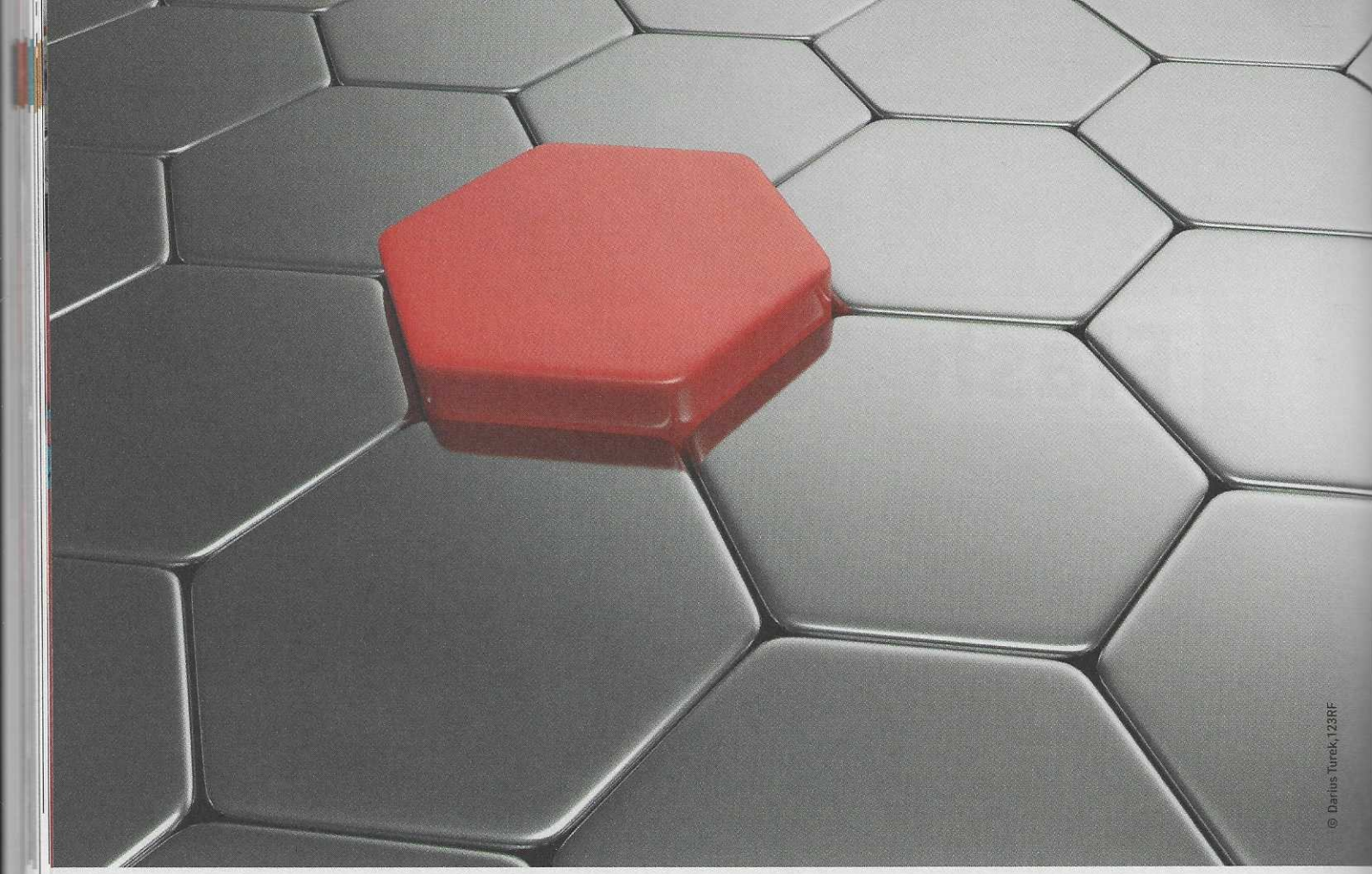


Gerätetreiber als Kernel-Modul implementieren

Kern-Technik

Der professionelle Zugriff auf Hardware im Kernel findet über Gerätetreiber statt. Der Einstieg in die Treiberprogrammierung ist noch kinderleicht, aber danach wird es knifflig.

