|  |
| --- |
|  |

**Kapitel 3. Zeichen-Treiber**

In diesem Kapitel werden wir einen vollständigen Zeichen-Gerätetreiber schreiben. Wir nehmen einen Zeichen-Treiber, weil diese Treiber-Klasse für die meisten einfachen Hardware-Geräte geeignet ist. Außerdem sind sie leichter zu verstehen als beispielsweise Block-Treiber. Das Endziel ist es dabei, einen *modularisierten* Zeichen-Treiber zu schreiben, aber wir werden in diesem Kapitel nicht auf Probleme der Modularisierung eingehen.

Während des ganzen Kapitels werden wir Code-Fragmente vorstellen, die aus einem echten Gerätetreiber stammen: scull, was für *Simple Character Utility for Loading Localities* steht. scull ist ein Zeichen-Treiber, der einen Speicherbereich so anspricht, als würde es sich um ein Gerät handeln. Als Nebeneffekt aus diesem Verhalten kann, solange es um scull geht, der Begriff *Gerät* mit der Bedeutung “der Speicherbereich, der von scull verwendet wird” benutzt werden.

Es ist ein Vorteil von scull, daß der Treiber nicht von irgendwelcher Hardware abhängt, weil ja jeder Computer über Speicher verfügt. scull arbeitet einfach nur auf einem Stück Speicher, das mit *kmalloc* alloziert wird. Jeder kann scull kompilieren und starten; scull ist portabel über alle Rechnerarchitekturen hinweg, auf denen Linux läuft. Auf der anderen Seite macht das Gerät natürlich nichts Nützliches, außer daß es die Schnittstelle zwischen dem Kernel und Zeichen-Treibern demonstriert und den Benutzer einige Tests starten läßt.

**Das Design von scull**

Beim Schreiben eines Treibers besteht der erste Schritt darin, die Funktionen (den *Mechanismus*) zu definieren, den der Treiber den Benutzerprogrammen bieten wird. Weil unser *Gerät* ein Teil des Computerspeichers ist, können wir damit machen, was wir wollen. Es kann sich um ein Gerät mit sequentiellem oder wahlfreiem Zugriff handeln, um ein Gerät oder viele usw.

Damit scull als nützliches Muster eines richtigen Treibers für richtige Geräte dienen kann, werden wir Ihnen zeigen, wie Sie mehrere Geräte-Abstraktionen auf dem Computerspeicher aufsetzen können, alle mit eigenen Charakteristika (einer sogenannten *Personality*).

Der Quellcode von scull implementiert die folgenden Geräte. Jedes Gerät, das vom Modul implementiert wird, wird als ein *Typ* bezeichnet:

scull0-3

Vier Geräte, die aus vier Speicherbereichen bestehen, die alle global und persistent sind. *Global* bedeutet hier, daß auf allen Dateideskriptoren die gleichen Daten benutzt werden, wenn das Gerät mehr als einmal geöffnet wird. *Persistent* bedeutet, daß die Daten nicht verlorengehen, wenn das Gerät geschlossen und wieder geöffnet wird. Es macht Spaß, mit so einem Gerät zu arbeiten, weil man auf das Gerät mit konventionellen Befehlen wie cp, cat und der Shell-I/O-Umleitung zugreifen und es testen kann. Wir werden die Interna des Gerätes in diesem Kapitel untersuchen.

scullpipe0-3

Vier *FIFO*-Geräte, die wie Pipes arbeiten. Ein Prozeß liest die Daten, die ein anderer Prozeß schreibt. Wenn mehrere Prozesse von demselben Gerät lesen wollen, dann konkurrieren sie um die Daten. Die Interna von scullpipe werden zeigen, wie blockierendes und nicht-blockierendes Lesen und Schreiben implementiert werden kann. Das funktioniert auch, ohne auf Interrupts auszuweichen. Obwohl echte Treiber ihre Geräte mittels Hardware-Interrupts synchronisieren, ist es dennoch wichtig, sich mit blockierenden und nicht-blockierenden Operationen zu beschäftigen. Auch kann dieses Thema unabhängig von Interrupts behandelt werden (über die wir in [Kapitel 9](http://www.oreilly.de/german/freebooks/linuxdrive2ger/irq.html) sprechen werden).

