugserver aktiviert, Linux fährt aber dennoch direkt hoch.

Nach dem Einloggen mit dem Login-Namen »root« lädt der Anwender das Modul per »insmod hello.ko« (Abbildung 3). Die folgenden Befehle ermitteln die Adressen

```
quade@felicia:~/buildroot-2011.08/output/build/linux-3.1>gdb
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) file vmlinux
Reading symbols from /home/quade/buildroot-2011.08/output/build/linux-3.1/vmlinu
x...done.
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
default_idle () at arch/x86/kernel/process.c:393
393                     current_thread_info()->status |= TS_POLLING;
(gdb) break vfs_mknod
Breakpoint 1 at 0xc108d4b0: file fs/namei.c, line 2429.
(gdb) continue
Continuing.

Breakpoint 1, vfs_mknod (dir=0xd740b06c, dentry=0xd7424a00, mode=8612,
    dev=265289728) at fs/namei.c:2429
2429    {
(gdb) bt
#0  vfs_mknod (dir=0xd740b06c, dentry=0xd7424a00, mode=8612, dev=265289728)
    at fs/namei.c:2429
#1  0xc108d780 in sys_mknodat (dfd=-100, filename=0xbf87eeb4 "/dev/hello",
    mode=8612, dev=265289728) at fs/namei.c:2503
#2  0xc108d7f7 in sys_mknod (filename=0xbf87eeb4 "/dev/hello", mode=8630,
    dev=64768) at fs/namei.c:2522
#3  <signal handler called>
#4  0xb772615b in ?? ()
Cannot access memory at address 0x1ba
(gdb)
```

Abbildung 2: Der GDB wird aus dem Linux-Quellcodeverzeichnis gestartet, damit hat er Zugriff auf die C-Dateien.

```
Welcome to Buildroot
buildroot login: root
# insmod hello.ko
# cat /sys/module/hello/sections/.text
0xd8817000
# cat /sys/module/hello/sections/.data
0xd88170e0
# cat /sys/module/hello/sections/.bss
0xd8817294
#
```

Abbildung 3: Nach dem Einloggen als Root lädt der Anwender unter Qemu das Kernelmodul und ermittelt die Adressen von Code- und Datensegmenten.

des Code- sowie der beiden Daten-Segmente des Kernelmoduls:

```
cat /sys/module/hello/sections/.text
cat /sys/module/hello/sections/.data
cat /sys/module/hello/sections/.bss
```

Da das mit Buildroot erzeugte Linux-System keinen Udev-Support bietet, ist außerdem mit dem Kommando »mknod /dev/hello c 254 0« die Gerätedatei anzulegen, über die eine Applikation Zugriff auf den Beispieltreiber erhält. Ob auf dem System die Major-Nummer 254 verwendet wird, ermittelt die Befehlszeile »cat /proc/devices | grep Hello«.

Den Debugger auf dem Hostsystem startet der Benutzer wie gehabt aus dem Quellcodeverzeichnis des Linux-Kernels. Nach Eingabe der Kommandos »file vmlinux« und »target remote :1234« hält Qemu das Linux-System an.

Das Kommando »add-symbol-file« teilt dem System die zuvor im Sys-FS ermittelten hexadezimalen Adressen des Code- und der beiden Daten-Segmente des Moduls mit:

```
add-symbol-file Pfad/hello.ko 0xd8817000
 -s .data 0xd88170e0 -s .bss 0xd8817294
```

Jetzt kann der Anwender bereits einen Breakpoint setzen, etwa auf die Funktion »driver_read()«. Ein »continue« lässt das Linux-System weiterarbeiten.

Gibt er nun das Kommando »cat /dev/hello« ein, aktiviert sich die Modulfunktion »driver_read()« und GDB hält den

Varianten des Kernel-Debugging

Linux bietet mit Qemu, dem KGDB und dem KDB drei Möglichkeiten, Kernel- und Modul-Code zu debuggen (Abbildung 4). Während Qemu seinen Debuggingserver selbst mitbringt und keine besondere Kernelunterstützung benötigt, hat Linus Torvalds vor drei Jahren zähneknirschend den In-Kernel-Debugserver »kgdb« akzeptiert. Wie bei der Qemu-Lösung läuft der eigentliche Debugger auf einem zweiten Rechner, dem Debughost. Die Kommunikation zwischen Server und Debugger findet über eine serielle Verbindung statt. Steht weder ein zweiter Rechner noch die serielle Verbindung zur Verfügung, kann mit ein paar Tricks eine Virtualisierungslösung wie Virtualbox aushelfen und das gewünschte Interface emulieren.

Ansonsten hat Torvalds in Version 2.6.35 mit dem »kdb« ein Frontend für den »kgdb« in den Standardkernel integriert, das einfache Operationen wie Lesen und Setzen von Speicherstellen sowie das Anzeigen der im System instanzierten Rechenprozesse ermöglicht – und das auf demselben System. Der »kdb« wird bei entsprechender Konfiguration aktiv, sobald der Linux-Kernel abstürzt. Alternativ lässt er sich über die Tastenkombination [Alt]+[S-Abf]+[G] aufrufen. Hochsprachen-Debugging ist damit jedoch nicht möglich.

Diese In-Kernel-Debugger stellt eine der nächsten Kern-Technik-Folgen ausführlicher vor.

DAS MAGAZIN FÜR UBUNTU-ANWENDER!
DigiSub-Mini*: 2 digitale Ausgaben Ubuntu User!



PDF DOWNLOAD

5€ FÜR 2 AUSGABEN

IHRE VORTEILE

- Hilfe für Ein- und Umsteiger bei den ersten Schritten mit Ubuntu
- Zusatzwissen rund um Ubuntu und seine Derivate
- 2x Testen ohne Risiko, das DigiSub-Mini ist jederzeit kündbar!
- Nutzbar auf Notebook und PC, Tablet oder Smartphone!

JETZT GLEICH BESTELLEN!

Tel.: 07131 / 2707 274 Fax: 07131 / 2707 78 601
URL: www.ubuntu-user.de/abo E-Mail: abo@ubuntu-user.de

*geht ohne Kündigung in ein digitales Jahresabo mit 4 Ausgaben pro Jahr über und ist jederzeit kündbar!

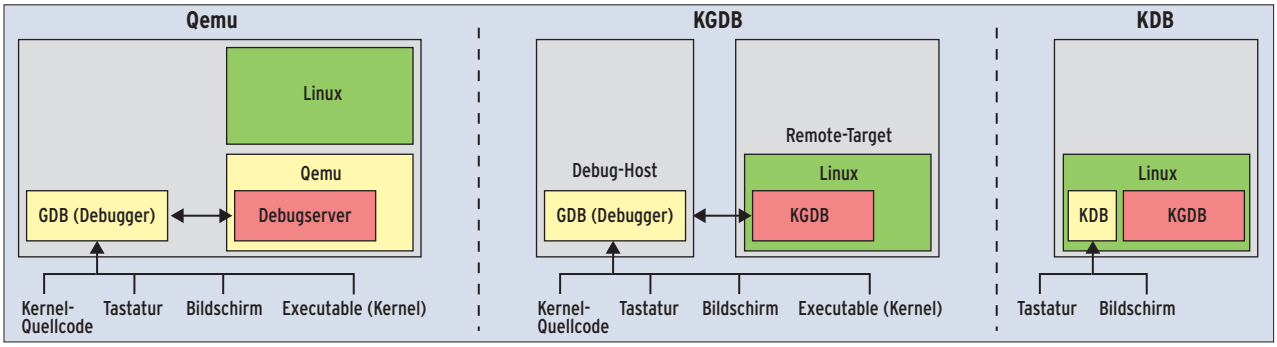


Abbildung 4: Mit dem Emulator Qemu, mit KGDB und KDB offeriert Linux mehrere Möglichkeiten, Kernel- und Modul-Code zu debuggen.