Eva-Katharina Kunst,

Jürgen Quade

Es gibt ganz unterschiedliche Motivationen, eigenen Code in den Linux-Kernel zu injizieren. Dazu gehören beispielsweise die Notwendigkeit, außergewöhnlich strikte Zeitanforderungen einzuhalten, oder die Neugier, in die Tiefen der System-Software vorzudringen und die Funktionsweise und Abläufe des Kernels kennenzulernen. An selbst erstelltem Kernel-Code kommt auch derjenige nicht vorbei, der professionell auf eigene Hardware zugreifen möchte.

Was auch immer der konktete Antrieb sein mag: Gute C-Kenntnisse vorausgesetzt, macht Linux den Einstieg schon seit Anbeginn erfreulich einfach. Außer dem standardmäßig ohnehin installierten Compiler benötigt man dazu keinerlei besondere Werkzeuge.

Sofern der Code als ladbares Kernel-Modul beziehungsweise als Kernel-Objekt vorliegt, muss der Betriebssystemkern nicht einmal neu kompiliert werden. In diesem Fall genügen die Kernel-Header-Dateien, die Kernel-Konfiguration und eine minimale Build-Umgebung, die zumeist zusammen mit den Header-Files unter `/usr/src/linux/` liegt.

Mit weniger als 10 Zeilen Code am Start

Mit kaum zehn Zeilen Code, der im Wesentlichen eine Funktion mit Namen `init_module()` implementiert, ist der Entwickler am Start (Listing 1). Gibt die Funktion `init_module()` einen Wert ungleich null zurück, signalisiert das dem Kernel, den gerade erst geladenen Modul-Code direkt wieder zu entfernen.

Mithilfe eines passenden Makefiles (Listing 2) lässt sich aus dem Quellcode ein Kernel-Modul generieren. Der Code muss dabei durch das sogenannte Kernel-Build-System generiert werden. Das stellt sicher, dass Kernel und Kernel-

Modul exakt zusammenpassen. In der Make-Variablen `obj-m` gibt der Entwickler die Namen der Module an, die es zu generieren gilt. Anschließend genügt ein einfaches `make`, um die Generierung anzuwerfen.

Nach dem erfolgreichen Abschluss des Vorgangs kann man das Modul mit dem Kommando `sudo insmod mod1.ko` laden. Wie Abbildung 1 zeigt, quittiert das System das Laden mit der Fehlermeldung *Operation not permitted*. Das spielt jedoch keine Rolle: Linux hat den Code des Kernel-Moduls geladen und die Funktion `init_module()` ausgeführt.

```
quade@ezs-mobil:~/linux-magazin$ make
make -C /lib/modules/5.4.0-42-generic/build/ M=/home/quade/linux-magazin modules
make[1]: Verzeichnis „/usr/src/linux-headers-5.4.0-42-generic“ wird betreten
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/quade/linux-magazin/mod1.o
see include/linux/module.h for more information
  CC [M]  /home/quade/linux-magazin/mod1.mod.o
  LD [M]  /home/quade/linux-magazin/mod1.ko
make[1]: Verzeichnis „/usr/src/linux-headers-5.4.0-42-generic“ wird verlassen
quade@ezs-mobil:~/linux-magazin$ sudo insmod mod1.ko
insmod: ERROR: could not insert module mod1.ko: Operation not permitted
quade@ezs-mobil:~/linux-magazin$ tail -n1 /var/log/kern.log
Aug 21 15:10:39 ezs-mobil kernel: [404060.293140] Greetings from Linux-Magazin
quade@ezs-mobil:~/linux-magazin$
```

1 Das Kernel-Modul editieren, generieren und laden.

Debugging über Bildschirmmeldungen

In Listing 1 hat `init_module()` lediglich die Funktion `printk` (print-kernel) aufgerufen, die analog zum bekannten `printf()` eine Ausgabe erzeugt. Die erscheint allerdings nicht direkt auf dem Bildschirm, sondern liegt als Text im Hauptspeicher und landet von dort schließlich im Syslog.

