

Linux-Treiber entwickeln

Eine systematische Einführung in die
Gerätetreiber- und Kernel-Programmierung

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

2 Theorie ist notwendig

Natürlich könnten Sie 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 sollten Sie sich daher mit der Systemarchitektur, den Kernelkomponenten, den internen Abläufen und dem Unterbrechungsmodell auseinandersetzen.

Diesen grundlegenden, theoretischen Unterbau legt das vorliegende Kapitel. Darüber hinaus werden wichtige Begriffe wie Kernelkontext oder User-Space 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

Unter einem Betriebssystem versteht man alle Softwarekomponenten, die Betriebssystem

Definition Betriebssystem

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

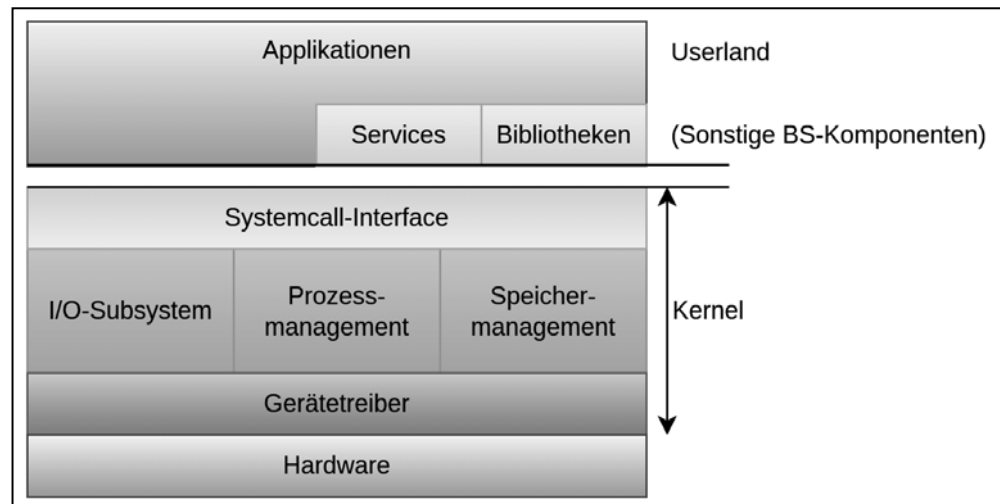
steuern und überwachen.

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 Verfü-

gung 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
Betriebssystemarchitektur



In der Abbildung 2.1 ist vereinfacht ein Rechensystem dargestellt. Die farbige unterlegte 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 klassisch per SW-Interrupt angefordert, wobei moderne Prozessoren hierfür einen eigenen Maschinenbefehl verwenden. Applikationen können die Dienste, die ein Betriebssystem zur Verfügung stellt, über das Systemcall-Interface in Gebrauch nehmen. 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 Applikationsentwicklerinnen und -entwickler 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. Damit wird bei einer Anwendung nicht nur der selbst programmierte Code abgearbeitet, sondern auch 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 architektur-spezifischen 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-gnueabi/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. Ein Kernel der Version 6.x bietet rund 450 Systemcalls.

Prozessmanagement

*Verteilung von Rechenzeit
auf Tasks und Threads*

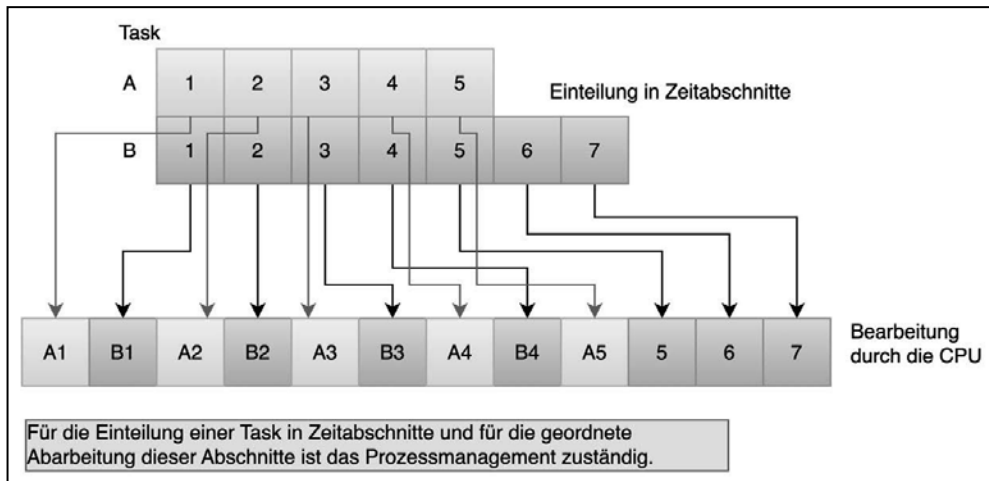
Eine zweite Komponente des Betriebssystemkerns stellt das Prozess-Subsystem dar. Im Wesentlichen verhilft es Einprozessorsystemen (Uniprocessor System, UP) dazu, mehrere Applikationen quasi parallel auf dem einen Mikroprozessor (CPU) abarbeiten zu können. Bei einem Mehrprozessorsystem werden die Applikationen 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. Tatsächlich sind es aber erheblich mehr.

