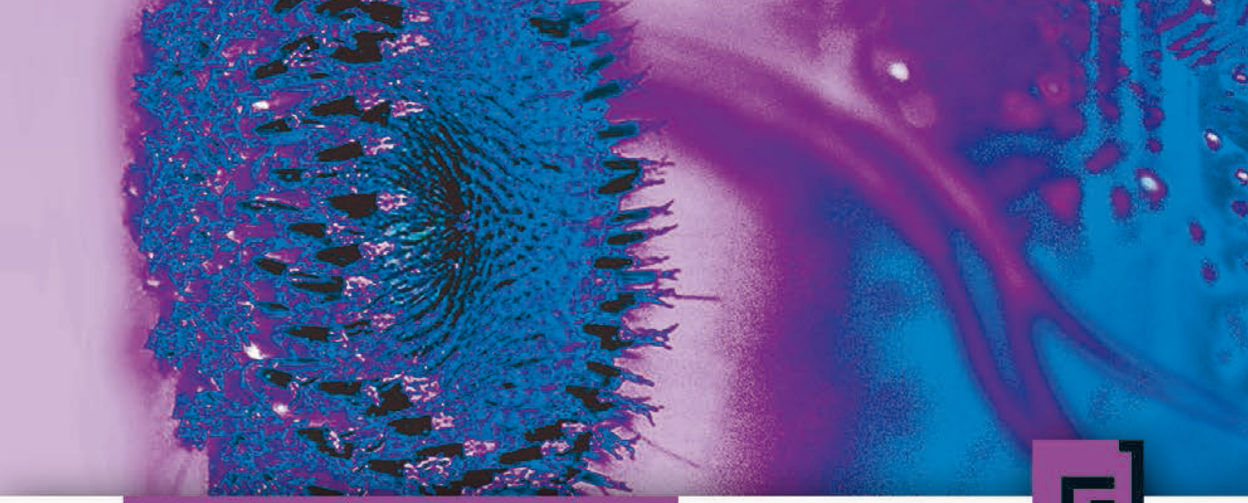


4., aktualisierte und erweiterte Auflage

Jürgen Quade · Eva-Katharina Kunst

Linux-Treiber entwickeln

Jetzt auch für Raspberry Pi

Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung

dpunkt.verlag

Jürgen Quade studierte Elektrotechnik an der TU München. Danach arbeitete er dort als Assistent am Lehrstuhl für Prozessrechner (heute Lehrstuhl für Realzeit-Computersysteme), promovierte und wechselte später in die Industrie, wo er im Bereich Prozessautomatisierung bei der Softing AG tätig war. Heute ist Jürgen Quade Professor an der Hochschule Niederrhein, wo er u.a. das Labor für Echtzeitsysteme betreut. Seine Schwerpunkte sind Echtzeitsysteme, Embedded Linux, Rechner- und Netzwerksicherheit sowie Open Source.

Eva-Katharina Kunst studierte Kommunikationswissenschaft an der LMU München sowie Wirtschaftsinformatik an der Fachhochschule München. Sie ist freiberuflich tätig als Journalistin. Ihre Arbeitsgebiete sind Open Source, Linux und Knowledge Management.

Jürgen Quade • Eva-Katharina Kunst

Linux-Treiber entwickeln

Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung – jetzt auch für Raspberry Pi

4., aktualisierte und erweiterte Auflage

dpunkt.verlag

Jürgen Quade
quade@hsnr.de

Eva-Katharina Kunst
eva.kunst@gmx.de

Lektorat: René Schönfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Da-TeX, Leipzig
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Media-Print Informationstechnologie, Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN
Buch 978-3-86490-288-8
PDF 978-3-86491-755-4
ePub 978-3-86491-756-1
mobi 978-3-86491-757-8

4., aktualisierte und erweiterte Auflage 2016

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Vorwort zur vierten Auflage

Professioneller, zugleich komplexer

Linux entwickelt sich unaufhaltsam weiter. Mehr Subsysteme, leichtere Portierbarkeit, besseres Realzeitverhalten und weitere Sicherheitsfunktionen. Die Liste lässt sich beliebig fortführen. In Version 4.x zeigt sich Linux professioneller, stabiler und sicherer, aber eben auch komplexer.

Einfacher per Raspberry Pi

Für den Entwickler wird es dadurch nicht einfacher. Gerade rechtzeitig hat daher Eben Upton 2012 den Raspberry Pi in den Markt gebracht. Das Besondere an diesem Minicomputer: Er ist mit dem Ziel entworfen worden, den Einstieg in die Computertechnik zu erleichtern und die Freude daran anzufachen. Und tatsächlich: Der Raspberry Pi vereinfacht vieles und eignet sich ausgezeichnet als Anschauungs- und Lehrobjekt. Problemlos kann unterschiedlichste Peripherie angeschlossen werden. Der Zugriff ist dabei ohne tiefgreifende Systemkenntnisse über beispielsweise PCI (Peripheral Interconnect) möglich.