Nicht umsonst ist `tail -f /var/log/kern.log` eines der Lieblingskommandos für Kernel-Code-Entwickler, das klassischerweise in einem separaten Fenster ständig läuft. Tail zeigt die letzten Zeilen einer Datei an, wobei die Option `-f` dafür sorgt, dass es das fortwährend macht. Damit erscheint jede Ausgabe in die Datei direkt auf dem Bildschirm – ein hilfreiches Debug-Instrument. Folglich lässt sich über dieses Werkzeug die Ausführung von `init_module()` verifizieren. Dazu genügt es schon, sich über die Option `-n1` die letzte Zeile der Kernel-Nachrichten ausgeben zu lassen.

Unser Modul selbst hat das Entladen eingefordert und die Falschmeldung des Kernels provoziert. Ploppt allerdings eine andere Fehlermeldung auf, verhindert womöglich ein aktives *Secure Boot* das Laden von Modulen. Um sich das Leben einfach zu machen, schaltet der Entwickler Secure Boot zum Beispiel über das BIOS aus. Es gibt aber auch eine dediziertere Lösung für das Problem.

Das vorgestellte Modul hat nur ein sehr kurzes Debüt gegeben. Klassischerweise soll der Modul-Code jedoch länger im Kernel verbleiben, beispielsweise als Gerätetreiber. Kein Problem: Dazu gibt `init_module()` schlichtweg eine Null zurück. Dann sollte der Programmierer allerdings auch noch eine Funktion `cleanup_modul()` implementieren. Diese wird aufgerufen, wenn man das Modul per `rmmod Modul` wieder aus dem Kernel entfernt. Während sich das Modul über die Funktion `init_module()` im Kernel verankert, löst `cleanup_module()` also diese Verankerung wieder auf.

Kernel-Profis wählen freie Lizenzen

Nach diesem technischen Preview wird es professioneller. Soll der Code nicht als Modul laufen, sondern als fester Bestandteil zum Kernel gehören, sodass er gleich mit den ersten Zuckungen beim Booten bereitsteht, muss man die bisherige Funktion `init_module()` umbenennen. Um diese Zwitter-Eigenschaft des Namens für den Entwickler transparent zu gestalten, empfiehlt Linus Torvalds, einen eindeutigen Namen für die Funktion zu wählen und sie per Makro `module_init()` (Listing 3) abhängig von der Generierungsform automatisiert zu ersetzen. Für denselben Zweck gibt es das Makro `module_exit()`, das beim Generieren zu einem Kernel-Modul den eindeutigen Namen in `cleanup_module()` tauscht.

Gewieften C-Programmierern fallen des Weiteren die beiden Schlüsselwörter `__init` und `__exit` in der Funktionsdefinition auf, die kein Teil der C-Programmiersprache sind. Diese Schlüsselwörter wertet das Kernel-Build-System aus, sie markieren die Funktion als Initialisierungs- beziehungsweise als Deinitialisierungsroutine. Eine Eigenschaft einer Initialisierungsroutine besteht darin, dass sie normalerweise ein einziges Mal aufgerufen wird und dann nie wieder. Nach diesem einen Aufruf liegt der Code der Funktion unnötig im Hauptspeicher herum. Nicht so bei Linux: Der Kernel entfernt nach dem Durchlauf den Code und nutzt den freiwerdenden Hauptspeicher sinnvoller – ganz schön pfiffig.

Darüber hinaus sticht das Makro `MODULE_LICENSE()` ins Auge, das in Listing 1 noch fehlte und beim Kompilieren eine unschöne Warnung provoziert hat. Dieses Makro legitimiert das Kernel-Modul, auf die volle Funktionalität des Linux-Kernels zuzugreifen – oder eben auch

Listing 1: `mod1.c`

```
#include <linux/module.h>
int init_module(void)
{
  printk("Greetings from
Linux-Magazin\n");
  return -1;
}
```

Listing 2: Makefile

```
obj-m := mod1.o
KDIR := /lib/modules/$(shell
uname -r)/build/
PWD := $(shell pwd)
default:
  $(MAKE) -C $(KDIR) M=$(PWD)
modules
```

nicht. Nur wer bereit ist, seinen Code mit anderen zu teilen, darf auf sämtliche Kernel-Funktionen zugreifen. Wer mit einer proprietären Lizenz ankommt, dem stehen dagegen nur einfachste Funktionen des Betriebssystemkerns zur Verfügung.