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 eines.

*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

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 (Abbildung 2.2). 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.

Das Prinzip des 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`).

Im Linux-Kernel selbst ist der Zustand schlafend weiter unterteilt. Unter anderem gibt es unterbrechbares, nicht unterbrechbares und ein killable Schlafen (`TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE` und `TASK_WAKEKILL`). Der Ablauf einer Task kann nicht nur durch den Scheduler, 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 unterbrechbares 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 Schlafensbedingung) oder aber ein Signal geschickt wird, das 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 (`EXIT_ZOMBIE`) mit sich. Beendet sich eine Task, wechselt sie nicht direkt in den Zustand 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 (oder alternativ die erste beim Booten vom System erzeugte Task), ist die Task wirklich terminiert.

Abbildung 2.4 verdeutlicht nochmals die Abläufe. Wird mithilfe des Systemcalls `fork` ein neuer Rechenprozess erzeugt, befindet sich dieser (über den Zwischenzustand `TASK_NEW`) im Zustand `TASK_RUNNING`. Wird dieser neue Prozess vom Scheduler ausgewählt, dann wird der Prozess aktiv. Der Zustand aktiv wird im Task-Kontrollblock als solcher nicht vermerkt. 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 Kernel-Funktion `sleep` in den Schlafenzustand (`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,

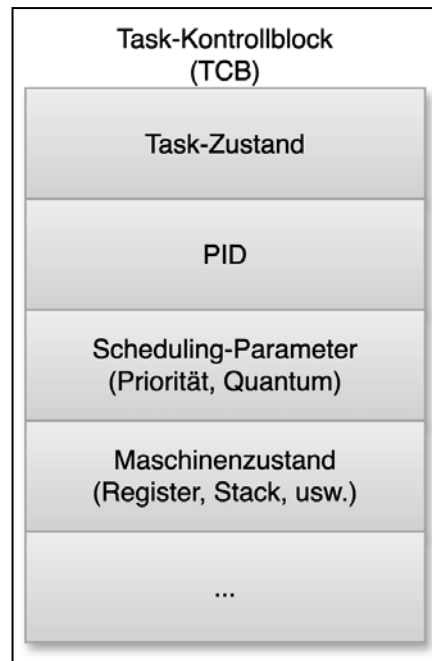
*Signale wecken
unterbrechbar schlafende
Tasks auf.*

Zustandsübergänge

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`, `TASK_NOLOAD` 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. Der Zustand `TASK_NOLOAD` ist exklusiv für Kernel-Threads reserviert und entspricht dem Zustand `TASK_UNINTERRUPTIBLE`. Durch diese Unterscheidung liefern Kernel-Threads keinen Beitrag bei der Berechnung der aktuellen Last. Diese Zustände haben für die Kernelprogrammierung typischerweise keine Relevanz.

Abb. 2.5
Task-Kontrollblock



Alle wesentlichen Kennwerte einer Task finden sich im TCB (Abbildung 2.5). 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 (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 physischen Adressen entsprechen. So können Speicherbereiche (Segmente) definiert werden, die dann – durch die Hardware unterstützt – bezüglich lesender und schreibender Zugriffe überwacht werden. Darüber hinaus wird sichergestellt, dass aus einem Datensegment kein Code gelesen wird bzw. in ein Codesegment keine Daten abgelegt werden.

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 Kernel-Space bezeichnet. Die Speicherbereiche der Applikationen heißen User-Space.