Beispiele für PC und Raspberry Pi

Das macht sich die vierte Neuauflage zunutze und zeigt relevante Interfaces wie beispielsweise diejenigen für Interrupts oder für den GPIO-Zugriff anhand von Codebeispielen, die direkt auf dem Raspberry Pi ablauffähig und daher mit sehr wenig Aufwand testbar sind. Die vierte Neuauflage nutzt den Raspberry Pi darüber hinaus, um in die Kernelcodeentwicklung für und auf dem Raspberry Pi einzuführen (cross- und native Entwicklung) und neue Konzepte vorzustellen, die wie beispielsweise Device Trees die Portierbarkeit des Kernels erweitern. Damit trägt sie auch dem Umstand Rechnung, dass die Mehrheit aktueller Linux-Systeme milliardenfach (!) auf Prozessoren der Firma ARM (Advanced Risc Machine) ablaufen.

Neue Subsysteme

Neben der Aufnahme der neuen Themenbereiche Kernel- und Treibercodeentwicklung für eingebettete Systeme und Cross-Entwicklung ist die vierte Neuauflage um neue Subsysteme ergänzt. Außerdem sind die bekannten Interfaces auf den aktuellen Stand gebracht worden. Wie gehabt werden wichtige Kernelinterfaces und Softwaremechanismen anhand vieler Codebeispiele anschaulich und zugleich kompakt vorgestellt. In Kombination mit dem vermittelten theoretischen Unterbau

hilft damit auch das vorliegende Buch, strukturiert das für die Codeentwicklung notwendige Wissen in kurzer Zeit (anwendbar) aufzubauen. Der aktualisierte und deutlich erweiterte Anhang mit der Beschreibung von über 750 Kernelfunktionen lässt das Buch gleichzeitig ein wertvolles Nachschlagewerk sein.

Open Source hilft.

Doch es gibt auch Grenzen bezüglich Umfang und Aktualität. Ein Blick in den Quellcode zeigt, dass es eine Reihe weiterer Subsysteme und Kernelfunktionen gibt, die hier nicht beschrieben sind. Und da sich der Linux-Kernel unaufhaltsam weiterentwickelt, haben die Entwickler die eine oder andere Kernelfunktion wohl in der Zeit zwischen Drucklegung und Ihrem Lesen geändert. Oft reicht es aus, den Quellcode nur um die Angabe einer Header-Datei zu ergänzen, manchmal tauschen Sie aber auch einen Parameter aus, selten verschwinden ganze Funktionen. Im Fall von Problemen helfen zumeist das Internet und vor allem auch der Linux-Quellcode weiter. Denn das ist eines der größten Assets, die Linux zu bieten hat: den offenen Quellcode.

Danke

Wir möchten die Gelegenheit nutzen und unseren Lesern für die wohlwollende Aufnahme der ersten drei Auflagen, für Korrekturhinweise und Anregungen danken. Möge auch die vierte Auflage Ihnen eine große Hilfe sein. Ein Dankeschön geht wieder an den dpunkt.verlag und hier insbesondere an Herrn Schönfeldt, der dieses Buch als Lektor bei allen Auflagen ausgesprochen freundlich und konstruktiv begleitet hat!

Kempen, im Oktober 2015

Jürgen Quade und Eva-Katharina Kunst

Inhaltsverzeichnis

1 Einleitung

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

Zentrale Komponente für den HW-Zugriff

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

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

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

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

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

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

Reale und virtuelle Geräte

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

Kernelprogrammierung

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

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

Ziel dieses Buches ist es damit,

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

Scope

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

Ubuntu und Kernel 4.0.3

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

UP und SMP

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

Aufbau des Buches

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

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

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

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

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

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

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

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

Notwendige Vorkenntnisse

C-Kenntnisse

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

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

Zusätzliche Informationsquellen

Errata und Beispielcode zum Buch

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

Quellcode online

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

Dokumentation als Teil der Kernel-Quellen

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

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

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

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

gadget Eine Einführung in die Erstellung von USB-Slavetreibern

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

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

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

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

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

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

Online-Quellen

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

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

Ergänzungen

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

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

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

2 Theorie ist notwendig

Natürlich könnte der Entwickler sofort in die Programmierung eines Gerätetreibers einsteigen, und innerhalb kurzer Zeit wären bereits erste Erfolge sichtbar. Doch Treiberprogrammierung ist Kernelprogrammierung, und das heißt auch: Nur wenn Betriebssystemkern und Treiber korrekt interagieren, kann die Stabilität des Systems erhalten bleiben. Andernfalls auftretende Fehler sind subtil und nur äußerst schwer zu finden. Auf jeden Fall sollte sich der Kernelentwickler daher mit der Systemarchitektur, den Kernelkomponenten, den internen Abläufen und dem Unterbrechungsmodell auseinandersetzen.

Diesen grundlegenden, theoretischen Unterbau soll das vorliegende Kapitel legen. Darüber hinaus werden wichtige Begriffe wie Kernelkontext oder Userspace erläutert, und es wird somit für eine einheitliche Begriffswelt gesorgt. Spätestens wenn es in den folgenden Kapiteln um das Schlafenlegen einer Treiberinstanz oder um das Sichern eines kritischen Abschnittes geht, werden Sie die hier vermittelten Kenntnisse benötigen.