Kernel-Know-how ist unverzichtbar

So einfach Linux dem Kernel-Neuling den Einstieg auch macht, nimmt mit jedem weiteren Schritt die Komplexität rasant zu. Um fehlerfreien Code zu produzieren, benötigt der Entwickler zunehmend Kernel-Know-how.

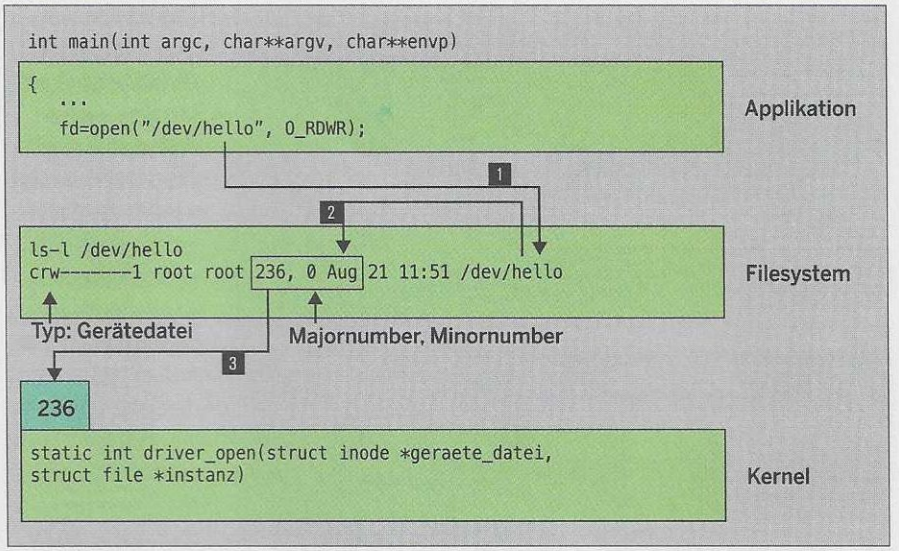
So zeigt Listing 3 ein Kernel-Modul, das das virtuelle Gerät `/dev/hello` zur Verfügung stellt. Der lesende Zugriff auf

Listing 3: Gerätetreiber `hello.c` für das virtuelle Gerät `hello`

```
01 #include <linux/module.h>
02 #include <linux/fs.h>
03 #include <linux/device.h>
04 #include <linux/cdev.h>
05 #include <linux/uaccess.h>
06
07 static char hello_world[]="hello, world\n";
08
09 static dev_t hello_dev_number;
10 static struct cdev *driver_object;
11 static struct class *hello_class;
12 static struct device *hello_dev;
13
14 static int driver_open( struct inode *geraete_datei, struct file *instanz )
15 {
16   printk( "driver_open called\n" );
17   return 0;
18 }
19
20 static int driver_close( struct inode *geraete_datei, struct file *instanz )
21 {
22   printk( "driver_close called\n" );
23   return 0;
24 }
25
26 static ssize_t driver_read( struct file *instanz, char __user *user, size_t count, loff_t *offset )
27 {
28   unsigned long not_copied, to_copy;
29
30   printk("driver_read( %px -- %ld\n", user, count );
31   to_copy = min( count, strlen(hello_world)+1 );
32   not_copied=copy_to_user(user,hello_world,to_copy);
33   *offset += to_copy-not_copied;
34   return to_copy-not_copied;
35 }
36
37 static struct file_operations fops = {
38   .owner= THIS_MODULE,
39   .read= driver_read,
40   .open= driver_open,
41   .release= driver_close,
42 };
43
44 static int __init hello_init( void )
45 {
46   if (alloc_chrdev_region(&hello_dev_number,0,1,"Hello")<0)
47     return -EIO;
48   driver_object = cdev_alloc();
49   if (driver_object==NULL)
50     goto free_device_number;
51   driver_object->owner = THIS_MODULE;
52   driver_object->ops = &fops;
53   if (cdev_add(driver_object,hello_dev_number,1))
54     goto free_cdev;
55   hello_class = class_create( THIS_MODULE, "Hello" );
56   if (IS_ERR( hello_class )) {
57     printk( "hello: no udev support\n");
58     goto free_cdev;
59   }
60   hello_dev = device_create( hello_class, NULL, hello_dev_number, NULL, "%s", "hello" );
61   if (IS_ERR( hello_dev )) {
62     printk( "hello: device_create failed\n");
63     goto free_class;
64   }
65   return 0;
66 free_class:
67   class_destroy( hello_class );
68 free_cdev:
69   kobject_put( &driver_object->kobj );
70 free_device_number:
71   unregister_chrdev_region( hello_dev_number, 1 );
72   return -EIO;
73 }
74
75 static void __exit hello_exit( void )
76 {
77   device_destroy( hello_class, hello_dev_number );
78   class_destroy( hello_class );
79   cdev_del( driver_object );
80   unregister_chrdev_region( hello_dev_number, 1 );
81   return;
82 }
83
84 module_init( hello_init );
85 module_exit( hello_exit );
86
87 MODULE_LICENSE("GPL");
```