Hauptspeicher wird in Kernel-Space und User-Space eingeteilt.

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

Die Umrechnung logischer Adressen auf physische 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:

- ein Interface zur systemkonformen Integration von Hardware anzubieten,
- eine einheitliche Programmierschnittstelle für den Zugriff auf die Peripherie zur Verfügung zu stellen und
- 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, ist bereits daran erkennbar, dass die Dateien an jedem beliebigen anderen Ort im Verzeichnisbaum erzeugt werden können.

```
-rw-r----- 1 syslog adm 249720 Aug 18 16:55 /var/log/syslog
crw-rw-rw- 1 root  tty   5, 0 Aug 18 16:54 /dev/tty
```

Im Abschnitt »Gerätetreiber« 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/tty`, 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 Schnitt-

stellen 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.

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. SSDs, Festplatten, Bandlaufwerke oder USB-Sticks sind typischerweise Block Devices. Diese werden zur Ablage von Dateien verwendet, wobei auf die Dateien über ein Dateisystem zugegriffen wird.

Block Devices

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 Subsysteme 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 GPIO-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.

Treiber werden durch Subsysteme unterstützt. Da diese Geräte darüber hinaus über diverse Bussysteme (z.B. PCI, SCSI, USB, I2C) 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
- Thunderbolt
- I2C (serielles Kommunikationsprotokoll)
- SPI (serielles Kommunikationsprotokoll)
- Watchdog

*Reichhaltiges
Schnittstellenangebot*

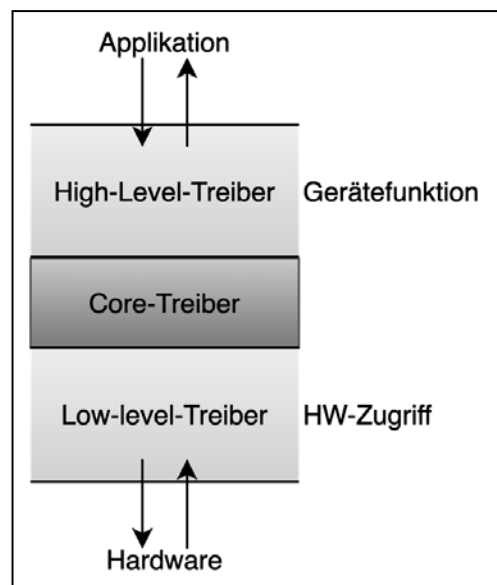
Für diese Vielfalt von Subsystemen ist die Applikationsschnittstelle erweitert worden. Nunmehr lassen sich folgende Interfaces differenzieren:

- das Standard-API (mit Funktionen wie open, close, read, write und ioctl)
- Kommunikations-API
- 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).

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
*Struktur eines
geschichteten Treibers*



*Low- und High-Level-
Treiber*

Der Low-Level-Treiber ist für die Ansteuerung der internen Hardware-schnittstelle, 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 verantwortlich. Der eigentliche Pakettransport wird aber durch den Low-Level-Treiber durchgeführt.

Zwischen Low-Level-Treiber und High-Level-Treiber liegt die Core-Treiberschicht. Diese erweitert die interne Treiberschnittstelle um gerätetypspezifische Funktionen. So stellt im Fall von USB der Core-Treiber Funktionen zum Geräte/Treiber-Management zur Verfügung. In dieser Zwischenschicht ist beispielsweise abgelegt, welche USB-Geräte am USB angeschlossen sind. Der Core-Treiber versucht zudem, den zu einem USB-Gerät passenden Treiber zu finden und zu laden bzw. wenn ein Treiber geladen wird, ein zugehöriges Gerät ausfindig zu machen und dem Treiber zuzuweisen.

Core-Treiber

SCSI-, PCI- und auch Parallelport-Treiber stellen Untergruppen der Character- und Blockgeräte-Treiber dar. USB und Netzwerktreiber bilden eine eigene Gruppe von Treibern.

Soll ein Kernel mit einem neuen Treiber versehen werden, muss theoretisch der gesamte Kernel neu generiert werden. Treiber, die auf diese Art mit dem Betriebssystemkern verbunden sind, nennt man Built-in-Treiber oder auch Kerneltreiber.

Built-in-Treiber sind integraler Teil des Kernels.