2.1 Betriebssystemarchitektur

Definition Betriebssystem

Unter einem Betriebssystem versteht man alle Softwarekomponenten, die

- die Ausführung der Applikationen und
- die Verteilung der Betriebsmittel (z. B. Interrupts, Speicher, Prozessorzeit)

steuern und überwachen.

Diese Komponenten lassen sich unterteilen in den *Kernel* (Betriebssystemkern) und in *sonstige Betriebssystemkomponenten*, auch *Userland* genannt. Der Betriebssystemkern ist ein Programm, das sämtlichen Applikationen Dienste in Form sogenannter Systemcalls zur Verfügung stellt; dies gilt insbesondere auch für Betriebssystemapplikationen. Mit

solchen Diensten lassen sich beispielsweise Daten schreiben und lesen, Daten auf einem Bildschirm oder einem Drucker ausgeben oder Daten von einer Tastatur oder einem Netzwerk-Interface entgegennehmen. Das Userland nutzt die Dienste, um damit Systemkonfigurationen vorzunehmen oder einem Anwender die Möglichkeit zu geben, seine Programme zu starten und ablaufen zu lassen.

2.1.1 Komponenten des Kernels

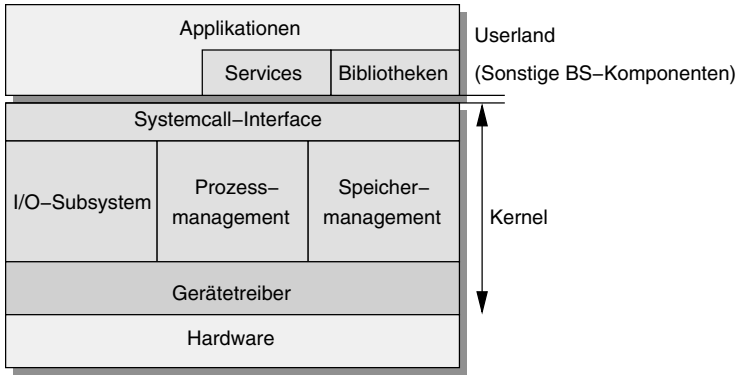


Abb. 2-1 *Betriebssystem-architektur*

In der Abbildung 2-1 ist vereinfacht ein Rechensystem dargestellt. Die grau unterlegten Subsysteme stellen das Betriebssystem dar, wobei sich im unteren Teil der Kernel, im oberen die sonstigen Betriebssystemkomponenten (Services und Bibliotheken) befinden.

Der Betriebssystemkern besteht damit im Wesentlichen aus den folgenden Komponenten, die im Anschluss genauer vorgestellt werden:

- Systemcall-Interface
- Prozessmanagement
- Speichermanagement
- IO-Management
- Gerätetreiber

Systemcall-Interface

Kerneldienste werden per SW-Interrupt angefordert.

Applikationen können die Dienste, die ein Betriebssystem zur Verfügung stellt, über das Systemcall-Interface in Gebrauch nehmen. Technisch ist diese Schnittstelle über eine Art Software-Interrupt realisiert. Möchte eine Applikation einen Dienst (zum Beispiel das Lesen von Daten aus einer Datei) nutzen, löst sie einen Software-Interrupt aus und

übergibt dabei Parameter, die den vom Kernel auszuführenden Dienst hinreichend charakterisieren. Der Kernel selbst führt nach Auslösen des Software-Interrupts die zugehörige Interrupt-Service-Routine (ISR) aus und gibt der aufrufenden Applikation einen Rückgabewert zurück.

Das Auslösen des Software-Interrupts wird im Regelfall durch die Applikationsentwickler nicht selbst programmiert. Vielmehr sind die Aufrufe der Systemcalls in den Standardbibliotheken versteckt, und eine Applikation nutzt eine dem Systemcall entsprechende Funktion in der Bibliothek.

Die Anwendungen fordern die Dienste des IO-Managements über Systemcalls an. Damit wird bei einer Anwendung nicht nur der Code abgearbeitet, der vom Programmierer erstellt wurde, sondern auch der Code, der über die Bibliotheken der eigenen Applikation hinzugebunden wurde, sowie der Kernelcode, der bei der Abarbeitung eines Systemcalls ausgeführt wird.