Kernel wegen des gesetzten Breakpoints an. Jetzt lassen sich Speicherzellen des Moduls untersuchen und die Bearbeitung schrittweise durchspielen.

Allerdings ist verwirrend, dass der Debugger scheinbar unmotiviert zwischen den Zeilen hin und her springt. Der Zugriff auf lokale Variablen weist auf die Ursache hin: »value optimized out« – der Compiler hat den Kernelcode optimiert. Dadurch sind einige der definierten Variablen im Debugger nicht sichtbar, Codefragmente sind umgemodelt. Dass innerhalb des Kernels einige Makros im Einsatz sind, vereinfacht die Fehlersuche ebenfalls nicht. Nicht selten entpuppt sich im Linux-Kernel eine Variable als ausgefuchstes Makro. Kernel-Debugging bleibt also eine Herausforderung, der man sich mit viel Geduld und Übung stellen muss. *(mhu)* ■

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Mittlerweile ist die dritte Auflage ihres Buches „Linux-Treiber entwickeln" erschienen.

Infos:

[1] Quade, Kunst, „Kern-Technik", Folge 59: Linux-Magazin 11/11, S. 96

[2] Downloadadresse von Buildroot: [http://buildroot.uclibc.org/download.html]

Listing 2: Modul »hello.c«

```
01 #include <linux/fs.h>
02 #include <linux/cdev.h>
03 #include <linux/device.h>
04 #include <asm/uaccess.h>
05
06 static char hello_world[]="Hello World\n";
07
08 static dev_t hello_dev_number;
09 static struct cdev *driver_object;
10 static struct class *hello_class;
11 static struct device *hello_dev;
12
13 static ssize_t driver_read( struct file *instanz,
14     char __user *user, size_t count, loff_t *offset )
15 {
16     unsigned long not_copied, to_copy;
17
18     to_copy = min( count, strlen(hello_world)+1 );
19     not_copied=copy_to_user(user,hello_world,to_copy);
20     return to_copy-not_copied;
21 }
22
23 static struct file_operations fops = {
24     .owner= THIS_MODULE,
25     .read= driver_read,
26 };
27
28 static int __init mod_init( void )
29 {
30     if (alloc_chrdev_region(&hello_dev_number,0,1,"Hello")<0)
31         return -EIO;
32     driver_object = cdev_alloc();
33     if (driver_object==NULL)
34         goto free_device_number;
35     driver_object->owner = THIS_MODULE;
36     driver_object->ops = &fops;
37     if (cdev_add(driver_object,hello_dev_number,1))
38         goto free_cdev;
39     hello_class = class_create( THIS_MODULE, "Hello" );
40     if (IS_ERR( hello_class )) {
41         pr_err( "hello: no udev support\n");
42         goto free_cdev;
43     }
44     hello_dev = device_create( hello_class, NULL,
45         hello_dev_number, NULL, "%s", "hello" );
46     return 0;
47 free_cdev:
48     kobject_put( &driver_object->kobj );
49 free_device_number:
50     unregister_chrdev_region( hello_dev_number, 1 );
51     return -EIO;
52 }
53
54 static void __exit mod_exit( void )
55 {
56     device_destroy( hello_class, hello_dev_number );
57     class_destroy( hello_class );
58     cdev_del( driver_object );
59     unregister_chrdev_region( hello_dev_number, 1 );
60     return;
61 }
62
63 module_init( mod_init );
64 module_exit( mod_exit );
65 MODULE_LICENSE("GPL");
```

Das Runtime-Powermanagement des Linux-Kernels

Einer schläft

Der Linux-Kernel versetzt einzelne Geräte in den Stromsparzustand, wenn er sie gerade nicht benötigt. Dank des Runtime-Powermanagement-Subsystems läuft das gut organisiert ab. Jürgen Quade, Eva-Katharina Kunst


© Stanislav Duben, 123RF

Linux versucht auch ohne Zutun von Distribution und Benutzer den Stromverbrauch zu minimieren. Hinter dem Schlagwort „Leistungsanpassung“ steht die Idee, genau so viele Ressourcen zur Verfügung zu stellen, wie die Anwendungen momentan benötigen. Das Betriebssystem taktet Prozessoren herunter (Frequency Scaling) und versorgt sie mit möglichst geringer Spannung (Voltage Scaling).

Selektiv abschalten

Nicht jede Hardwarekomponente unterstützt diese dynamische Anpassung, einige lassen sich aber zumindest aus- und bei Bedarf wieder einschalten. Damit dies koordiniert und zum richtigen Zeitpunkt passiert, stellt der Linux-Kernel neben dem Schlafenlegen des gesamten Rechners (System Sleep, siehe „Kern-Technik“ in diesem Linux-Magazin) seit 2009 das Runtime-Powermanagement (Runtime-PM) zur Verfügung (Abbildung 1). Dabei handelt es sich im Wesentlichen um ein Framework, das die Hauptverantwortung für das Powermanagement dem eigentlichen Gerätetreiber übergibt [1].

Das ist durchaus sinnvoll, denn das Schlafenlegen und Aufwecken sowie das damit verbundene Einsparpotenzial sind gerätespezifisch. Im Gerätetreiber findet sich der zugehörige Code erstens bei der Initialisierung, zweitens bei den Zugriffsfunktionen (beispielsweise »driver_read()« und »driver_write()«) und drittens in Form der Funktionen »runtime_idle()«, »runtime_suspend()« und »runtime_resume()«.

Bei der Initialisierung des Treibers konfiguriert der Entwickler den stromsparmäßigen Grundzustand für das Gerät (»suspended« oder »active«, Abbildung 2). Außerdem kann er noch Timeouts angeben, nach denen das Gerät bei Inaktivität „suspendiert“ werden soll.

Die für das Runtime-PM modifizierten Zugriffsfunktionen schließlich setzen den Mechanismus in Gang, der das zugehörige Gerät schlafen legt oder auch aufweckt (Abbildung 3). Dazu haben die Kernelentwickler Zugriffsfunktionen wie »driver_read()« oder »driver_write()« um den Aufruf von Hilfsfunktionen erweitert, die dem Kernel mitteilen, dass das Gerät benötigt oder nicht mehr benötigt wird (siehe Tabelle 1: „Wichtige Hilfsfunktionen des Runtime-PM“).