Daneben bietet Linux auch die Möglichkeit, zu einem bereits aktiven Kernel einen Treiber hinzuzuladen. In einem solchen Fall ist der Treiber als ladbares Modul realisiert. Diese sogenannten Modultreiber haben mehrere Vorteile. So muss nicht jedes Mal ein neuer Kernel generiert werden, wenn eine Version des Treibers getestet werden soll. Auch entfallen damit das Runterfahren und der Neustart des Systems. Ist der Treiber fertiggestellt, kann er unter Umständen als Modul weitergegeben werden. Verwendet der Nutzer einen gleichkonfigurierten Kernel, kann er das Modul einfach – ebenfalls ohne Neugenerierung von Modul oder Kernel – installieren und verwenden.

Gerade bei Linux kommt es aber vor, dass Funktionen bzw. Schnittstellen unterschiedlicher Kernelversionen voneinander abweichen. Daher kann es im Fall eines Modultreibers dazu kommen, dass ein Treiber für einen älteren Kernel kompiliert wurde und jetzt auf einem Betriebssystemkern eingesetzt werden soll, dessen interne Schnittstellen gegen-

über der früheren Version modifiziert sind. Doch das führt möglicherweise zu Instabilitäten. Daher verlangt Linux, dass Modultreiber passend zur jeweiligen Kernelversion generiert sein müssen.

Wird der Treiber als Open Source herausgegeben, ist das Problem vergleichsweise einfach lösbar, indem (automatisiert) der Treiber auf dem jeweiligen Zielsystem kompiliert wird.

Nicht jeder Treiber kann in Form eines Moduls realisiert werden. Muss der Linux-Kernel beispielsweise gleich nach dem Start auf eine SSD zugreifen, um von dort Systemprogramme und Konfigurationen zu lesen, benötigt er natürlich Treiber für den Zugriff auf eine SSD.

Dennoch geht die Tendenz in die Richtung, dass Treiber grundsätzlich als Module erstellt werden. Einige Linux-Varianten haben nur noch einen Ramdisk-Treiber (Tmp-Filesystem) fest in den Kernel integriert. In einem Tmp-Filesystem (im Hauptspeicher simulierte SSD) schließlich werden die übrigen Treiber als Module abgelegt. Zum Systemstart werden der Linux-Kernel und das Tmp-Filesystem in den Hauptspeicher geladen, im Anschluss wird das System hochgefahren.

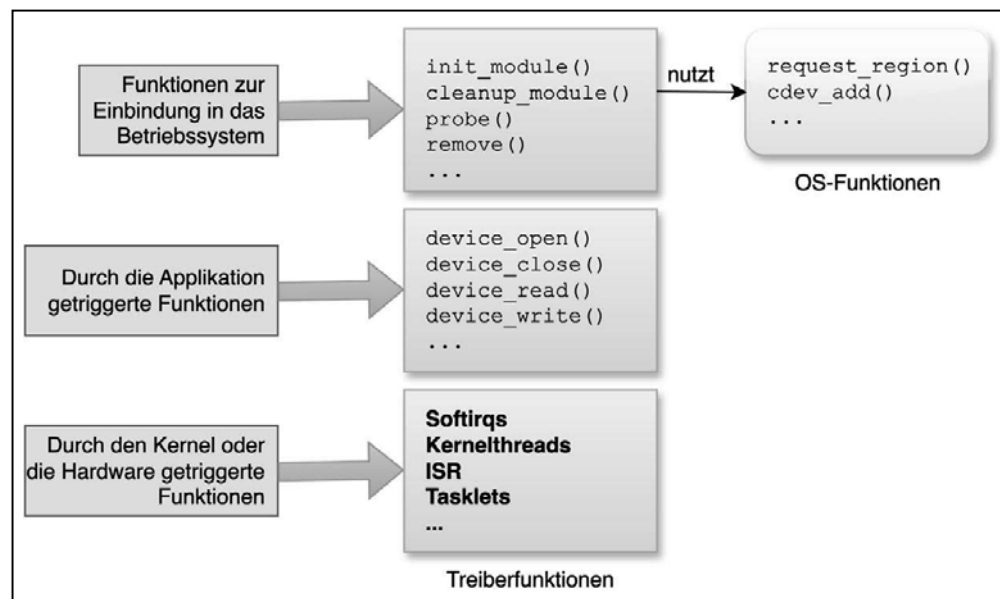
Die nachfolgenden Arten von Funktionen können sich in einem Gerätetreiber befinden (Abbildung 2.7):

- Funktionen, die zur Einbindung des Gerätetreibers in den Kernel notwendig sind,
- Funktionen, die durch die Applikation angestoßen (getriggert) werden, und
- Funktionen, die durch den Betriebssystemkern getriggert werden und unabhängig von Applikationen sind.