scullsingle, scullpriv, sculluid, scullwuid

Diese Geräte ähneln *scull0*, haben aber einige Einschränkungen hinsichtlich der Zeitpunkte, zu denen ein *open* zugelassen ist. Das erste Gerät (*scullsingle*) erlaubt nur jeweils einem Prozeß, auf das Gerät zuzugreifen, während *scullpriv* privat zur jeweiligen virtuellen Konsole (oder X-Terminalsitzung) ist, weil Prozesse auf jeder Konsole und jedem Terminal einen anderen Speicherbereich als Prozesse aus anderen Konsolen bekommen. *sculluid* und *scullwuid* können mehrfach geöffnet werden, aber nur von jeweils einem Benutzer. Die erste Version gibt “Gerät besetzt” zurück, wenn ein anderer Benutzer das Gerät sperrt, während die letzte ein blockierendes *open* implementiert. Diese Varianten von scull stellen mehr “Policy” als “Mechanismus” dar, sind aber trotzdem interessant, weil manche Geräte eine derartige Verwaltung benötigen.

Jedes der scull-Geräte demonstriert verschiedene Features des Treibers und zeigt auch verschiedene Schwierigkeiten auf. In diesem Kapitel werden wir die Interna von *scull0-3* besprechen; die fortgeschritteneren Geräte kommen dann in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html) zur Sprache: *scullpipe* wird in [the Section called *Eine Beispiel-Implementation: scullpipe* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exblocking.html#EXIMPL) beschrieben, die anderen Geräte in [the Section called *Zugriffskontrolle auf Gerätedateien* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exaccess.html).

# 

# Major- und Minor-Nummern

Auf Zeichen-Geräte wird über Namen (oder *Nodes*, dt. Knoten) im Dateisystem zugegriffen. Diese Namen nennt man “spezielle Dateien” oder “Gerätedateien” oder einfach “Knoten im Dateisystem-Baum”; normalerweise liegen sie im /dev-Verzeichnis. Gerätedateien sind besondere Dateien, die durch ein “c” in der ersten Spalte der Ausgabe von *ls -l* kenntlich gemacht werden. Auch Block-Geräte stehen in /dev, werden aber durch ein “b” gekennzeichnet. Auch wenn einige der folgenden Informationen auch auf Block-Geräte zutreffen, werden wir uns hier auf Zeichen-Treiber konzentrieren.

Wenn Sie den Befehl ls -l eingeben, werden Sie bei jeder Gerätedatei zwei durch ein Komma getrennte Zahlen direkt vor dem Modifikationsdatum sehen, wo sonst die Dateilänge steht. Diese Zahlen sind die *Major*- und *Minor*-Nummern des jeweiligen Gerätes. Das folgende Listing zeigt einige wenige Geräte, wie sie auf unserem System aussehen. Die Major-Nummern sind 1, 4, 7 und 10, die Minor-Nummern 1, 3, 5, 64, 65 und 129.

|  |
| --- |
| crw-rw-rw- 1 root root 1, 3 Feb 23 1999 null  crw------- 1 root root 10, 1 Feb 23 1999 psaux  crw------- 1 rubini tty 4, 1 Aug 16 22:22 tty1  crw-rw-rw- 1 root dialout 4, 64 Jun 30 11:19 ttyS0  crw-rw-rw- 1 root dialout 4, 65 Aug 16 00:00 ttyS1  crw------- 1 root sys 7, 1 Feb 23 1999 vcs1  crw------- 1 root sys 7, 129 Feb 23 1999 vcsa1  crw-rw-rw- 1 root root 1, 5 Feb 23 1999 zero |

Die Major-Nummer gibt an, welcher Treiber zu diesem Gerät gehört. Beispielsweise werden sowohl /dev/null als auch /dev/zero vom Treiber 1 verwaltet, während alle virtuellen Konsolen und seriellen Terminals vom Treiber 4 verwaltet werden; gleichermaßen ist der Treiber 7 für die Geräte *vcs1* und *vcsa1* zuständig. Der Kernel verwendet die Major-Nummer, um einem Gerät den passenden Treiber zuzuordnen.

Die Minor-Nummer wird nur vom durch die Major-Nummer angegebenen Gerätetreiber benutzt; andere Teile des Kernels betrachten sie nicht und geben sie lediglich an den Treiber weiter. Es ist nicht ungewöhnlich, daß ein Treiber mehrere Geräte kontrolliert (wie im obigen Beispiel) — die Minor-Nummer ist eine Möglichkeit für den Treiber, zwischen den Geräten zu unterscheiden.