die Datei gibt den String „hello, world" zurück. Das Modul verankert sich in der Funktion `hello_init()` im IO-Subsystem des Kernels. Dazu übergibt es dem Kernel mithilfe der Datenstruktur `struct file_operations` eine Reihe von Adressen von Funktionen, die im Modul selbst definiert sind, im Beispiel für die Funktionen `driver_open()`, `driver_close()` und `driver_read()`. Der Kernel ruft diese Funktionen auf, sobald eine Applikation auf die Gerätedatei `/dev/hello` zugreift.

Anwendungen nutzen für den Zugriff auf Dateien und Peripherie die universell einsetzbaren Systemcalls `open()`, `close()`, `read()` und `write()`. Dem Systemcall `open()` übergibt der Anwendungsprogrammierer den Namen der Gerätedatei zusammen mit dem Hinweis, ob der Zugriff lesend, schreibend oder lesend und schreibend erfolgen soll 2.



2 **Gerätedateien bilden das Bindeglied zwischen Anwendung und Treiberfunktionen.**

Der Kernel überprüft zunächst die Existenz der Gerätedatei, danach die Zugriffsrechte. Ist alles in Ordnung, ruft er die vom Treiber zur Verfügung gestellte Funktion `driver_open()` auf. Im einfachsten Fall signalisiert diese mittels des Rückgabewerts `0` dem Kernel, dass auch aus Sicht des Treibers der Zugriff in Ordnung geht, und die Applikation erhält einen gültigen File-Deskriptor (Integer-Wert), den sie für die nachfolgenden Zugriffe (Lesen, Schreiben) nutzen kann.

Ähnlich verhält es sich mit den Systemcalls `read()` und `write()`. Sie erwarten neben dem gültigen File-Deskriptor als Eingabe eine Speicheradresse und die Anzahl von Bytes, die die Speichergröße repräsentieren. `Read()` soll die nächsten Bytes lesen und ab der übergebenen Speicheradresse ablegen. `Write()` holt sich die im Speicher abgelegten Bytes und verarbeitet sie.

Da das eigentliche Lesen und Schreiben von der Hardware abhängt, auf die zugegriffen werden soll, gibt es keinen allgemeinen Code. Der Entwickler implementiert den Zugriff im Treiber in den Funktionen `driver_read()` und `driver_write()`. Da jeder Treiber solche Routinen zur Verfügung stellt, werden die zugehörigen Varianten über den File-Deskriptor identifiziert. Plant die Applikation später keine weiteren Zugriffe auf die Peripherie mehr, ruft sie den Systemcall `close()` auf, der wiederum, über den File-Deskriptor ausgewählt, `driver_close()` aktiviert.