Welche Systemcalls in Linux vorhanden bzw. implementiert sind, ist in der architekturspezifischen Header-Datei `unistd.h` aufgelistet. Hier ein Ausschnitt für eine 64-Bit-X86-Architektur (Datei `<asm/unistd_64.h>`, genauer `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`):

```
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
...
```

Der folgende Auszug zeigt, dass auf einem Raspberry Pi die Systemcall-Nummern etwas anders nummeriert sind (Datei `/usr/include/arm-linux-gnueabihf/asm/unistd.h`):

```
#ifndef __ASM_ARM_UNISTD_H
#define __ASM_ARM_UNISTD_H

#define __NR_OABI_SYSCALL_BASE 0x900000

#if defined(__thumb__) || defined(__ARM_EABI__)
#define __NR_SYSCALL_BASE 0
#else
#define __NR_SYSCALL_BASE __NR_OABI_SYSCALL_BASE
#endif
```

```
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall (__NR_SYSCALL_BASE+ 0)
#define __NR_exit (__NR_SYSCALL_BASE+ 1)
#define __NR_fork (__NR_SYSCALL_BASE+ 2)
#define __NR_read (__NR_SYSCALL_BASE+ 3)
#define __NR_write (__NR_SYSCALL_BASE+ 4)
#define __NR_open (__NR_SYSCALL_BASE+ 5)
#define __NR_close (__NR_SYSCALL_BASE+ 6)
...
```

Der Systemcall mit der Nummer 1 ist auf einer ARM-Plattform (Raspberry Pi) der Aufruf, um einen Rechenprozess zu beenden (exit); das Erzeugen eines neuen Rechenprozesses erfolgt über den Systemcall fork, welcher die Nummer 2 hat. Über den Systemcall mit der Nummer 3 (read) lassen sich Daten aus Dateien oder von Geräten lesen. Kernel 4.0 bietet als 64-Bit-System 300 unterschiedliche Systemcalls, auf dem 32-Bit-System sind es sogar mehr als 350.

Prozessmanagement

Verteilung von Rechenzeit auf Tasks und Threads

Eine zweite Komponente des Betriebssystemkerns stellt das Prozess-Subsystem dar. Im Wesentlichen verhilft es Einprozessorsystemen (Uniprocessor System, UP) dazu, mehrere Applikationen quasi parallel auf dem einen Mikroprozessor (CPU) abarbeiten zu können. Bei einem Mehrprozessorsystem werden die Applikationen auf die unterschiedlichen Prozessoren verteilt (Symmetric Multiprocessing, SMP). Aus Sicht des Betriebssystems werden Applikationen als Tasks – genauer *Prozesse* oder *Threads* – bezeichnet.

Jede Task besteht aus Code und Daten. Dafür wird im Rechner jeweils ein eigener Speicherblock reserviert. Ein weiterer Speicherblock kommt hinzu, um die während der Abarbeitung der Task abzulegenden Daten zu speichern, der sogenannte Stack. Damit belegt jede Task mindestens drei Speicherblöcke: ein Codesegment, ein Datensegment und ein Stacksegment.

Um Ressourcen (Speicher) zu sparen, können sich mehrere Tasks auch Segmente teilen. Wird beispielsweise dieselbe Office-Applikation zweimal gestartet, wird vom Betriebssystemkern nicht zweimal ein identisches Codesegment angelegt, sondern für beide Tasks nur eins.

Definition von Prozess und Thread

Teilen sich zwei Tasks sowohl das Codesegment als auch das Datensegment, spricht man von *Threads*. Tasks, die jeweils ein eigenes Datensegment besitzen, werden *Prozesse* genannt. Die Threads, die ein

gemeinsames Datensegment verwenden, werden als Thread-Gruppe bezeichnet.

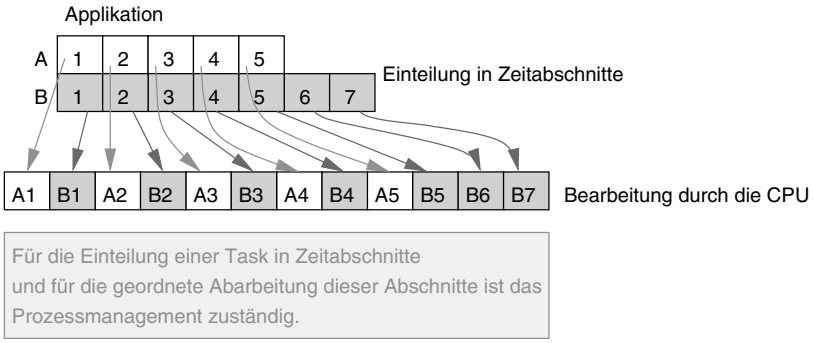


Abb. 2-2
Verarbeitung mehrerer Applikationen auf einer CPU

Das Prinzip des Scheduling

Da auf einer Single-Core-CPU nicht wirklich mehrere Tasks gleichzeitig ablaufen können, sorgt das Task-Management dafür, dass jeweils nur kurze Abschnitte der einzelnen Tasks hintereinander bearbeitet werden. Am Ende einer derartigen Bearbeitungsphase unterbricht das Betriebssystem – ausgelöst durch einen Interrupt – die gerade aktive Task und sorgt dafür, dass ein Folgeabschnitt der nächsten Task bearbeitet wird. Hierdurch entsteht der Eindruck der Parallelität. Welcher der rechenbereiten Prozesse bzw. Threads wirklich rechnen darf, wird durch einen Scheduling-Algorithmus bestimmt, der auch kurz als Scheduler bezeichnet wird. Den Vorgang der Auswahl selbst nennt man Scheduling.