In der Version 2.4 des Kernels kam ein neues (optionales) Feature hinzu, das *Device Filesystem* (*Geräte-Dateisystem*, devfs). Wenn dieses Dateisystem verwendet wird, dann ist die Verwaltung der Gerätedateien einfacher und deutlich anders. Auf der anderen Seite bringt das neue Dateisystem auch mehrere für den Benutzer sichtbare Inkompatibilitäten mit sich und ist von den Distributoren derzeit noch nicht als Default eingesetzt worden. Die obenstehende Beschreibung und die folgenden Anweisungen über das Hinzufügen eines neuen Treibers und einer speziellen Datei gehen davon aus, daß devfs nicht vorhanden ist. Wir füllen diese Lücke später in diesem Kapitel, im Abschnitt "[the Section called *Das Device-Dateisystem*](http://www.oreilly.de/german/freebooks/linuxdrive2ger/chardevfs.html)>".

Wenn devfs nicht verwendet wird, muß einem neuen Treiber eine Major-Nummer zugewiesen werden, um ihn zum System hinzuzufügen. Diese Zuweisung sollte während der Initialisierung des Treibers (bzw. des Moduls) vorgenommen werden, indem die folgende Funktion aufgerufen wird, die in <linux/fs.h> definiert ist:

|  |
| --- |
| int register\_chrdev(unsigned int major, const char \*name,  struct file\_operations \*fops); |

Der Rückgabewert ist ein Fehlercode. Ein negativer Rückgabewert zeigt einen Fehler an; eine Null oder ein positiver Rückgabewert bedeuten eine erfolgreiche Ausführung. Das Argument major ist die angeforderte Major-Nummer, name ist der Name des Gerätes, wie er in /proc/devices erscheinen soll, und fops ist ein Zeiger auf ein Array aus Funktionszeigern, über die die Funktionen des Treibers aufgerufen werden (siehe [the Section called *Datei-Operationen*](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charfops.html) weiter unten in diesem Kapitel).

Die Major-Nummer ist eine kleine Integer-Zahl, die als Index in ein statisches Array von Zeichen-Treibern verwendet wird. > weiter hinten in diesem Kapitel beschreibt, wie man eine Major-Nummer auswählt. Die 2.0-Kernel unterstützten 128 Geräte; in 2.2 und 2.4 wurde dieser Wert auf 256 erweitert (wobei die Werte 0 und 255 für zukünftige Einsatzzwecke vorgesehen sind). Auch Minor-Nummern bestehen aus acht Bits, sie werden aber nicht an *register\_chrdev* übergeben, weil sie ja, wie bereits gesagt, nur vom Treiber verwendet werden. Es gibt einen gewaltigen Druck von seiten der Entwicklergemeinde, die Anzahl der im Kernel möglichen Geräte heraufzusetzen; die Unterstützung von Gerätenummern mit wenigstens 16 Bits ist eines der Ziele in der 2.5-Entwicklung.

Wenn der Treiber erst einmal in der Kernel-Tabelle registriert ist, wird jedesmal, wenn eine Operation auf einer Zeichen-Datei durchgeführt wird, deren Major-Nummer mit der des Treibers übereinstimmt, vom Kernel die korrekte Funktion im Treiber aufgerufen. Dazu wird die Adresse aus der file\_operations-Sprungtabelle genommen. Aus diesem Grund sollte der an *register\_chrdev* übergebene Zeiger auf eine globale Struktur im Treiber zeigen und nicht auf eine lokale Struktur in der Initialisierungsfunktion des Moduls.

Als nächstes stellt sich die Frage, wie man Programmen den Namen bekanntgibt, unter dem sie auf ihren Treiber zugreifen können. Ein solcher Name muß im /dev-Verzeichnis stehen und mit der Major- und Minor-Nummer des Treibers verknüpft sein.

Der Befehl, um einen Knoten im Dateisystem zu erzeugen, lautet mknod und kann nur vom Superuser ausgeführt werden. Der Befehl erwartet neben dem Namen der zu erzeugende Knoten drei Argumente. Beispielsweise erzeugt der Befehl

|  |
| --- |
| mknod /dev/scull0 c 254 0 |

ein Zeichen-Gerät (c), dessen Major-Nummer 254 und dessen Minor-Nummer 0 ist. Minor-Nummern sollten im Bereich von 0 bis 255 liegen, weil sie aus historischen Gründen manchmal in nur einem Byte abgespeichert werden. Es gibt gute Gründe, den Bereich der verfügbaren Minor-Nummern zu erweitern, aber zur Zeit gilt noch die 8-Bit-Grenze.