Kategorisierung der
Treiberfunktionen

Abb. 2.7

Funktionen eines Treibers



Der Betriebssystemkern besteht aus einer Vielzahl unterschiedlicher Softwarekomponenten, darunter auch den Gerätetreibern. Damit ein

Treiber Teil des Betriebssystemkerns werden kann, müssen bei der Erstellung einige Konventionen eingehalten werden. So müssen – unabhängig davon, ob der Treiber als integraler Bestandteil des Kerns (Built-in-Treiber) oder als Modul geplant ist – eine Initialisierungs- und eine Deinitialisierungsfunktion geschrieben werden. Innerhalb der durch den Kernel aufgerufenen Initialisierungsfunktion klinkt sich der Treiber in den Kernel ein, führt eventuell eine Hardwareerkennung durch und reserviert Ressourcen. Die Deinitialisierungsfunktion wird aufgerufen, wenn der Treiber wieder deaktiviert wird, beispielsweise beim Entladen des Treibermoduls oder beim Herunterfahren des Kerns. Innerhalb dieser Funktion gibt der Treiber allozierte Ressourcen frei und meldet sich beim Kernel wieder ab.

Wird in der Applikation ein Systemcall aufgerufen, der eine Interaktion mit dem Treiber bedingt, ruft der Betriebssystemkern eine der Applikationsfunktion entsprechende Treiberfunktion auf. So triggert beispielsweise innerhalb einer Applikation das Lesen (read) eine Lesefunktion im Treiber. Das Betriebssystem befindet sich aus Sicht des aktivierten Treibers im Prozesskontext (Abbildung 2.8). Wichtig daran: Nur im Prozesskontext ist der Treiber in der Lage, Daten in die Applikation zu kopieren bzw. von der Applikation abzuholen. Die Funktionen zur Integration des Treibers in den Kernel und die Treiberfunktionen, die durch die Applikation getriggert werden, werden in Kapitel 4 vorgestellt.

Bei den in Kapitel 5 vorzustellenden Modulfunktionen liegt der Fall anders. Werden Funktionen des Kernelmoduls unabhängig von irgendwelchen Applikationen aktiviert – einleuchtend ist der Fall, dass eine Hardware selbst, z.B. per Interrupt, eine Funktion triggert –, befindet sich das Betriebssystem im Interrupt-Kontext oder im Kernelkontext. Im Interrupt-Kontext oder im Kernelkontext kann der Treiber nicht auf die Speicherbereiche einer den Treiber nutzenden Applikation zugreifen. Eine im Interrupt-Kontext aufgerufene Funktion des Treibers kann zudem den Treiber nicht schlafen legen. Im Kernelkontext ist das Schlafenlegen allerdings möglich.

Die Aufrufumgebung entscheidet über die Möglichkeiten einer Treiberfunktion.

2.1.2 Sonstige Betriebssystemkomponenten

Ein Betriebssystem besteht nicht nur aus dem Betriebssystemkern, sondern auch aus einer Reihe von Betriebssystemapplikationen und Bibliotheken (Libraries). Die Bibliotheken sind bereits erwähnt worden, beinhalten diese beispielsweise doch Funktionen, die die Systemcalls des Kerns aufrufen.

Auch bei der Treiberentwicklung sind gegebenenfalls Bibliotheken mit einzuplanen, um der Anwendungsprogrammiererin oder dem An-

wendungsprogrammierer vereinfachten Zugang zu komplexen Funktionen zu verschaffen.

Betriebssystemapplikationen werden oft auch Dienste genannt. Diese Dienste gilt es jedoch gegenüber den Diensten des Betriebssystemkerns, die über das Systemcall-Interface durch Applikationen genutzt werden können, abzugrenzen. Die Dienste des Betriebssystems auf Anwenderebene sind meist ständig aktiv, ohne eine spezifische Ausgabe zu machen. In der Unix-Welt bezeichnet man sie auch als Daemons, in der Windows-Welt als Services.