Auf einer Mehrkern- oder Mehrprozessormaschine gibt es reale Parallelität. Hier muss der Scheduler die Tasks auf die vorhandenen CPU-Kerne verteilen (Multicore-Scheduling). Grundsätzlich arbeitet jeder Kern den Code des Single-Core-Schedulers ab. Darüber hinaus arbeitet jeder Kern periodisch einen Kernel-Thread ab (migration), der die Last der einzelnen Cores vergleicht und bei starker Ungleichheit für eine Umverteilung der Jobs (Taskmigration) sorgt. Eine Taskmigration kann es ebenfalls geben, wenn sich eine Task beendet (Systemcall exit), neu gestartet wird (Systemcall clone) oder ihr Codesegment austauscht (Systemcall execve).

Basiszustände der Tasks

Konkurrieren zum Beispiel drei Tasks auf einem Einprozessorsystem um die Ressource CPU, dann rechnet zu einem Zeitpunkt maximal eine der drei. Diese eine Task ist im Zustand *aktiv*. Die anderen beiden Tasks werden dagegen unterbrochen (preempted) und befinden sich im Zustand *lauffähig* (running). Neben diesen beiden Zuständen gibt es noch den Zustand *schlafend*, der häufig auch *wartend* genannt wird. Wie der Name bereits andeutet, schläft eine Applikation auf ein Ereignis, zum Beispiel darauf, dass Daten von der Peripherie geliefert

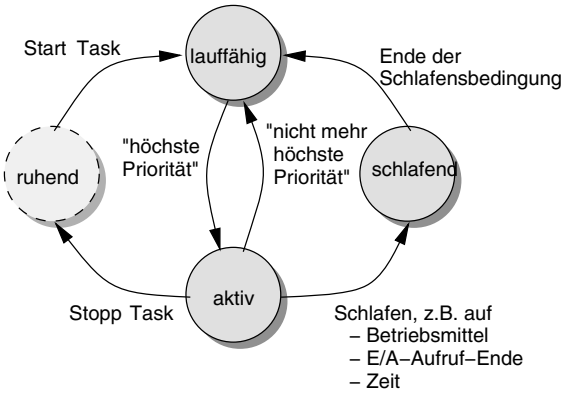


Abb. 2-3
Task-Zustände (Theorie)

werden oder darauf, dass eine bestimmte Zeit verstreicht. Ein vierter (Meta-)Zustand schließlich wird als *ruhend* oder *terminiert* (terminated) bezeichnet. Er steht für die Situation, bevor eine Task gestartet – und damit lauffähig – wird oder nachdem die Task beendet worden ist.

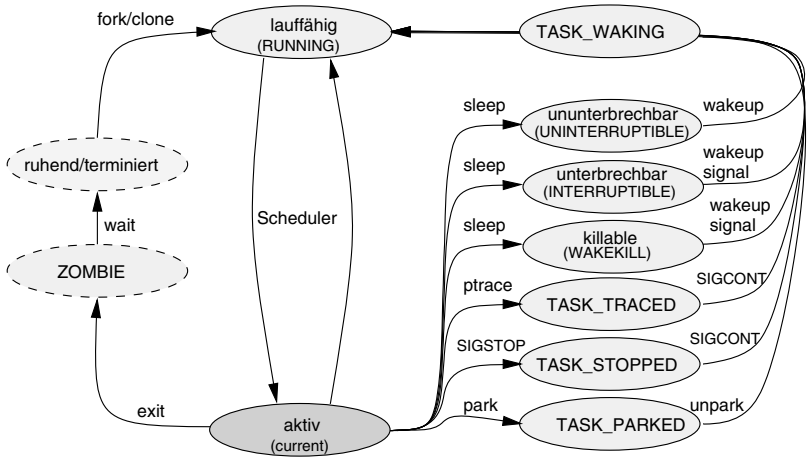


Abb. 2-4
Task-Zustände in Linux

Signale wecken unterbrechbar schlafende Tasks auf.

Im Linux-Kernel selbst ist der Zustand *schlafend* weiter unterteilt in *unterbrechbares*, in *nicht unterbrechbares* und in ein *killable* Schlafen (`TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE` und `TASK_WAKEKILL`). Der Ablauf einer Task kann nicht nur durch den Kernel, sondern eventuell auch durch andere Tasks über sogenannte Signale beeinflusst werden. Diese werden durch Applikationen verschickt. Im Zustand *unterbrechbares Schlafen* wird eine schlafende Task durch ein Signal wieder in den Zustand *lauffähig* (`TASK_RUNNING`) versetzt, im Zustand *nicht unterbrechba-*

res Schlafen dagegen nicht. Der mit Kernel 2.6.25 aufgenommene Zustand `TASK_WAKEKILL` verhält sich ähnlich wie `TASK_INTERRUPTIBLE`. Jobs, die sich in diesem Zustand befinden, werden aufgeweckt, wenn entweder das zugehörige *wakeup* aufgerufen wird (Ende der Wartebedingung) oder aber ein Signal geschickt wird, welches den Job definitiv beendet. Fängt ein Job also einzelne Signale ab, führen diese Signale nicht zu einem Aufwecken. Hintergrund dieses Zustandes ist, dass viele Applikationen unterbrochene Systemcalls nicht korrekt behandeln, also die Rückgabewerte von beispielsweise `read` oder `write` nicht richtig auswerten.