Bitte beachten Sie, daß die spezielle Gerätedatei nach ihrer Erzeugung durch mknod wie alle anderen Informationen auf der Festplatte verbleibt, sofern sie nicht ausdrücklich wieder entfernt wird. Sie können das in diesem Beispiel erzeugte Gerät mit rm /dev/scull0 wieder entfernen.

## Dynamische Zuweisung von Major-Nummern

Manche Major-Nummern werden den gängigsten Geräten statisch zugewiesen. Eine Liste solcher Geräte finden Sie im Kernel-Quellenbaum in der Datei documentation/devices.txt. Weil viele Nummern schon vergeben sind, kann es schwierig sein, eine eindeutige Nummer für einen neuen Treiber zu wählen, weil es sehr viel mehr spezielle Treiber als Major-Nummern gibt. Sie könnten eine der für “experimentelle oder lokale Verwendung” gekennzeichneten Nummern verwenden,[[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "FTN.AEN2385) aber wenn Sie mit mehreren “lokalen” Treibern arbeiten oder Ihren Treiber auch anderen zur Verfügung stellen, haben Sie wieder das gleiche Problem.

Glücklicherweise (obwohl das weniger auf Glück als vielmehr auf der Genialität eines Entwicklers beruht) kann man eine dynamisch zugewiesene Major-Nummer anfordern. Wenn das Argument major beim Aufruf von *register\_chardev* auf Null gesetzt ist, wählt die Funktion eine freie Nummer aus und gibt diese zurück. Die Major-Nummer ist immer positiv und kann daher nicht mit Fehlercodes verwechselt werden. Bitte beachten Sie die Unterschiede im Verhalten in den beiden Fällen: Die Funktion gibt die allozierte Major-Nummer zurück, wenn der Aufrufer eine dynamische Nummer angefordert hat, aber 0 (nicht die Major-Nummer), wenn eine vordefinierte Major-Nummer erfolgreich registriert werden konnte.

Bei privaten Treibern raten wir Ihnen dringend, die dynamische Zuweisung von Major-Nummern zu verwenden, anstatt eine willkürliche Nummer aus den gerade freien zu verwenden. Wenn Ihr Treiber dagegen allgemein verwendet werden und in den offiziellen Kernel aufgenommen werden soll, dann sollten Sie sich eine für Sie reservierte Major-Nummer zuweisen lassen.

Der Nachteil der dynamischen Zuweisung besteht darin, daß Sie die Geräteknoten nicht vorher anlegen können, weil nicht garantiert ist, daß Ihr Treiber jedesmal dieselbe Nummer erhält. Sie können also das in [Kapitel 11](http://www.oreilly.de/german/freebooks/linuxdrive2ger/kerneld.html) verwendete Laden-bei-Bedarf nicht verwenden. Bei normalen Treibern ist das aber kaum ein Problem, weil Sie die Nummer aus /proc/devices auslesen können, wenn sie einmal zugewiesen worden ist.

Um einen Treiber mit einer dynamischen Major-Nummer zu laden, muß der Aufruf von insmod daher durch ein einfaches Skript ersetzt werden, das nach dem Aufruf von insmod die Datei /proc/devices ausliest, um dann die spezielle(n) Gerätedatei(en) zu erzeugen.

Eine typische /proc/devices-Datei sieht folgendermaßen aus:

|  |
| --- |
| Character devices:  1 mem  2 pty  3 ttyp  4 ttyS  6 lp  7 vcs  10 misc  13 input  14 sound  21 sg  180 usb  Block devices:  2 fd  8 sd  11 sr  65 sd  66 sd |

Das Skript, das ein Modul mit dynamisch zugewiesenen Major-Nummern laden soll, kann also ein Hilfsprogramm wie awk verwenden, um die Informationen aus /proc/devices auszulesen und damit die Dateien in /dev zu erzeugen.