Technisch betrachtet ist der Mechanismus simpel: »pm_runtime_get()« erhöht den Zähler »usage_count«, der die auf das Gerät zugreifenden Instanzen zählt. Ist der Zähler ungleich null und befindet

Tabelle 1: Wichtige Hilfsfunktionen des Runtime-PM

pm_runtime_disable()	Runtime-PM für das Gerät deaktivieren
pm_runtime_enable()	Runtime-PM für das Gerät aktivieren
pm_runtime_get()	Gerät belegen (asynchron)
pm_runtime_get_sync()	Gerät belegen (synchron)
pm_runtime_put()	Gerät freigeben (asynchron)
pm_runtime_put_sync()	Gerät freigeben (synchron)
pm_runtime_resume()	Gerät aufwecken
pm_runtime_set_active()	Zustand auf »active« setzen
pm_runtime_set_autosuspend_delay(struct device *dev, int delay)	Timeout-Zeit setzen
pm_runtime_set_suspended()	Zustand auf »suspended« setzen
pm_runtime_suspend()	Gerät in den Low-Power-Status versetzen

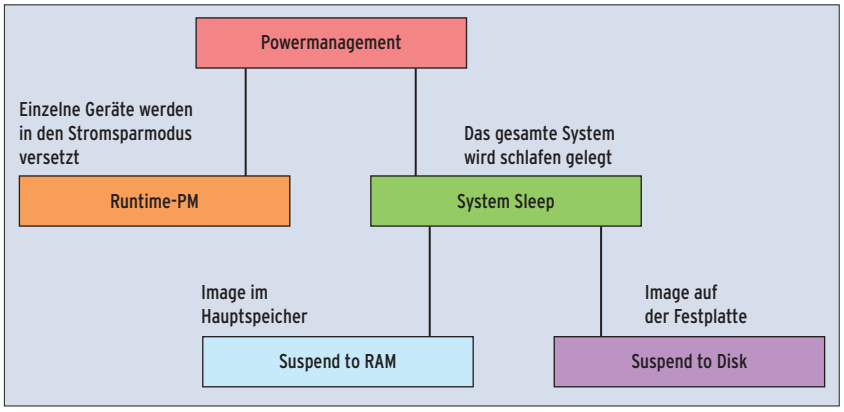


Abbildung 1: Zwei eigenständige Subsysteme implementieren im Linux-Kernel das Powermanagement.

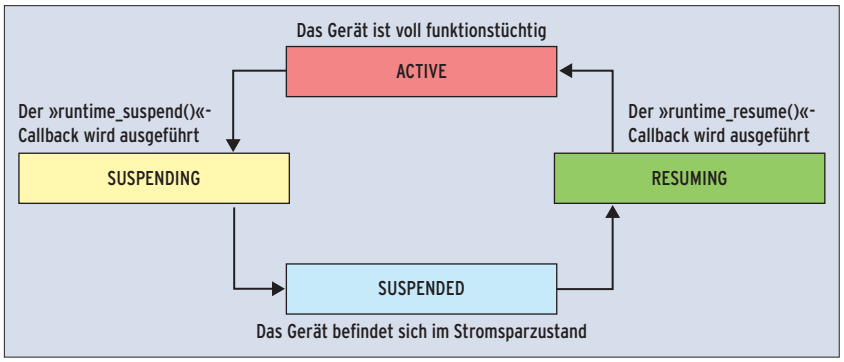


Abbildung 2: Das Runtime-PM eines Geräts durchläuft vier Zustände.

sich das Gerät im Schlafzustand, triggert die Funktion das Aufwecken. »pm_runtime_put()« dekrementiert den Instanzenzähler wieder und ruft – falls der Zähler null ist und keine Instanz auf das Gerät zugreift – die erwähnte Funktion »runtime_idle()« auf, die das Gerät in den Stromsparmodus versetzt.

Allerdings werden beim Aufruf von »pm_runtime_get()« die Funktionen »runtime_resume()« und »runtime_suspend()« des zugehörigen Treibers nicht direkt abgearbeitet. Vielmehr reiht der PM-Kern Aufträge zum Aufruf der Funktionen in eine Workqueue ein, in der sie ein Kernelthread abarbeitet. Dieser asynchrone Weg weist in den meisten Umgebungen Performancevorteile auf. Daneben gibt es aber auch den synchronen Weg über die Funktionen »pm_runtime_get_sync()« und »pm_runtime_put_sync()«.

Die Funktion »runtime_suspend()« versetzt das Gerät in den Stromsparmodus. Dazu rettet sie den aktuellen Hard-

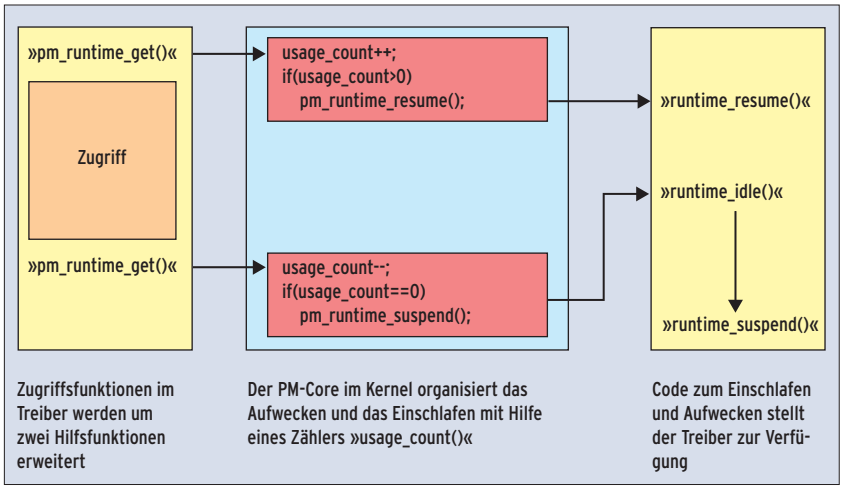
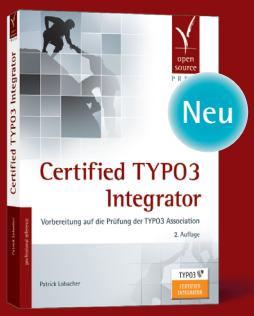


Abbildung 3: Zugriffe auf die Geräte rahmt der Programmierer durch Hilfsfunktionen ein.

opensourcepress.de

open source PRESS

BÜCHER



ISBN 978-3-941841-62-8
384 Seiten · brosch. · € 39,90 [D]



ISBN 978-3-941841-50-5
224 Seiten · brosch. · € 29,90 [D]



ISBN 978-3-941841-61-1
ca. 300 Seiten · brosch. · € 29,90 [D]