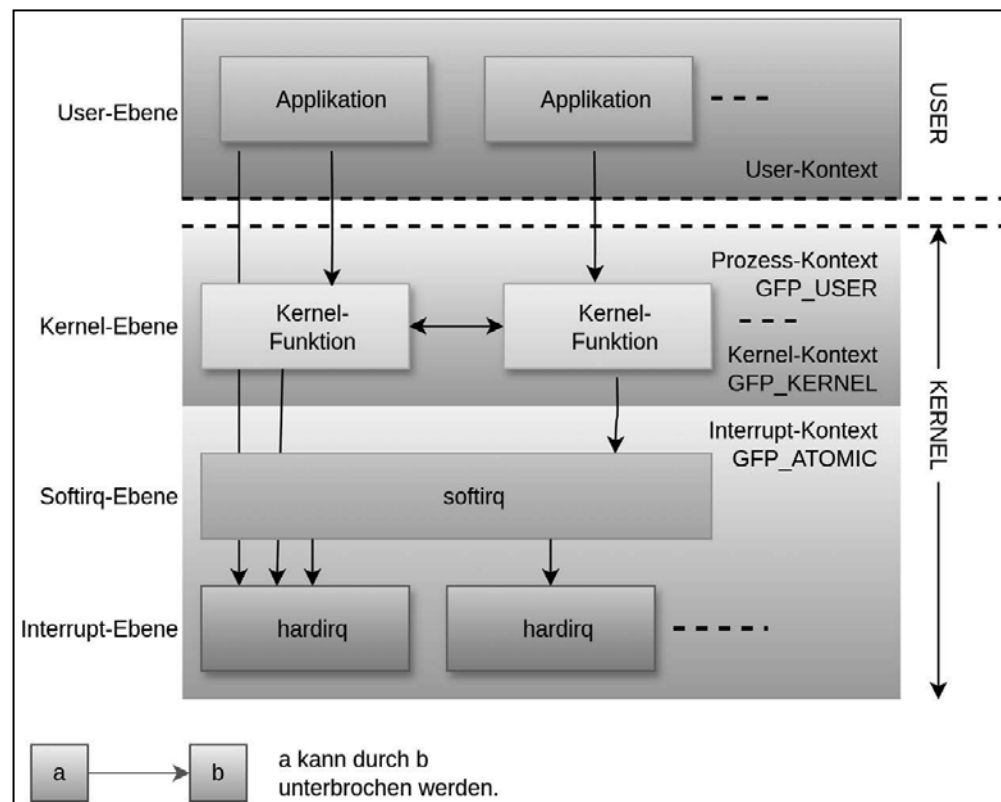
Ein solcher Service ist beispielsweise der syslog-Daemon (syslogd), der für die Protokollierung wichtiger Systemzustände eingesetzt wird.

2.2 Abarbeitungskontext und Unterbrechungsmodell

Das Unterbrechungsmodell ist von zentraler Bedeutung!

Programmcode kann in verschiedenen Umgebungen ablaufen. Die Umgebung – man spricht auch vom Kontext – spezifiziert, welche Dienste und welche Ressourcen einem Codefragment zur Verfügung stehen. Darüber hinaus definiert der Kontext, ob ein Codefragment unterbrechbar ist oder nicht. Diese Kenntnis wiederum ist für Kernelentwicklerinnen und Kernelentwickler unabdingbar, wenn es darum geht zu entscheiden, ob und wie ein kritischer Abschnitt geschützt werden muss.

Abb. 2.8
Unterbrechungsmodell im
Linux-Kernel



Ein kritischer Abschnitt bezeichnet ein Codestück, während dessen Abarbeitung ein Zugriff auf mehrfach genutzte Betriebsmittel, zumeist Daten, stattfindet. Wenn beispielsweise die Abarbeitung einer Funktion per Interrupt unterbrochen wird, arbeitet die zugehörige ISR parallel zur Funktion. Sobald beide dieselbe Variable verwenden, liegt bereits ein kritischer Abschnitt vor. Bleiben kritische Abschnitte ungeschützt, kommt es in der Folge häufig zu Instabilitäten des Systems.

Die verschiedenen Umgebungen, in denen Code ablaufen kann, lassen sich bei Linux gemäß einem Vier-Ebenen-Modell klassifizieren (Abbildung 2.8):

1. einer User- beziehungsweise Applikationsebene,
2. einer Kernelebene,
3. einer Softirq-Ebene und
4. einer ISR-Ebene.