Eine weitere Änderung gegenüber dem vereinfachten Task-Zustandsmodell bringt der Linux-Kernel durch den Zustand *Zombie* (`TASK_ZOMBIE`) mit sich. Beendet sich eine Task, wechselt sie nicht direkt in den Zustand *ruhend/terminiert*, sondern zunächst in den Zustand *Zombie*. In diesem Zustand hat das Betriebssystem noch den Exitcode der Task gespeichert. Erst wenn die Task, die den gerade beendeten Rechenprozess ursprünglich gestartet hat, diesen Exitcode abholt, ist die Task wirklich terminiert.

Abbildung 2-4 verdeutlicht nochmals die Abläufe. Wird mit Hilfe des Systemcalls fork ein neuer Rechenprozess erzeugt, befindet sich dieser im Zustand *TASK_RUNNING*. Wird dieser neue Prozess vom Scheduler ausgewählt, dann wird der Prozess *aktiv*. Dieser Zustand wird im Task-Kontrollblock als solcher nicht abgespeichert. In einem System können genauso viele Tasks aktiv sein, wie Verarbeitungseinheiten (CPUs) vorhanden sind. Welche Task aktiv ist, ist in Linux in dem Variablenfeld »current« abgelegt. Für jeden Prozessor (CPU) ist in diesem Feld ein Element angelegt. Muss eine Task schlafen, wird er per Kernelfunktion »sleep« in den Schlafenszustand (*TASK_INTERRUPTIBLE* oder *TASK_UNINTERRUPTIBLE*) versetzt. Per `wakeup` bzw. im Fall des Zustandes *TASK_INTERRUPTIBLE* auch per Signal verändert sich der Zustand wieder in *TASK_RUNNING*. Ein Rechenprozess beendet sich schließlich über den Systemcall exit. Der Parameter dieses Systemcalls ist der Exitcode, der im sogenannten Task-Kontrollblock (Task Control Block, TCB) eingetragen wird. Solange dieser Exitcode nicht von einem Elternprozess abgeholt wurde, bleibt der TCB im System bestehen. Der Prozesszustand jedoch wird auf *TASK_ZOMBIE* gesetzt. Erst wenn (per Systemcall wait) der Elternprozess den Exitcode abgeholt hat, wird auch der TCB freigegeben.

Das in Abbildung 2-4 vorgestellte Modell weist noch die Zustände *TASK_STOPPED*, *TASK_TRACED* und *TASK_PARKED* auf. Die ersten beiden werden im Kontext von Debugging und Systemcall-Tracing benötigt; Letzterer ermöglicht, einen Per-CPU-Kernel-Thread

in dem Fall, dass eine CPU entfernt wird (CPU-Hotplug), in einen Park-Zustand zu versetzen, aus dem er leicht aufgeweckt werden kann, wenn die CPU wieder aktiviert wird. Diese Zustände haben aber für die Kernelprogrammierung typischerweise keine Relevanz.

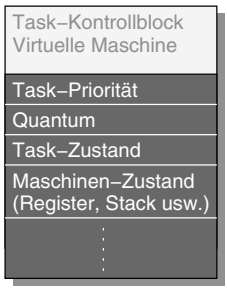


Abb. 2-5
Task-Kontrollblock

Alle wesentlichen Kennwerte einer Task finden sich im TCB.

Werden die Prozesse bzw. Threads durch den Betriebssystemkern unterbrochen, muss dieser eine Reihe von Informationen speichern. Dazu wird ebenfalls der TCB verwendet. Im Task-Kontrollblock werden unter anderem der Task-Zustand, die Prozessidentifikation (PID) und der Inhalt sämtlicher Register zum Zeitpunkt der Unterbrechung, der sogenannte Maschinenzustand, abgelegt. Auch Scheduling-Parameter, beispielsweise Priorität des Rechenprozesses oder verbrauchte Rechenzeit, sind hier gespeichert.

Im Linux-Kernel ist der TCB durch die Task-Struktur `struct task_struct` repräsentiert (siehe Header-Datei `<linux/sched.h>`). So stellt beispielsweise das Feld `pid` die Prozessidentifikation dar, in der Unterstruktur `thread` wird der Maschinenzustand abgelegt, und das Feld `rt_priority` enthält die Task-Priorität.

Speichermanagement

Die dritte Komponente moderner Betriebssysteme ist die Speicherverwaltung. Hard- und Software machen es möglich, dass in Programmen Adressen (sogenannte logische Adressen) verwendet werden, die nicht den physikalischen Adressen entsprechen. Der Entwickler kann Speicherbereiche (Segmente) definieren, die er dann – durch die Hardware unterstützt – bezüglich lesender und schreibender Zugriffe überwachen kann. Darüber hinaus kann sichergestellt werden, dass aus einem Datensegment kein Code gelesen wird bzw. in ein Codesegment keine Daten abgelegt werden.