ISBN 978-3-941841-42-0
328 Seiten · brosch. · € 29,90 [D]

opensourceschool.de

open source SCHOOL

TRAINING

	Programmierung
27.–28.02.	C++: STL und Boost
05.–07.03.	SCJP/OCP Java 6 Programmer
12. –13.03.	Lua
16. –18.04.	Einstieg in C++
19. –20.04.	Qt für Fortgeschrittene
	Webentwicklung
08.–09.03.	TYPO3 für Administratoren
19. –21.03.	TYPO3 für Webdesigner
26.–27.03.	jQuery
11. – 13.04	ExtJS
07.–09.05	HTML5
	Administration/Netzwerk
19. –23.03.	Vorbereitung auf LPIC-2
26.–28.03.	Administration von OTRS
16. –20.04.	Vorbereitung auf LPIC-1
23.–25.04.	IPv6
11. – 13.06.	Bacula Grundlagen

weitere Themen und Termine:
opensourceschool.de/kurstermine

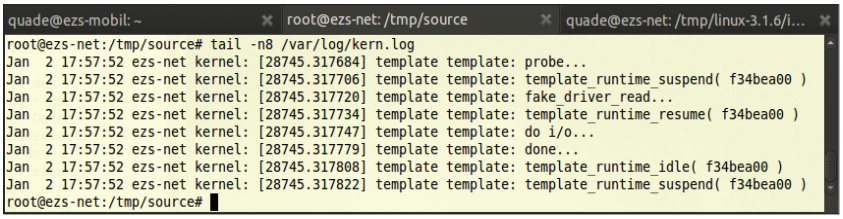


Abbildung 4: Das Kernel-Log zeigt: Für den Zugriff aktiviert (»resume«) Linux das Gerät und versetzt es anschließend wieder in den Stromsparzustand (»suspend«).

warezustand, aktiviert – falls vom Gerät unterstützt – das Wakeup aus dem Low-Power-State und fährt schließlich die Stromversorgung und/oder Taktgeneratoren herunter beziehungsweise koppelt das Gerät von diesen ab. Typischerweise ruft Linux »runtime_suspend()« nicht direkt auf, sondern durch »runtime_idle()«. Letzteres überprüft erst mal, ob ein Schlafenlegen überhaupt sinnvoll ist und zu einer Energieersparnis führt.

Die Funktion »runtime_resume()« ist für das Wiederherstellen des Hardwarezustands, das Deaktivieren des Wakeup-Signaling und das Hochfahren der Stromquellen und Taktgeneratoren zuständig. Nach dem Abarbeiten dieser Funktion kann das System wieder normal auf das Gerät zugreifen. Den erfolgreichen Zustandswechsel melden die Funktionen über den Rückgabewert 0.

Der Entwickler muss für Wakeup-Signaling übrigens die dazu benötigte Interrupt-Serviceroutine selbst implementieren, die ihrerseits die Funktion »runtime_resume()« triggert.

Erst die Kinder

Die Adressen der Suspend-, Resume- und Idle-Funktionen werden zunächst einer Power-Domain zugeordnet, die wiederum mit dem Geräteobjekt (»struct device«) verknüpft ist. Power-Domains (in Kernel 3.1 über das Objekt »struct dev_pm_domain«, in früheren Versionen über »struct dev_power_domain« implementiert) repräsentieren die kontrollierbaren Strom- und Taktgeber für eine Gruppe von Geräten oder auch nur für ein einzelnes Gerät. Erst wenn alle Geräte innerhalb einer Power-Domain »idle« sind, fährt Linux die Strom- und Taktgeber herunter.

Das Geräteobjekt (»struct device«) wird für jedes Gerät im Kernel benötigt und ist Teil des Gerätemodells. Dieses Modell reflektiert den hierarchischen Aufbau der Hardware und ist baumartig strukturiert. Der Kernel stellt sicher, dass er nur dann ein Gerät in den Low-Power-Mode versetzt, wenn alle abhängigen Geräte bereits schlafen. Ansonsten würde beispielsweise der USB-Hub schlafen gelegt, obwohl Skype noch auf die darüber angeschlossene Webcam zugreift.

Zusätzlich offeriert das Gerätemodell eine Schnittstelle zur Information und Konfiguration der Hardware, die auch vom Runtime-PM unterstützt wird (siehe Kasten „Konfiguration des Runtime-Powermanagements“).

Listing 1 zeigt auszugsweise die Implementierung von Runtime-Powermanagement in einem Treiber. Die Methode zum

Listing 1: Runtime-Powermanagement im Treiber

```
[...]
#ifdef CONFIG_PM_RUNTIME /* Powermanagement */
static int template_runtime_suspend(struct device *dev )
{
    dev_info(dev,"template_runtime_suspend( %p )\n", dev );
    return 0;
}

static int template_runtime_resume(struct device *dev)
{
    dev_info(dev,"template_runtime_resume( %p )\n",dev);
    return 0;
}

static int template_runtime_idle(struct device *dev)
{
    dev_info(dev,"template_runtime_idle( %p )\n",dev);
    if ( does_not_make_sense_to_suspend )
        return -EAGAIN;
    pm_runtime_suspend( dev );
    return 0;
}

//static struct dev_pm_domain template_pm_ops = { /* Kernel 3.1 */
static struct dev_power_domain template_pm_ops = {
    .ops = {
        .runtime_resume = template_runtime_resume,
        .runtime_suspend = template_runtime_suspend,
        .runtime_idle = template_runtime_idle,
    },
};
#endif

static void fake_driver_probe( struct device *dev )
{
    dev_info(dev,"probe...\n");
    pm_runtime_set_active( template_dev );
    pm_runtime_enable( template_dev );
    pm_runtime_suspend( dev );
}

static void fake_driver_read( struct device *dev )
{
    dev_info(dev,"fake_driver_read...\n");
    pm_runtime_get_sync( dev );
    dev_info(dev,"i/o...\n");
    pm_runtime_put( dev );
}

static int __init template_init( void )
{
[...]
#ifdef CONFIG_PM_RUNTIME /* Powermanagement */
    template_dev->pwr_domain = &template_pm_ops;
    //template_dev->pm_domain = &template_pm_ops; /* Kernel 3.1 */
#endif
[...]
```

Schlafenlegen, die hier keinen produktiven Code enthält, findet sich in Zeile 3; die Funktion zum Aufwecken in Zeile 9. In Zeile 15 steht die Funktion, die im Fall, dass das Gerät sinnvoll schlafen gelegt werden kann, das Schlafenlegen triggert. Die Initialisierung des Runtime-PM ist in den Zeilen 25, 34 und 54 zu sehen; »fake_driver_read()« in Zeile 42 symbolisiert eine Zugriffsfunktion. Der vollständige Code steht auf dem Server des Linux-Magazins unter [2] zum Download bereit.

Abbildung 4 zeigt die Ausgaben im Syslog nach dem Laden des aus Listing 1 erzeugten Moduls. Die Probe-Funktion im Treiber versetzt das (virtuelle) Gerät in den Stromsparzustand. Greift eine Applikation auf den Treiber zu, wacht das Gerät durch den Kernel getriggert auf. Dank synchroner Methode kann der Zugriff direkt erfolgen (symbolisiert durch die Ausgabe »do i/o« im Syslog). Nach erfolgtem Zugriff ruft der Kernel die Idle-Funktion auf, die ihrerseits das Schlafenlegen anstößt.