Applikationsebene Code normaler Rechenprozesse, also Code von Tasks und Threads, wird im sogenannten Userkontext abgearbeitet. Ihm stehen die Dienste des Betriebssystems zur Verfügung, wie sie im Systemcall-Interface spezifiziert sind. Code dieser Ebene ist jederzeit unterbrechbar. Die Unterbrechungen finden aufgrund von Interrupts statt, sei es durch einen Hardware-Interrupt oder einen (selbst ausgelösten) Software-Interrupt.

Kernelebene Nach einem Software-Interrupt wird Code auf der Kernelebene abgearbeitet. Diesem Code stehen sämtliche Funktionalitäten des Betriebssystemkerns zur Verfügung – auch die Funktionalität, das Verarbeiten des Codes für einige Zeit anzuhalten (schlafen zu legen) oder Daten zwischen den Speicherbereichen der Applikationen und des Kerns zu übertragen. Applikationsgetriggerte Funktionen eines Gerätetreibers oder Systemcalls werden damit in dieser Prozesskontext genannten Umgebung abgearbeitet.

Sonstige Funktionen des Betriebssystemkerns (Kernel-Threads und damit auch Workqueues) werden im Kernelkontext abgearbeitet. Funktionen in dieser Umgebung können ebenfalls schlafen, da sie aber an keine Applikation gebunden sind, können Sie keine Daten zwischen User- und Kernel-Space austauschen.

Code im Prozess- oder Kernelkontext wird nur durch einen Hardware-Interrupt unterbrochen. Die zugehörigen Hardware-ISRs können dafür sorgen, dass – bevor das Codefragment zu Ende bearbeitet wird – zwischendurch anderer Code im Prozess- oder Kernelkontext abgearbeitet wird. Man spricht von sogenannter Kernel-Preemption.

Darüber hinaus kann jede Funktion der Kernelebene auf einer Mehrprozessormaschine mehrfach parallel abgearbeitet werden.

Softirq-Ebene Codestücke der Softirq-Ebene sind allein durch Hardware-Interrupts unterbrechbar. Da der Code hier im Interrupt-Kontext abläuft, stehen dem Code nicht alle Dienste des Betriebssystemkerns zur Verfügung. Funktionen in diesem Kontext können sich nicht schlafen legen. Sie dürfen auch keine anderen Funktionen wie beispielsweise `kmalloc` aufrufen, die ebenfalls Code schlafen legen könnten. Der Zugriff auf Speicherbereiche der Applikationen ist ebenfalls tabu. Routinen, die auf dieser Ebene ablaufen, sind Softirqs, Tasklets und Timer.

ISR-Ebene Interrupt-Service-Routinen laufen auf der ISR-Ebene ab. Dieser Code ist normalerweise nicht unterbrechbar. Der Abarbeitungskontext ist der Interrupt-Kontext. Auch den ISRs ist es damit untersagt, sich schlafen zu legen oder Funktionen aufzurufen, die einen Rechenprozess schlafen legen wollen.

Zwei weitere Begriffe sind in diesem Zusammenhang noch zu erläutern: Kernel- und User-Space.

Kernel-Space Als Kernel-Space bezeichnet man Speicherbereiche, auf die eine Routine innerhalb des Kernels direkt (über Zeiger) zugreifen kann, ohne besondere Funktionen zu bemühen. Nur der Kernel, also Code, der im Kernelkontext abläuft, hat direkten Zugriff auf den Kernel-Space. Selbst für Applikationen mit Superuser-Rechten ist er tabu.

User-Space Die Speicherbereiche, auf die eine Applikation direkt, zum Beispiel über Zeiger, zugreifen kann, heißen User-Space.

2.3 Quellensuche

Kernel-Quellcode wird benötigt.

Zur Treiberentwicklung wird nicht nur ein aktueller Kernel, sondern es werden Teile der Kernel-Quellen benötigt. Sind die Quellen noch nicht auf dem System installiert, lassen sie sich vom offiziellen Kernelserver <https://www.kernel.org> oder aus dem Git-Repository von Linus Torvalds runterladen. Die Konfiguration der Quellen, das Kompilieren und die Installation des fertigen Kernels sind im Kapitel 10 für eine PC-Plattform und für den Raspberry Pi beschrieben.