Das folgende Skript scull\_load ist ein Bestandteil der scull-Distribution. Der Benutzer eines Treibers, der in Form eines Moduls ausgeliefert wird, kann solch ein Skript aus der rc.local-Datei des Systems oder bei Bedarf, also wenn das Modul gebraucht wird, aufrufen. Es gibt auch noch eine dritte Möglichkeit: die Verwendung von kerneld.

|  |
| --- |
| #!/bin/sh  module="scull"  device="scull"  group="wheel"  mode="664"  # insmod mit allen übergebenen Parametern aufrufen  # dabei den Pfad angeben, weil neuere modutils defaultmäßig nicht in . suchen  /sbin/insmod -f $module $\* || exit 1  # alte Nodes entfernen  rm -f /dev/${device}[0-3]  major=`cat /proc/devices | awk "\\$2==\"$module\" {print \\$1}"` /proc/devices`  mknod /dev/${device}0 c $major 0  mknod /dev/${device}1 c $major 1  mknod /dev/${device}2 c $major 2  mknod /dev/${device}3 c $major 3  # passende Gruppe und Zugriffsrechte zuweisen und die Gruppe ändern  # Nicht alle Distributionen enthalten staff; auf manchen muß "wheel" verwendet werden  group="staff"  grep '⁁staff:' /etc/group > /dev/null || group="wheel"  chgrp $group /dev/${device}[0-3]  chmod $mode /dev/${device}[0-3] |

Das Skript kann an andere Treiber angepaßt werden, indem man die Variablen anders belegt und die mknod-Zeilen entsprechend modifiziert. Das Skript erzeugt vier Geräte, weil das der Vorgabewert in den scull-Quellen ist.

Die letzten paar Zeilen des Skripts werden Ihnen vielleicht merkwürdig vorkommen: Warum sollte man die Gruppe und die Zugriffsrechte eines Gerätes ändern? Der Grund dafür besteht darin, daß das Skript vom Superuser ausgeführt werden muß und daher neu angelegte Knoten auch diesem gehören. Die Zugriffsrechte sind per Default so, daß nur der Superuser Schreibzugriff, daß aber jeder Lesezugriff hat. Ein Geräteknoten benötigt aber normalerweise andere Zugriffsrechte, weswegen diese verändert werden müssen. Als Default bekommt in unserem Skript eine Gruppe von Benutzern Zugriffsrechte, aber Sie brauchen möglicherweise etwas anderes. Im Abschnitt [the Section called *Zugriffskontrolle auf Gerätedateien* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exaccess.html) in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html) werden wir im Code von *sculluid* zeigen, wie ein Gerätetreiber selbst Zugriffsrechte auf Geräte implementieren kann. Es steht dann ein Skript namens scull\_unload zur Verfügung, um das /dev-Verzeichnis aufzuräumen und das Modul zu entfernen.

Als Alternative zu einem Skriptenpaar zum Laden und Entladen könnten Sie ein init-Skript schreiben, das in das Verzeichnis kommt, das Ihre Distribution für diese Skripten vorsieht.[[2]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "FTN.AEN2445) In den scull-Quellen finden Sie ein ziemlich vollständiges und konfigurierbares Beispiel für ein init-Skript namens scull.init. Es versteht die üblichen Argumente (“start”, “stop” und “restart”) und übernimmt die Rolle sowohl von scull\_load als auch von scull\_unload.

Wenn es Ihnen zu aufwendig ist, immer wieder die /dev-Knoten zu erzeugen und wieder zu löschen, haben wir eine Lösung für Sie: Wenn Sie immer nur einen Treiber laden und entladen, dann können Sie nach dem ersten Erzeugen der Spezialdateien einfach rmmod und insmod verwenden: Dynamische Nummern werden nicht zufällig zugewiesen; Sie können davon ausgehen, wieder die gleiche Nummer zu bekommen, wenn Sie in der Zwischenzeit nichts mit anderen (dynamischen) Modulen machen. In der Entwicklungsphase ist es sicher sinnvoll, längliche Skripte zu vermeiden. Unser Trick funktioniert aber natürlich leider nur solange, wie nicht mehr als ein Treiber beteiligt ist.