Hauptspeicher wird in Kernelspace und Userspace eingeteilt.

Systemtechnisch wird dies dazu genutzt, sowohl dem Betriebssystemkern als auch jeder einzelnen Applikation eigene Segmente zuzuordnen. Damit wird verhindert, dass eine Applikation auf den Speicher der anderen Applikation oder gar auf den Speicher des Betriebssystemkerns zugreift. Der Speicherbereich, der vom Kernel genutzt werden kann, wird mit *Kernelspace* bezeichnet. Die Speicherbereiche der Applikationen heißen *Userspace*.

Allerdings kann aber auch der Kernel, und hier insbesondere der Gerätetreiber, nicht direkt auf den Speicher einer Applikation zugreifen. Zwar ist der physikalische Speicher – zumindest unter Linux für die x86-Prozessoren – direkt auf die logischen Adressen umgesetzt (lineares Address-Mapping), doch kennt der Kernel bzw. Treiber damit immer noch nicht die physikalischen Adressen einer bestimmten Task. Schließlich sind die identischen logischen Adressen zweier Tasks auf unterschiedliche physikalische Adressen abgelegt. Erschwerend kommt hinzu, dass das Speicherverwaltungs-Subsystem auch für das sogenannte Paging und Swapping zuständig ist, also die Einbeziehung von Hintergrundspeicher (Festplatte) als Teil des Hauptspeichers. Durch das Swappen kann es geschehen, dass sich der Inhalt eines Segmentes überhaupt nicht im Hauptspeicher, sondern auf der Festplatte befindet. Bevor auf solche Daten zugegriffen werden kann, müssen sie erst wieder in den Hauptspeicher geladen werden.

Die Umrechnung logischer Adressen auf physikalische Adressen wird durch Funktionen innerhalb des Kernels durchgeführt. Das funktioniert aber immer nur für die eine Task, die sich im Zustand *aktiv* befindet (auf die also die globale Variable `current` zeigt).

IO-Management

Ein vierter großer Block des Betriebssystemkerns ist das IO-Management. Dieses ist für den Datenaustausch der Programme mit der Peripherie, den Geräten, zuständig.

Das IO-Management hat im Wesentlichen drei Aufgaben:

1. ein Interface zur systemkonformen Integration von Hardware anzubieten,
2. eine einheitliche Programmierschnittstelle für den Zugriff auf die Peripherie zur Verfügung zu stellen und
3. Ordnungsstrukturen für Daten in Form von Verzeichnissen und Dateien über das sogenannte Filesystem zu realisieren.

Applikationen greifen über Gerätedateien zu.

Idee dieses Programmier-Interface ist es, den Applikationen jegliche Peripherie in Form von Dateien zu präsentieren, die dann *Gerätedateien* genannt werden. Die Gerätedateien sehen für den normalen Anwender wie herkömmliche sonstige Dateien aus. Innerhalb des Dateisystems sind sie aber durch ein Attribut als Gerätedatei gekennzeichnet. In einem Unix-System sind die meisten Gerätedateien im Verzeichnis `/dev/` abgelegt. Dass dies nicht zwingend der Fall ist, liegt daran, dass die Dateien an jedem beliebigen anderen Ort im Verzeichnisbaum erzeugt werden können.

Beispiel 2-1 *Datei und Gerätedatei*

```
-rw-r-----  1 root adm  35532 Oct  1 11:50 /var/log/messages
crw-rw----  1 root lp  6,   0 Feb 23  1999 /dev/lp0
```

In Beispiel 2-1 ist die Ausgabe des Kommandos ls -l für eine normale Datei (ordinary file) und für eine Gerätedatei (device file) angegeben. Gleich anhand des ersten Zeichens, dem »c«, erkennt man bei der Datei `/dev/lp0`, dass es sich um eine Gerätedatei, genauer um ein sogenanntes Character Device File, handelt.

Innerhalb des IO-Managements bzw. IO-Subsystems sind die Zugriffsfunktionen auf Dateien und Geräte realisiert. Dabei ist das Interface vor allem in der jüngeren Vergangenheit ausgebaut und abhängig von den unterschiedlichen Geräten differenziert worden. So existieren inzwischen neben den klassischen Dateizugriffsfunktionen beispielsweise eigene Zugriffsfunktionen für Multimediageräte.

Character Devices

Ähnliches gilt auch für die internen Schnittstellen zur systemkonformen Ankopplung der Peripherie. Klassisch wurden interne Schnittstellen für zeichenorientierte Geräte, sogenannte »Character Devices«, und blockorientierte Geräte, »Block Devices«, zur Verfügung gestellt. Ein Gerät ist zeichenorientiert, wenn die Daten zeichenweise verarbeitet werden. Zeichen kommen mehr oder minder einzeln in einem Strom (Stream) an bzw. gehen in einem Strom weg. Bei einem Character Device ist damit im Regelfall eine Positionierung innerhalb des Datenstroms nicht möglich. Deshalb lassen sich auch die letzten Zeichen nicht vor den ersten lesen bzw. schreiben.