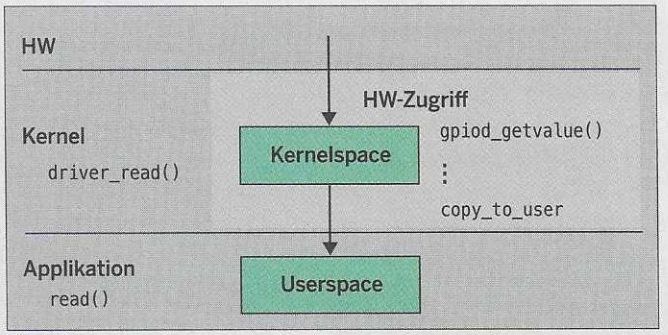
Interessant ist der Aufbau der Funktion `driver_read()`, der der Applikation im Beispiel den String „hello, world" zurückgeben soll. Normalerweise stammen die zu lesenden Informationen von irgendeiner Peripherie, beispielsweise einem Sensor. In diesem Fall finden in der Funktion auch reale Hardware-Zugriffe statt, auf GPIOs zum Beispiel über die Funktion `gpiod_getvalue()` 3.

Die Hardware kopiert die Daten aus diversen Gründen meist nicht direkt in den von der Applikation vorgegebenen Speicher, sondern legt sie zunächst im Kernel ab. Das Kopieren in den Anwendungsspeicher findet in einem zweiten Schritt statt. Beim virtuellen Gerät aus dem Beispiel liegen die Daten – der String „hello world" – bereits im Kernel-Speicher vor, ein Hardware-Zugriff erübrigt sich also.

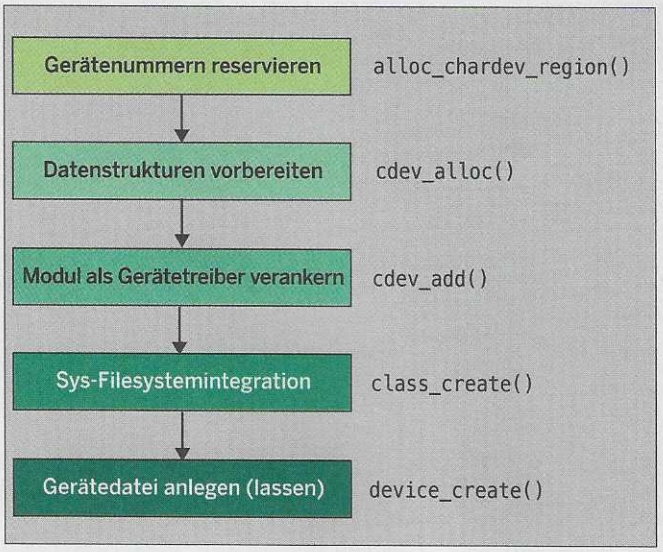
An dieser Stelle ist es notwendig, darauf hinzuweisen, dass Linux zwischen physischem und virtuellem Speicher unterscheidet. Aus Sicht einer Applikation – und ebenso aus Sicht des Kernels – sieht es so aus, als würde aller Speicher im System der jeweiligen Applikation oder dem Kernel exklusiv gehören.

Demnach hat der Kernel seinen eigenen Speicher (Kernelspace), die zugreifende Applikation ebenfalls. Das hat den durchaus erwünschten Nebeneffekt, dass die Anwendung keinen Zugriff auf den Speicher anderer Applikationen und schon gar nicht auf den Kernelspace erhält. Umgekehrt kann aber auch der Kernel nicht einfach in den Applikationsspeicher schreiben.

Mit den Funktionen `copy_to_user()` und `copy_from_user()` stellt Linux dem Programmierer glücklicherweise zwei Funktionen zur Verfügung, die eine Brücke zwischen Anwendungs- und Kernel-Speicher schlagen. Damit kopiert der Entwickler ganz einfach Daten zwischen Kernel- und Userspace hin und her. Ähnlich wie bei `memcpy()` gibt er die Quell- und Zieladresse sowie die zu kopierende Anzahl Bytes an, den Rest erledigt der Kernel. Das Schlüsselwort `__user` im



3 **Hardware-Zugriffe finden nur im Kernel statt.**



4 **Das Verankern im Kernel als Gerätetreiber.**

Prototyp der Funktion `driver_read()` signalisiert übrigens dem Compiler, dass es sich um einen Zeiger auf Applikationsspeicher handelt. Ein Versuch, diesen Zeiger zu dereferenzieren, straft er dann gnadenlos ab.