Schwieriges Zusammenspiel

Bezüglich der Basismechanismen ist das Runtime-Powermanagement von Linux durchaus überschaubar. Probleme treten erst im Zusammenspiel mehrerer Subsysteme und Hardwarekomponenten zutage, die unter Umständen benötigt werden, um einzelne Komponenten in den Low-Power-State zu überführen.

Das hat die Problematik um das Active State Power Management (ASPM) für PCI-Express-Geräte gezeigt. Die Kernelentwickler hatten ASPM in Linux 2.6.38 deaktiviert, da es Rechner mit bestimmter Hardware einfrieren ließ. Dadurch liefen die Komponenten in einem Modus, der die maximale Energie verbrauchte. Das betraf unter anderem die Anwender der Ubuntu-Release Natty Narwhal, die einen 10 bis 30 Prozent höheren Energieverbrauch verzeichneten als mit der Vorgängerversion Maverick Meerkat [3].

Nach einer Zwischenlösung mit dem Argument »pcie_aspm = force« für den Kernel hat sich die Situation bereits durch angepasste Treiber in Kernel 3.2 gebessert. Die Überarbeitung des ASPM-Codes von Red-Hat-Entwickler Matthew Garrett wird allerdings erst in Kernel 3.3 einziehen. *(mhu)*

Infos

[1] Rafael Wysocki, Alan Stern, „Runtime Power Management Framework für I/O Devices“: [https://www.kernel.org/doc/Documentation/power/runtime_pm.txt]

[2] Quellcode zum Artikel: [http://www.linux-magazin.de/static/listings/magazin/2012/03/powermanagement/]

[3] Launchpad Bug #760131, „Power consumption raised significantly in Natty“: [https://bugs.launchpad.net/ubuntu/+source/linux/+bug/760131]

[4] Rafael Wysocki, „Runtime Power Management Framework“: Vortrag auf der Linuxcon Japan 2011, [https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_wysocki2.pdf]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. In der Zwischenzeit ist die dritte Auflage ihres Buches „Linux-Treiber entwickeln“ erschienen.

Konfiguration des Runtime-Powermanagements

Zur Konfiguration des Runtime-Powermanagements legt der Kernel für jedes Gerät beziehungsweise für jede Geräteklasse, die Runtime-PM unterstützt, im Sys-Dateisystem ein Verzeichnis an: »/sys/devices/virtual/*Gerätename*/*Gerätename*/power/« [4]. Es enthält die jeweiligen Informations- beziehungsweise Konfigurationsdateien.

Die Datei »runtime_status« beispielsweise teilt den aktuellen PM-Zustand – »suspended« oder »active« – mit. Die Datei »control« schaltet das Runtime-Powermanagement für das Gerät an (»auto«) beziehungsweise aus (»on«). Die Einträge »runtime_suspended_time« und »runtime_active_time« haben die Entwickler für das Diagnosetool Powertop implementiert. Sie summieren die Zeiten auf, in denen sich das Gerät im Zustand »suspended« beziehungsweise »active« befand. Steht in der Datei »asyncn« der String »enabled«, wird bei einem System-Sleep das Gerät asynchron aufgeweckt oder auch schlafen gelegt. Steht dort »disabled« vollziehen sich die Übergänge zwischen den Zuständen synchron.

I:HOSTSERVER

Managed Server



Profitieren Sie von:

- ✓ ISO 9001 zertifiziertem Managed Hosting
- ✓ Hochsicherheits-Datacenter
- ✓ DE-CIX Direktanbindung
- ✓ IPv4/IPv6 Unterstützung
- ✓ 24/7 Service und Support

Professionelle Server

Individuelle Managed Server mit persönlichem und kompetentem Support. Für mehr Performance, Sicherheit und Verfügbarkeit, jeden Tag, rund um die Uhr.

Wir bieten Hostinglösungen vom Server bis zum Clustersystem inkl. Beratung, Planung und Service 24/7.

hostserver.de/server

TÜV SÜD ISO 9001

Managed Hosting zertifiziert nach ISO 9001:2008

I:HOSTSERVER

0 30 / 420 200 24 ■ hostserver.de

Berlin ■ Marburg ■ Frankfurt am Main

Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 61

Kern-Technik

Wer schläft, spart. Daher organisiert der Linux-Kernel das kollektive Schlafen der Rechnerkomponenten, bis das komplette System ruht. Er strukturiert Abhängigkeiten baumartig und verwendet ein Phasenmodell. Damit sind bei Bedarf alle Komponenten schnell wieder hellwach. Jürgen Quade, Eva-Katharina Kunst



© psdesignt, Fotolia

Es liegt nahe, das Runtime-Powermanagement (siehe Artikel „Einer schläft“ im Schwerpunkt dieses Linux-Magazins) zum gleichzeitigen Schlafenlegen aller Komponenten zu nutzen. Dadurch ließe sich recht einfach das bekannte Suspend to RAM oder das Suspend to Disk (Hibernation) realisieren.

Dem widersprechen jedoch die Kernelentwickler [1]: Für viele Komponenten mag es zwar kein Unterschied sein, ob sie alleine oder im Kollektiv Strom sparen, für andere aber sehr wohl: Je nach Zustand soll sich die Hardware beim Aufwachen und beim Aufwecken anders verhalten. Wake on LAN beispielsweise ließe sich in dem einen Fall ignorieren, im anderen soll es zum Aufwachen schlafender Komponenten führen.

Legt sich das ganze System schlafen, muss Linux zudem ein Abbild des Arbeitsspeichers ziehen und sichern. Diese Vorgänge benötigen mehr Zeit als das bloße Schlafenlegen einer einzelnen Komponente. Daher behandelt Linux den System Sleep (Suspend) getrennt vom Runtime-Powermanagement. Der Kernel ermöglicht und ermuntert den Entwickler jedoch dazu, die Suspend- und Resume-Funktionen der Runtime-Energieverwaltung auch für den Systemschlaf zu verwenden.

Basis des System-Sleep-Framework ist die Information, die im Gerätemodell – repräsentiert durch das Sys-Filesystem – abgelegt ist. Schwierig gerät das Schlafenlegen und Aufwecken nämlich dadurch, dass es nicht in beliebiger Reihenfolge erfolgen darf. Beispielsweise kann ein USB-Hub nicht vor den angeschlossenen Geräten in den Stromsparmodus gehen. Daher greift der Kernel auf den im Gerätemodell abgelegten, baumartig strukturierten Aufbau der gesamten Hardware zurück.

Beim System Sleep ist das Schlafenlegen von den Blättern hin zur Wurzel organisiert: erst das am USB-Hub angeschlossene Gerät, dann der USB-Hub selbst. Beim Aufwachen verläuft alles anders herum. Hier arbeitet Linux den Baum von der Wurzel zu den Blättern hin ab, der USB-Hub ist vor dem USB-Gerät aktiv.