Block Devices

Bei einem blockorientierten Gerät liegt der Fall anders. Hier werden die Daten in Blöcken verarbeitet. Dabei kann durchaus zunächst das Ende eines Datenstroms, dann die Mitte und zuletzt der Anfang gelesen werden. Festplatten, Bänder oder Disketten sind typischerweise Block Devices. Diese werden zur Ablage von Dateien verwendet, wobei auf die Dateien über ein Dateisystem zugegriffen wird.

Die für den Zugriff auf zeichen- oder blockorientierte Geräte definierten Schnittstellen reichen für eine moderne Multimedia-Peripherie nicht mehr aus. Innerhalb des IO-Managements sind daher spezifische Sub-

systeme für die Integration von Netzwerkkarten, Grafikkarten, Soundkarten usw. implementiert. Diese Subsysteme existieren jedoch nicht nur für die unterschiedlichen Gerätetypen, sondern auch für die unterschiedlichen Arten, Geräte anzukoppeln. So gibt es ein SCSI-Subsystem, ein PCI-Subsystem, ein USB-Subsystem oder ein PCMCIA-Subsystem innerhalb des Linux-Kernels.

Gerätetreiber

Die fünfte Komponente eines Betriebssystems sind die Gerätetreiber. Als Softwarekomponente erfüllen sie eine überaus wichtige Funktion: Sie steuern den Zugriff auf alle Geräte! Erst der Treiber macht es einer Applikation möglich, über ein bekanntes Interface die Funktionalität eines Gerätes zu nutzen.

Ganz verschiedene Arten von Hardware werden über Gerätetreiber in ein Betriebssystem integriert: Drucker, Kameras, Tastaturen, Bildschirme, Netzwerkkarten, Scanner, um nur einige Beispiele anzuführen.

Treiber werden durch Subsysteme unterstützt.

Da diese Geräte darüber hinaus über diverse Bussysteme (z. B. PCI, SCSI, USB, I^2C) angeschlossen werden können, haben Betriebssysteme im Allgemeinen und Linux im Besonderen unterschiedliche Treibersubsysteme .

Während traditionell zwischen zeichenorientierten Geräten (Character Devices) und Blockgeräten (Block Devices) unterschieden wird, findet man bei Linux die folgenden Subsysteme (unvollständige Liste):

- Character Devices
- Block Devices
- USB (Universal Serial Bus)
- Netzwerk
- Bluetooth
- FireWire (IEEE1394)
- SCSI (Small Computer System Interface)
- Pincontrol
- GPIO
- IrDA (Infrared Data Association)
- Cardbus und PCMCIA
- Parallelport
- I2C (serielles Kommunikationsprotokoll)
- SPI

Reichhaltiges Schnittstellenangebot

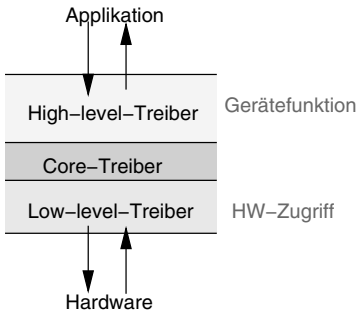
Für diese Vielfalt von Subsystemen ist die Applikationsschnittstelle erweitert worden. Nunmehr lassen sich folgende Interfaces differenzieren:

- das Standard-API (mit Funktionen wie open, close, read, write und ioctl)
- Kommunikations-API
- Card-Services
- Multimedia-Interfaces (z. B. Video4Linux)

Realisiert sind die Interfaces zumeist auf Basis eines Sets standardisierter Datenstrukturen und IO-Controls (um das Systemcall-Interface nicht erweitern zu müssen).

Der Treiber muss die IO-Controls auswerten. Viele Treiber bestehen dabei aus mehreren Schichten (Low-Level, Core und High-Level) mit jeweils spezifischen Aufgaben. Man nennt sie deshalb auch geschichtete Treiber (»stacked driver«).

Abb. 2-6
Treiberstruktur eines geschichteten Treibers



Low- und High-Level-Treiber

Der Low-Level-Treiber ist für die Ansteuerung der internen Hardwareschnittstelle, also beispielsweise eines ganz spezifischen USB-Controllers, zuständig. Da die Anzahl bzw. Auswahl der USB-Komponenten für den direkten Hardwarezugriff gering ist, kommt man hier mit einer geringen Anzahl von Treibern aus. Der Low-Level-Treiber greift direkt auf die Register der Hardware zu. Der High-Level-Treiber dagegen ist für einen Gerätetyp, z. B. eine Webcam, zuständig. Der notwendige Datentransfer zwischen dem Gerät (der Webcam) und dem Treiber wird durch den Low-Level-Treiber durchgeführt; der High-Level-Treiber greift also *nicht* direkt auf die Register der Hardware zu. Bei USB werden beispielsweise zwischen Gerät und Treiber Kommandopakete verschickt. Damit ist der High-Level-Treiber für die Zusammenstellung der richtigen Pakete und die Auswertung der Antworten ver-