Schutzmaßnahmen gegen böse Buben

Die Funktionen `copy_to_user()` respektive `copy_from_user()` geben gemäß dem Ansatz, dass bei einem Rückgabewert von null alles in Ordnung ist, die Anzahl der nicht kopierten Bytes zurück. Sollte die Anwendung aber bewusst oder in böser Absicht die Lese- oder Schreibfunktion mit unsinnigen Adress- oder Längenangaben aufrufen, kommt es vor, dass alle oder auch ein Teil der Bytes nicht transferiert werden. Linux passt hier auf. Die Anzahl der zu transferierenden Bytes ist ohnehin ein kritischer Parameter, der zum Schutz des Kernels und auch der Applikation überprüft werden muss. So dürfen auf keinen Fall mehr Bytes kopiert werden, als entweder von der Anwendung angefordert werden oder aber im Kernel an Daten zur Verfügung stehen. Das bewirkt die Minimum-Funktion in Zeile 31 von Listing 3.

In Listing 3 ist in Zeile 33 noch die Aktualisierung des Parameters `offset` zu sehen. Die Funktion `driver_read()` – und korrespondierend `driver_write()` – liest immer die nächsten Daten, was vor allem bei einem Zugriff auf Dateien interessant ist. Mit dem ersten Aufruf von `read()` werden die ersten x Bytes der Datei gelesen, mit dem nächsten Aufruf dann die darauffolgenden und so weiter. Intern führt die Funktion dazu die Variable `offset` mit. In einem sogenannten zeichenorientierten Gerätetreiber spielt der Parameter in den meisten Anwendungsfällen eine untergeordnete Rolle. So verwundert es nicht, dass viele Treiberimplementierungen den Parameter nicht anpassen.

Die Routinen `driver_read()` und `driver_write()` bekommen nicht nur die Adresse des Speicherbereichs übergeben, aus oder zu dem es Daten zu transferieren gilt, sondern auch die Anzahl (Parameter `count`). Des Weiteren gibt es den Parameter `struct file *instance`, über den der Treiber Zugriff auf sämtliche Attribute hat, die die zugreifende Applikation spezifizieren. Unter anderem kann der Treiber darüber auch feststellen, über welche Gerätedatei eine Applikation gerade zugreift. Das nutzt ein Treiber häufig, um in Abhängigkeit von der Gerätedatei unterschiedlich zu reagieren. So könnte er beispielsweise einmal die Temperatur in Celsius zurückgeben, im anderen Fall in Fahrenheit.

Daran lässt sich schon erkennen, dass man einen Treiber über mehrere Gerätedateien als Bindeglied zwischen Applikation und Gerätetreiber ansprechen kann. Wer sich per `ls -l` die Gerätedateien im Ordner `/dev/` auflisten lässt, erkennt, dass für jede Datei zwei Nummern hinterlegt sind, die sogenannte Major und Minor Number 2. Unabhängig davon, dass das im Kernel nur eine sogenannte Gerätenummer ist, referenziert die Major Number den Treiber selbst; die Minor Number stellt so etwas wie einen Parameter dar, der sich wie oben beschrieben vom Kernel-Code auswerten lässt. War es früher notwendig, die Gerätedateien mit ihrer Gerätenummer per Kommando `mknod` im Dateisystem anzulegen, hat mittlerweile das `devtmpfs` diese Aufgabe übernommen.

Anker werfen im Sys-Dateisystem

Die Funktion `hello_init()`, in der sich das Kernel-Modul als Gerätetreiber im System verankert, reserviert in Zeile 46 als Erstes die benötigten Gerätenummern 4. Im Anschluss wird eine Datenstruktur vom Typ `struct cdev` alloziert und mit den Funktionsadressen (`struct file_operations`) sowie den Gerätenummern initialisiert.