Die Kernel-Quellen befinden sich auf dem Linux-System unterhalb des Verzeichnisses `/usr/src/`. Hier wird für jede Kernelversion ein eigener Ordner angelegt. So finden sich die Kernel-Quellen zur Version 6.8 im Verzeichnis `/usr/src/linux-6.8/`. Zusätzlich existiert im Regelfall ein symbolischer Link unter dem Namen `linux`, der auf den gerade verwendeten Kernel zeigt.

Wir gehen im Folgenden davon aus, dass die Quellen des Kernels, für den die Treiberentwicklung stattfinden soll, über das Verzeichnis `/usr/src/linux/` auffindbar sind.

| /usr/src/linux/ | |
|------------------------|-------------------------------------|
| arch | Architekturspezifischer Code |
| block | Blockgeräte-Subsystem |
| crypto | Verschlüsselungsalgorithmen (IPSec) |
| Documentation | Dokumentation |
| drivers | Gerätetreiber |
| firmware | Firmware-Hexcodes |
| fs | Filesysteme |
| include | Header-Dateien |
| init | Initialisierung des Kerns |
| ipc | Interprozesskommunikation |
| kernel | Der Kern des Kerns |
| lib | Oft benötigte Hilfsfunktionen |
| mm | Memory-Management |
| net | Netzwerk Code |
| samples | Beispielcode für Tracepoints |
| scripts | Generierungsskripte |
| security | Sicherheitsmodule (LSM) |
| sound | Audiounterstützung |
| tools | Tools zur Performance-Analyse |
| usr | User-Software des Init-Filesystems |
| virt | Virtualisierung (kvm) |

Abb. 2.9

Verzeichnisstruktur des
Kernel-Quellcodes

Dass der Quellcode von Linux offengelegt ist, hat für alle erhebliche Vorteile: Es stehen genügend Beispiele zur Verfügung, anhand derer sich die Schnittstellen und ihre Verwendung nachvollziehen lassen.

Zur schnelleren Orientierung zeigt Abbildung 2.9 die oberste Verzeichnisstruktur der Kernel-Quellen mit jeweils einer kurzen Erläuterung. Für Treiberentwicklerinnen und Treiberentwickler sind die folgenden Verzeichnisse besonders wichtig:

Quelloffene SW hat erhebliche Vorteile.

drivers/ Sämtliche Gerätetreiber befinden sich unterhalb dieses Verzeichnisses. Zeichenorientierte Gerätetreiber befinden sich im Unterverzeichnis `drivers/char/`, blockorientierte im Unterverzeichnis `drivers/block/`. Die Anbindung an das Gerätemodell (Abschnitt 7.2) befindet sich im Verzeichnis `drivers/base/`.

Documentation/ Unterhalb dieses Verzeichnisses steht die Dokumentation zum Linux-Kernel, die von den Entwicklerinnen und Entwicklern selbst herausgegeben wird. Teile dieser Dokumentation sind leider veraltet. Des Weiteren finden sich hier Hilfsdateien, mit deren Hilfe sich die im Quellcode einkodierte Dokumentation zu wichtigen Systemfunktionen generieren lässt.

include/ Unterhalb dieses Verzeichnisses sind die Header-Dateien aufgelistet. Für den Entwickler sind die Standard-Header-Dateien unter `include/linux/` interessant. Innerhalb des Buches referenziere ich die Dateien unterhalb dieses Verzeichnisses über spitze Klammern. So finden Sie beispielsweise die Datei `<linux/fs.h>` an der Stelle `/usr/src/linux-6.8/include/linux/fs.h`.

arch/ Unterhalb dieses Verzeichnisses sind die architekturenspezifischen Quellcodeteile des Linux-Kernels abgelegt. Die Kernelprogrammiererin und der Kernelprogrammierer schauen insbesondere des Öfteren in den Header-Dateien unter `arch/<platform>/include` beziehungsweise `arch/<platform>/include/asm` nach. `<platform>` ist auf dem PC durch `x86` (also `arch/x86/include/asm`) zu ersetzen.

kernel/ In diesem Verzeichnis sind die zentralen Komponenten des Kernels, beispielsweise das Prozessmanagement, untergebracht.

lib/ In diesem Verzeichnis befindet sich der Quellcode zu Funktionen allgemeiner Natur (klassische Bibliotheksfunktionen).