Callback-Funktionen

Der Treiber implementiert im einfachen Fall das Schlafenlegen in einer »suspend()«-Funktion, die die Zustände der Hardware sichert und das Gerät in den Stromsparmodus schaltet. Das Aufwecken in der »resume()«-Funktion aktiviert das Gerät und restauriert den ursprünglichen Zustand.

Allerdings sind während der Abarbeitung dieser Funktionen Interrupts freigegeben, sodass es zu Inkonsistenzen kommen kann, wenn die schlafen zu legende Hardware zeitgleich einen Interrupt auslöst. Um auch mit solchen Fällen koordiniert umgehen zu können, lässt der Kernel

Kommunikation per Notifier

Das Notifier-Subsystem dient als zentrales Informations- und Ereignismanagement im Kernel. Ähnlich einem RSS-Feed versorgt der Kernel hierüber seine Abonnenten – also Subsysteme des Kernels und Gerätetreiber – mit Neuigkeiten, etwa über das Ändern des Prozessortakts oder auch über das Aktivieren von Powermanagement-Funktionen.

Für Letzteres sind insgesamt sechs Nachrichten definiert: »PM_HIBERNATION_PREPARE«, »PM_POST_HIBERNATION«, »PM_RESTORE_PREPARE«, »PM_POST_RESTORE«, »PM_SUSPEND_PREPARE« und »PM_POST_SUSPEND«. Treiber nutzen diese Nachrichten, um beispielsweise Firmware in den Hauptspeicher zu laden. Beim Aufwecken schreiben sie diese zurück in die Hardware.

Was die einzelnen Notifier bedeuten und wann sie verschickt werden, ist in der Dokumentation des Kernels genau nachzulesen [3]. Allgemeine Informationen und auch Codebeispiele zum Umgang mit Notifiern bietet [4].

das Einschlafen in mehreren Phasen ablaufen und bietet den Kernelobjekten an, je nach Phase unterschiedliche Callback-Funktionen zu aktivieren [2].

Beim Suspend to RAM (Abbildung 1) informiert der Notifier »PM_SUSPEND_PREPARE« zunächst interessierte Treiber und Systemkomponenten vom bevorstehenden Einschlafen (siehe Kasten „Kommunikation per Notifier"). Anschließend friert das System alle Rechenprozesse ein und ruft in hierarchischer Reihenfolge die »prepare()«-Callbacks auf. Treiber implementieren diese Funktionen, um das Hinzufügen neuer Geräte (Hotplugging) zu verhindern.

Nach den »prepare()«-Funktionen sind die beschriebenen »suspend()«-Callbacks an der Reihe. Der Kernel sperrt Geräte-Interrupts und ruft »suspend_noirq()«-Callbacks auf. Daraufhin schaltet Linux mit Ausnahme des ersten alle Rechnerkerne ab, sperrt die noch freigegebenen Interrupts und legt in Betrieb befindliche Teile schlafen.

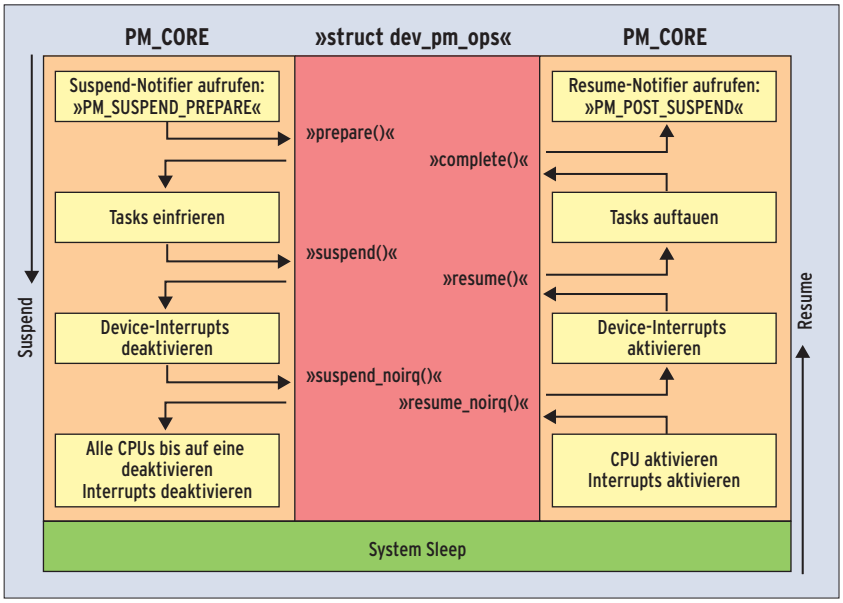


Abbildung 1: Beim Suspend to RAM durchläuft der Kernel vier Phasen, in die sich Treiberprogrammierer per Callbacks einklinken können.

Alles rückwärts

Das Aufwecken vollzieht sich in umgekehrter Reihenfolge. Erst gibt Linux die Systeminterrupts frei, dann aktiviert es die schlafenden Rechnerkerne. Der Kernel ruft die »resume_noirq()«-Callbacks auf, aktiviert die Geräte-Interrupts, ruft die »resume()«-Funktionen auf und taut die Rechenprozesse wieder auf. Zuletzt informiert der »PM_POST_SUSPEND«-Notifier die Subsysteme und Treiber über das Ende des Aufweckens.

Richtig komplex wird es beim Suspend to Disk. Schließlich sind nach dem Anfertigen des Hauptspeicher-Image noch die Treiber erforderlich, die das Image auf den Massenspeicher sichern. Dazu ruft der Kernel nach dem Informieren von Kernel-Subsystemen und Treibern per Notifier »PM_HIBERNATION_PREPARE« erst die »prepare()«-, »freeze()«- und »freeze_noirq()«-Callbacks auf (siehe Abbildung 2).

Damit sind Tasks und Geräte inaktiv, das Image kann erzeugt werden. Ist dieser Vorgang abgeschlossen, aktiviert Linux die Geräte zum Abspeichern des Image wieder. Hierzu ruft der Kernel die »thaw_noirq()«-, die »thaw()«- und schließlich die »complete()«-Callbacks auf. Ist das Image auf Massenspeicher geschrieben, versetzt er die aktiven Geräte tatsächlich in den Schlafzustand. Entsprechenden Code bringen Programmierer in den »prepare()«-, »poweroff()«- und »poweroff_noirq()«-Callbacks unter.

Um das System aus dem Suspend to Disk wieder in den Normalbetrieb zu überführen, ist sogar ein zweiter Kernel erforderlich – schließlich ist das System komplett stromlos gewesen. Theoretisch könnte der Bootloader das Image von Kernel und Hauptspeicher in den Speicher zurückbefördern, in der Praxis aber erweisen sich die Bootloader als zu unflexibel.

Daher startet ein zweiter Kernel als flexibler Bootloader, der den richtigen Kernel plus Image in den Hauptspeicher wuchtet. Bei diesem Vorgang sind einige der Hardwarekomponenten durch die Treiber im Bootloaderkernel bereits initialisiert, andere nicht. Um diesem Umstand gerecht zu werden, definiert der PM-Core für diesen Teil des Aufweckens eigene Callbacks: »restore_noirq()«, »restore()« und »complete()«. Innerhalb der Callbacks werden die Geräte in den ursprünglichen Zustand zurückversetzt.