Per `cdev_add()` findet die eigentliche Verankerung statt. Ist das erfolgreich, legt die Funktion im Sys-Filesystem, das Informationen über vorhandene Hardware und deren Software-technische Ansteuerung enthält, einen Eintrag (Verzeichnis) für den Treiber als solchen an (`class_create()`, Zeile 55). Außerdem erzeugt sie einen Eintrag (Datei) für jede Gerätedatei mit der zugehörigen Gerätenummer (`device_create()`, Zeile 60). Um das tatsächliche Erzeugen der Gerätedatei muss sich der Programmierer dann, wie oben beschrieben, erfreulicherweise nicht mehr kümmern.

Falls bei diesen Vorgängen Fehler auftreten, erfolgt das Aufräumen in umgekehrter Reihenfolge, dasselbe gilt insbesondere auch für das Entladen des Kernel-Moduls per `rmmod`. Zunächst werden die Einträge im Sys-Filesystem entfernt, was automatisiert zunächst die Gerätedateien löscht, anschließend das den Treiber repräsentierende Verzeichnis. Nach der Freigabe der Gerätenummern wird schließlich die Verankerung des Moduls als Treiber gelöst. Im letzten Schritt entfernt Linux den Modul-Code wieder aus dem Speicher.

Um das Modul `hello.c` aus Listing 3 zu testen, gilt es zunächst, das Makefile anzupassen. In der ersten Zeile von Listing 2 hängt der Programmierer dazu noch `hello.o` an. Daraufhin generiert ein `make` das Kernel-Objekt, das der Entwickler dann per `insmod hello.ko` in den Kernel lädt. Beobachten lässt sich dieser Vorgang per `tail -f /var/log/kern.log`.

Um kein separates C-Programm für den Zugriff auf die beim Laden angelegte Gerätedatei `/dev/hello` schreiben zu müssen, lässt sich das Kommando `cat /dev/hello` nutzen; [Strg]+[C] bricht die Ausgabe ab.

Vorsicht ist die Mutter der Porzellankiste

Noch einmal zusammengefasst: Die Applikationsfunktionen `open`, `read`, `write` und `close` triggern in Abhängigkeit von der verwendeten Gerätedatei die korrespondierenden Treiberfunktionen `driver_open()`, `driver_read()`, `driver_write()` und `driver_close()`.

Während der Kernel-Hacker `driver_open()` für Vorbereitungen für die nachfolgenden Zugriffe nutzen kann, um beispielsweise Hardware zu initialisieren, finden bei `driver_read()` und `driver_write()` die Zugriffe auf die Hardware selbst und der Transfer zwischen den Daten im Kernel und in der Applikation statt. Manche Aspekte wurden hier nicht thematisiert, wie etwa das Schlafenlegen der zugreifenden Applikationen, falls Daten noch nicht verfügbar sind.

Die Verankerung eines Moduls in den Kernel, beispielsweise als Gerätetreiber, findet in der mithilfe des Makros `module_init()` identifizierten Initialisierungsfunktion statt. Die über `module_exit()` definierte Funktion räumt schließlich auf, bevor der Code des Moduls wieder aus dem Kernel entfernt wird.

Bevor jetzt das große Hacken von Kernel-Code ansteht, noch die obligatorische Warnung: Das Entwickeln von Kernel-Code ähnelt einer Operation am offenen Herzen. Unglücklicherweise geht dabei gelegentlich etwas schief, und so gutmütig der Linux-Kernel auch ist, kann man bei seiner Programmierung auch das komplette System zerschießen.

Bekanntlich ist ja Vorsicht die Mutter der Porzellankiste, und so installiert sich der kluge Entwickler in einer virtuellen Maschine ein frisches, schlankes System, an dem er entspannt herumexperimentieren kann. (*jlu*)

Weitere Infos und interessante Links

www.lm-online.de/qr/44068

Der Autor

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade, Professor an der Hochschule Nieder-rhein, bietet auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux an.