Unserer Meinung nach ist es am besten, die dynamische Zuweisung von Major-Nummern als Default vorzusehen, sich aber die Hintertür offenzuhalten, die Major-Nummer zur Lade- oder gar zur Kompilierzeit anzugeben. Die Implementation von scull verwendet eine globale Variable, scull\_major, die die gewählte Nummer enthält. Diese Variable wird mit SCULL\_MAJOR aus scull.h initialisiert. Der Default-Wert von SCULL\_MAJOR ist 0, steht also für *dynamische Zuweisung verwenden*. Der Benutzer kann diesen Default übernehmen oder eine bestimmte Major-Nummer wählen, indem er entweder das Makro vor dem Kompilieren verändert oder einen Wert für scull\_major auf der Kommandozeile angibt. Schließlich kann der Benutzer auch mit dem Skript *scull\_load* auf der Kommandozeile Argumente für insmod übergeben.[[3]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "FTN.AEN2471)

Wir verwenden in scull.c den folgenden Code, um eine Major-Nummer zu bekommen:

|  |
| --- |
| result = register\_chrdev(scull\_major, "scull", &scull\_fops);  if (result < 0) {  printk(KERN\_WARNING "scull: kann Major-Nummer nicht bekommen %d\n",scull\_major);  return result;  }  if (scull\_major == 0) scull\_major = result; /\* dynamisch \*/ |

## Einen Treiber aus dem System entfernen

Wenn ein Modul aus dem System entfernt wird, sollte die Major-Nummer freigegeben werden. Das geschieht mit der folgenden Funktion, die von *cleanup\_module* aufgerufen wird:

|  |
| --- |
| int unregister\_chrdev(unsigned int major, const char \*name); |

Die Argumente sind die freizugebende Major-Nummer und der Name des dazugehörenden Geräts. Der Kernel vergleicht den Namen mit dem für diese Nummer registrierten Namen (sofern vorhanden). Wenn diese nicht übereinstimmen, wird -EINVAL zurückgegeben. Das geschieht auch wenn die Major-Nummer nicht im erlaubten Bereich für Major-Nummern liegt.

Wenn die Ressource nicht in der Aufräumfunktion freigegeben wird, hat das unschöne Nebeneffekte. /proc/devices wird beim nächsten Leseversuch einen Fehler erzeugen, weil einer der Namensstrings noch auf den Speicher des Moduls verweist, der aber längst nicht mehr gültig ist. Diese Art von Fehler wird als *Oops* bezeichnet, weil der Kernel diese Meldung ausgibt, wenn er versucht, auf eine ungültige Adresse zuzugreifen.[[4]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "FTN.AEN2500)

Wenn Sie den Treiber entladen, ohne die Major-Nummer zu deregistrieren, kommen Sie aus dieser Situation schwerlich wieder heraus, denn der Aufruf von *strcmp* in *unregister\_chrdev* muß den Zeiger (name) auf das ursprüngliche Modul verwenden. Wenn Sie jemals vergessen, eine Major-Nummer zu deregistrieren, müssen Sie sowohl das gleiche Modul als auch eines zum Deregistrieren entladen. Das fehlerhafte Modul bekommt mit etwas Glück die gleiche Adresse, und der String name liegt an der gleichen Stelle, wenn Sie den Code nicht verändert haben. Die sichere Alternative ist natürlich, das System zu rebooten.

Über das Entladen des Moduls hinaus müssen Sie oft die Geräteknoten des entladenen Treibers entfernen. Wenn die Geräteknoten während des Ladens erzeugt wurden, dann kann man ein einfaches Skript schreiben, das diese beim Entladen entfernt. Für unser Beispiel-Gerät macht das das Skript scull\_unload; alternativ dazu können Sie auch scull.init stop aufrufen.

Wenn dynamische Gerätedateien nicht aus /dev entfernt werden, kann es unerwartete Fehler geben: Ein überschüssiger /dev/framegrabber-Eintrag auf dem Rechner eines Entwicklers kann einen Monat später zu einem Feueralarm-Gerät gehören, wenn beide Treiber eine dynamische Zuweisung verwenden, um an eine Major-Nummer zu kommen. Wenn beim Öffnen von /dev/framegrabber “No such file or directory” erscheint, dann ist das sicherlich angenehmer als das, was der neue Treiber erzeugen würde.

## dev\_t und kdev\_t

Bisher haben wir über die Major-Nummer gesprochen. Jetzt wird es Zeit, die Minor-Nummern zu untersuchen und zu sehen, wie der Treiber die Minor-Nummer verwendet, um zwischen verschiedenen Geräten zu unterscheiden.