Alle Funktionen – sowohl die des Runtime-Powermanagements als auch die des System Sleep – sind in der Datenstruktur »struct dev_pm_ops« zusammengefasst.

Advanced Configuration and Power Interface (ACPI)

Typischerweise ist auf den PC-Plattformen ACPI für das Schlafenlegen von Hardware zuständig. Der von HP, Intel, Microsoft, Phoenix und Toshiba im Jahr 1996 definierte Standard [5] legt verschiedene Zustände fest, unter anderem die Sleep-Zustände (S-States). Die S-States wiederum korrespondieren mit den Device-Power-States (D-States). Sie definieren nur Stromspar-, aber keine Leistungszustände. Ein Rechner ist also nicht betriebsbereit, wenn er sich in einem der Zustände S1 bis S5 befindet (siehe Tabelle 1).

ACPI ist im Bios in Form von Tabellen, aber auch in Form von Methoden abgelegt. Die Tabellen beschreiben die Hardware, während die Methoden Code darstellen, der zu bestimmten Zeitpunkten durch das Betriebssystem aufzurufen ist. Damit läuft dieser für die Betriebssystementwickler unbekannte Code im Kontext des Linux-Kernels.

Tabelle 1: S-States

Zustand	Beschreibung
S0	Betriebsbereit (Working)
S1	Schlafend, der CPU-Kontext bleibt erhalten
S2	Schlafend, der CPU-Kontext geht verloren
S3	Suspend to RAM (STR), die meiste Hardware ist abgeschaltet
S4	Suspend to Disk (STD), Zustand auf dem Hintergrundspeicher gesichert
S5	Ausgeschaltet

Die lässt sich über die Power-Domain an das Geräteobjekt binden, aber auch an ein Klassenobjekt oder in selteneren Fällen auch an ein Busobjekt. Sind die Callbacks an das Klassenobjekt gebunden, werden diese für jedes zur Klasse gehörende Gerät einmal mit der Adresse des jeweiligen Geräteobjekts als Parameter aufgerufen. Gibt es jedoch ein Callback sowohl im Klassenobjekt als auch im Geräteobjekt, favorisiert der Kernel die Geräteobjekt-Funktion.

Wer die Datenstruktur der Geräteklassenobjekte studiert, stellt fest, dass sich hier unabhängig von der Struktur »struct dev_pm_ops« Funktionen für Suspend und Resume einhängen lassen. Diese Elemente sind jedoch Relikte früherer Kernelversionen, die nur noch aus Kompatibilitätsgründen existieren. Kernelentwickler sind gehalten, das in diesem Artikel vorgestellte Interface zu verwenden.

Listing 1 zeigt die Unterstützung von System Sleep in einem Treiber für den Kernel 3.0, hier in Ubuntu 11.10, durch die beiden Funktionen »template_suspend()« (Zeile 12) und »template_resume()« (Zeile 18). Sobald der Treiber übersetzt und das erzeugte Modul per »insmod« in den Kernel geladen sowie System Sleep aktiviert ist, versetzt das Betriebssystem sämtliche Geräte der Geräteklasse »template« – im Beispiel nur eines – in den inaktiven Zustand.

Konfigurationssache

Der Kernelentwickler rahmt Powermanagement-Code übrigens mit »CONFIG_PM« ein, wie in den Zeilen 11 bis 23 und 47 bis 49 zu sehen ist. Dadurch werden die Programmfragmente nur dann übersetzt, wenn Powermanagement im Kernel konfiguriert ist. Zum Aktivieren von Suspend to RAM reicht es bei vielen Linux-Distributionen aus, den Deckel eines Net- oder Notebooks zu schließen. Alternativ kann Root Suspend to RAM aber auch über das Sys-Dateisystem durch das Kommando

```
echo "mem" >/sys/power/state
```

auslösen (siehe Kasten „Sys-Filesystem-Interface fürs Powermanagement“). Die Aktivitäten des Treibers und von System Sleep lassen sich anhand der Meldungen in »/var/log/kern.log« gut verfolgen (Abbildung 3). Da er sich zeitlich betrachtet als Letzter beim Gerätemodell angemeldet hat, wird der Treiber »template« als Erster suspendiert. Die Zeitstempel im Logfile werden erst beim Aufwecken ge-

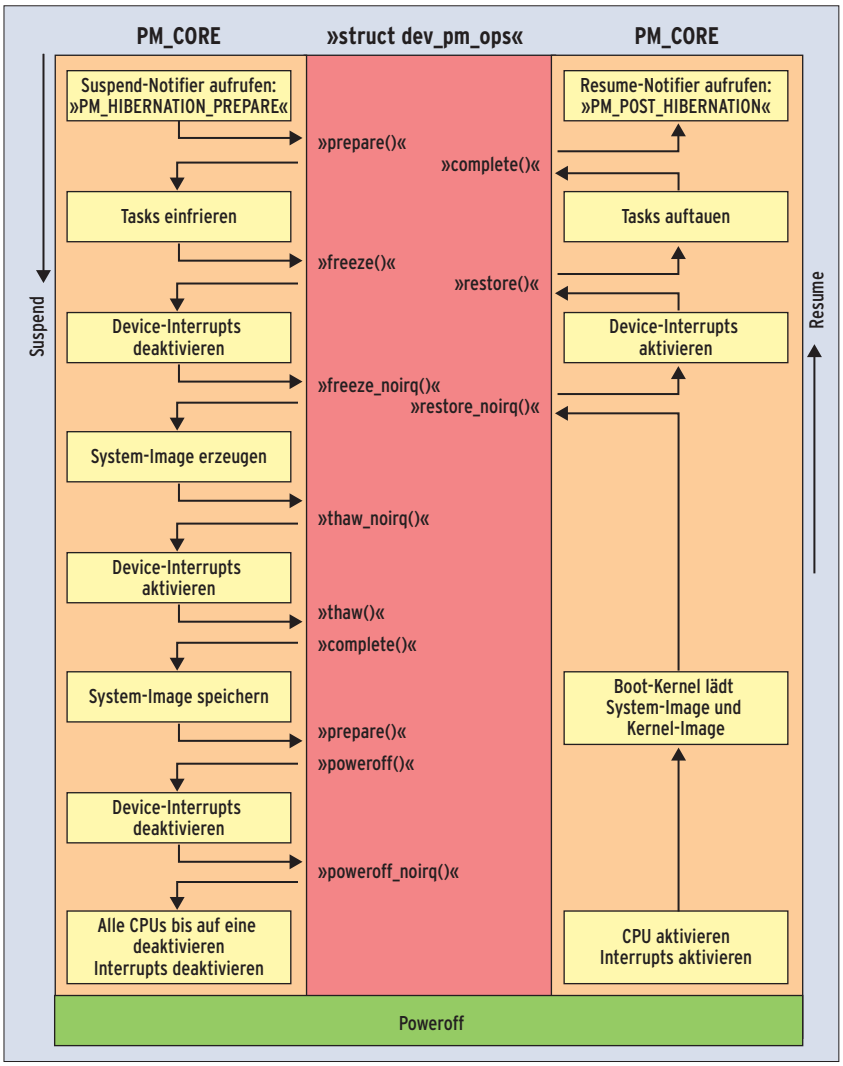


Abbildung 2: Suspend to Disk ist der kompliziertesteste aller Schlafzustände.