Jedesmal, wenn der Kernel einen Gerätetreiber aufruft, teilt er diesem mit, um welches Gerät es geht. Die Major- und Minor-Nummern werden in einem einzigen Datentyp zusammengefaßt, der vom Treiber verwendet wird, um ein bestimmtes Gerät zu identifizieren. Die kombinierte Gerätenummer (also die aneinandergehängte Major- und Minor-Nummer) steht im Feld i\_rdev der *inode*-Struktur, die wir später kennenlernen werden. Manche Treiberfunktionen bekommen einen Zeiger auf struct inode als erstes Argument übergeben. Wenn Sie diesen Zeiger (wie die meisten Treiber-Autoren) inode nennen, dann kann die Funktion durch Zugriff auf inode->i\_rdev die Gerätenummer herausbekommen.

Traditionell wird unter Unix dev\_t (*device type*) deklariert, um die Gerätenummern abzuspeichern. Früher war das ein 16-Bit-Integer-Wert, der in <sys/types.h> definiert war. Heutzutage braucht man manchmal mehr als 256 Minor-Nummern, aber da es Applikationen gibt, die die Interna von dev\_t “kennen” und nicht mehr laufen würden, wenn diese Struktur geändert werden würde, kann man das nicht so einfach ändern. Die Grundlagen für größere Gerätenummern sind zwar vorhanden, die Nummern selbst sind aber immer noch 16-Bit-Integer-Zahlen.

Im Linux-Kernel wird jedoch ein neuer Typ, kdev\_t, verwendet. Dieser neue Typ ist für alle Kernel-Funktionen eine Blackbox. Benutzerprogramme wissen nicht einmal von der Existenz dieses Typs, und Kernel-Funktionen wissen nicht, was sich dahinter verbirgt. Dadurch, daß kdev\_t versteckt bleibt, kann dieser Typ nach Belieben von einer Kernel-Version zur nächsten geändert werden, ohne daß gleich alle Gerätetreiber angepaßt werden müssen.

Die Informationen über kdev\_t sind in <linux/kdev\_t.h> versteckt. Diese Datei besteht zum größten Teil aus Kommentaren. Wenn Sie sich für die Überlegungen hinter dem Code interessieren, dann lohnt es sich, diese Header-Datei einmal durchzulesen. Sie müssen diese Datei nicht explizit in Ihre Treiber einbinden, weil <linux/fs.h> das schon für Sie macht.

Sie können mit den folgenden Makros und Funktionen auf kdev\_t-Variablen operieren:

MAJOR(kdev\_t dev);

Holt die Major-Nummer aus einer kdev\_t-Struktur.

MINOR(kdev\_t dev);

Holt die Minor-Nummer.

MKDEV(int ma, int mi);

Erzeugt aus Major- und Minor-Nummer einen kdev\_t.

kdev\_t\_to\_nr(kdev\_t dev);

Konvertiert einen kdev\_t-Typ in eine Zahl (einen dev\_t).

to\_kdev\_t(int dev);

Konvertiert eine Zahl in einen kdev\_t. Beachten Sie, daß dev\_t im Kernel-Modus nicht definiert ist, weswegen int verwendet wird.

Solange Ihr Code diese Operationen verwendet, um mit Gerätenummern umzugehen, sollte er auch dann noch funktionieren, wenn sich die internen Datenstrukturen verändern.

### Fußnoten

|  |  |
| --- | --- |
| [[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "AEN2385) | Major-Nummern in den Bereichen 60 bis 63, 120 bis 127 und 240 bis 254 sind für die lokale und experimentelle Verwendung reserviert; kein richtiges Gerät wird eine dieser Nummern zugewiesen bekommen. |
| [[2]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "AEN2445) | Wo die init-Skripten liegen, unterscheidet sich deutlich von Distribution zu Distribution; am häufigsten werden /etc/init.d, /etc/rc.d/init.d und /sbin/init.d verwendet. Wenn Ihr Skript schon beim Booten ausgeführt werden soll, müssen Sie außerdem im passenden Runlevel-Verzeichnis (d.h. ../rc3.d) einen Link darauf anlegen. |
| [[3]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "AEN2471) | Das init-Skript scull.init akzeptiert keine Treiberoptionen auf der Kommandozeile, unterstützt aber eine Konfigurationsdatei, weil es für die automatische Verwendung beim Booten und Herunterfahren gedacht ist. |
| [[4]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charmami.html" \l "AEN2500) | Das Wort “Oops” wird von Linux-Freaks sowohl als Substantiv als auch als Verb gebraucht. |