Infos

[1] Rafael Wysocki, „System Sleep vs. Runtime Power Management“: Vortrag auf der Linuxcon Japan 2011, [https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_wysocki.pdf]

[2] Rafael Wysocki, Alan Stern, „Device Power Management“: Dokumentation zum Kernel-Quellcode, [https://www.kernel.org/doc/Documentation/power/devices.txt]

[3] Rafael Wysocki, Alan Stern, „Suspend Notifiers“: Dokumentation zum Kernel-Quellcode, [https://www.kernel.org/doc/Documentation/power/notifiers.txt]

[4] Jürgen Quade, Eva-Katharina Kunst, „Kern-Technik“, Folge 46: Linux-Magazin 07/09, S. 90

[5] Hewlett-Packard et al., „Advanced Configuration and Power Interface Specification“: [http://www.acpi.info]

Sys-Filesystem-Interface fürs Powermanagement

Zur Konfiguration von System Sleep befinden sich im Sys-Filesystem im Verzeichnis »/sys/power/« Dateien, die teilweise dokumentiert, teilweise auch nur für die Kernelentwickler selbst vorgesehen sind. Die Datei »/sys/power/state« steuert den Powermanagement-Zustand. Schreibt man »mem« in diese Datei, führt das System ein Suspend to RAM aus, »disk« aktiviert Suspend to Disk.

Über die Datei »/sys/power/disk« wird Suspend to Disk konfiguriert, wobei sich hier auch ein Testmodus einstellen lässt. »/sys/power/image_size« legt eine Obergrenze für das zu erzeugende Image fest, wobei das Limit jedoch weich ist.

nommen; Tasks und auch der Syslog-Daemon sind zu Beginn des Einschlafvorgangs eingefroren.

Auf der PC-Plattform kommt bei System Sleep auch noch ACPI ins Spiel, das der Kasten „Advanced Configuration and Power Interface“ genauer beschreibt. Die Linux-Entwickler sind ziemlich unglücklich über ACPI, denn es handelt sich um eine Spezifikation von mehr als 600 Seiten. Viele ACPI-Implementierungen sind fehlerhaft, nicht gut dokumentiert und nicht Open Source. Zudem ist ACPI betriebssystemabhängig implementiert. Aus diesem Grund meldet sich auch Linux beim ACPI als Windows-Betriebssystem an. Nur so erreichen die Entwickler ein konsistentes Verhalten. Darüber hinaus gab es Bestrebungen, Linux durch Patente von der Verwendung von ACPI auszuschließen.

Powermanagement ist ein komplexer Vorgang, den alle Kernelkomponenten unterstützen müssen. Sowohl System Sleep als auch Runtime-Powermanagement zu implementieren hat sich unter den Treiberprogrammierern leider noch nicht zur Gänze herumgesprochen. Positiv formuliert: Der Linux-Kernel bietet noch reichlich Stromsparpotenzial. *(mhu)*

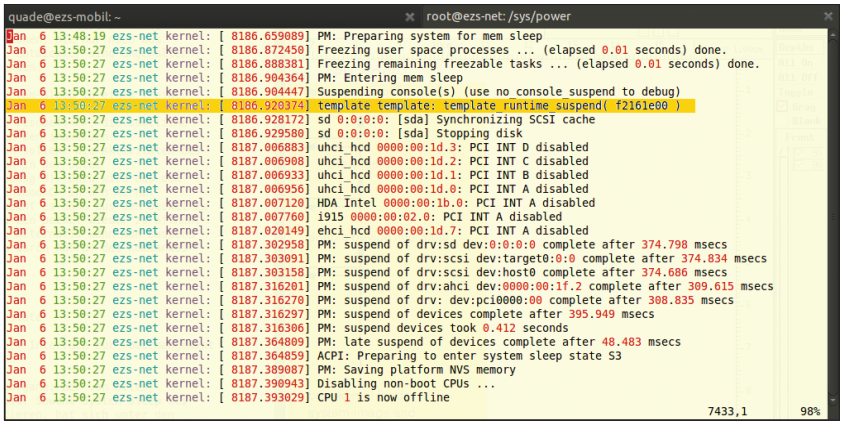


Abbildung 3: In der Logdatei »/var/log/kern.log« protokolliert System Sleep alle Transitionen.

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. In der Zwischenzeit ist die dritte Auflage ihres Buches „Linux Treiber entwickeln“ erschienen.

Listing 1: System Sleep im Treiber

```
01 #include <linux/fs.h>
02 #include <linux/cdev.h>
03 #include <linux/device.h>
04
05 #define TEMPLATE "template"
06 static dev_t template_dev_number;
07 static struct cdev *driver_object;
08 struct class *template_class;
09 static struct device *template_dev;
10
11 #ifdef CONFIG_PM /* Powermanagement */
12 static int template_suspend(struct device *dev )
13 {
14     dev_info(dev,"template_runtime_suspend( %p )\n", dev );
15     return 0;
16 }
17
18 static int template_resume(struct device *dev)
19 {
20     dev_info(dev,"template_runtime_resume( %p )\n",dev);
21     return 0;
22 }
23 #endif
24
25 static const struct dev_pm_ops template_class_pm = {
26     .suspend = template_suspend,
27     .resume  = template_resume,
28 };
29
30 static int __init template_init( void )
31 {
32     if( alloc_chrdev_region(&template_dev_number,0,1,TEMPLATE)<0 )
33         return -EIO;
34     driver_object = cdev_alloc();
35     if( driver_object==NULL )
36         goto free_device_number;
37     driver_object->owner = THIS_MODULE;
38     if( cdev_add(driver_object,template_dev_number,1) )
39         goto free_cdev;
40     template_class = class_create( THIS_MODULE, TEMPLATE );
41     if( IS_ERR( template_class ) ) {
42         pr_err("template: no udev support\n");
43         goto free_cdev;
44     }
45     template_dev = device_create( template_class, NULL,
46         template_dev_number, NULL, "%s", TEMPLATE );
47 #ifdef CONFIG_PM /* Powermanagement */
48     template_class->pm = &template_class_pm;
49 #endif
50     return 0;
51 free_cdev:
52     kobject_put( &driver_object->kobj );
53 free_device_number:
54     unregister_chrdev_region( template_dev_number, 1 );
55     return -EIO;
56 }
57
58 static void __exit template_exit( void )
59 {
60     device_destroy( template_class, template_dev_number );
61     class_destroy( template_class );
62     cdev_del( driver_object );
63     unregister_chrdev_region( template_dev_number, 1 );
64     return;
65 }
66
67 module_init( template_init );
68 module_exit( template_exit );
69 MODULE_LICENSE( "GPL" );
```