# Datei-Operationen

In den nächsten Abschnitten schauen wir uns die verschiedenen Operationen an, die ein Treiber auf den von ihm verwalteten Geräten ausführen kann. Ein offenes Gerät wird intern über eine file-Struktur identifiziert, und der Kernel benutzt die Struktur file\_operations, um auf die Funktionen des Treibers zuzugreifen. Diese Struktur, die in <linux/fs.h> definiert wird, ist ein Array aus Funktionszeigern. Jede Datei hat ihren eigenen Satz von Funktionen (indem ein Feld namens f\_op aufgenommen wird, das auf eine file\_operations-Struktur verweist). Die Operationen sind vor allem dafür zuständig, die Systemaufrufe zu implementieren, und heißen daher *open*, *read* usw. Wir können die Datei als “Objekt” und die darauf operierenden Funktionen als “Methoden” ansehen und so die Terminologie der objektorientierten Programmierung verwenden, mit der Aktionen bezeichnet werden, die von einem Objekt auf sich selbst deklariert werden. Hier stoßen wir zum erstenmal auf objekt-orientiertes Design im Linux-Kernel. Davon werden wir später noch mehr sehen.

Per Konvention wird eine file\_operations-Struktur oder ein Zeiger darauf fops (oder so ähnlich) genannt; wir haben so einen Zeiger schon als Argument im Aufruf von *register\_chrdev* kennengelernt. Jedes Feld der Struktur muß auf die Funktion im Treiber verweisen, die die jeweilige Operation implementiert, oder auf NULL bleiben, wenn die Aktion nicht unterstützt wird. Was der Kernel genau macht, wenn dort ein NULL-Zeiger steht, ist von Funktion zu Funktion unterschiedlich, wie die Liste im nächsten Abschnitt zeigen wird.

Die Struktur file\_operations ist langsam gewachsen, während neue Funktionalität zum Kernel hinzugefügt wurde. Das Hinzufügen neuer Operationen kann natürlich zu Portabilitätsproblemen bei Gerätetreibern führen. Instantiierungen der Struktur in jedem Treiber erfolgten früher in Standard-C-Syntax, und neue Operationen wurden normalerweise am Ende der Struktur hinzugefügt. Ein einfaches Neukompilieren der Treiber setzte damit einen NULL-Wert für die neue Operation ein und wählte so das Default-Verhalten aus, was man normalerweise genau erreichen wollte.

Inzwischen sind die Kernel-Entwickler auf ein “Tag”-Initialisierungsformat übergegangen, das die Initialisierung von Strukturfeldern anhand ihres Namens ermöglicht und so die meisten Probleme mit geänderten Datenstrukturen vermeidet. Die Tag-Initialisierung ist aber kein Standard-C, sondern eine (nützliche) Spezialerweiterung des GNU-Compilers. Wir werden uns gleich ein Beispiel für eine Tag-Initialisierung einer Struktur anschauen.

Die folgende Liste führt alle Operationen auf, die eine Applikation auf einem Gerät ausführen kann. Wir haben uns bemüht, die Liste so kurz zu halten, daß sie als Referenz verwendet werden kann. Jede Operation ist nur kurz zusammengefaßt. Außerdem finden Sie das Default-Verhalten des Kernels, wenn ein NULL-Zeiger verwendet wird. Sie können diese Liste beim ersten Lesen überspringen und später darauf zurückkommen.

Nachdem eine weitere wichtige Datenstruktur (die file-Struktur) beschrieben worden ist, erklärt der Rest des Kapitels die Aufgabe der wichtigsten Operationen, gibt Hinweise, weist auf Fallen hin und enthält echte Code-Beispiele. Wir werden die komplexeren Operationen in einem späteren Kapitel besprechen, denn es fehlen uns jetzt noch Kenntnisse über die Speicherverwaltung, über blockierende Operationen und über asynchrone Benachrichtigung.

Die folgende Liste zeigt, welche Operationen in welcher Reihenfolge in der 2.4-Serie des Kernels in struct file\_operations stehen. Obwohl es kleinere Unterschiede zwischen 2.4 und früheren Kerneln gibt, behandeln wir diese erst weiter hinten in diesem Kapitel und beschränken uns hier auf 2.4. Der Rückgabewert der einzelnen Operationen ist im Erfolgsfall 0 oder ein negativer Fehlercode, um einen Fehler anzuzeigen, wenn nichts anderes angegeben ist.

loff\_t (\*llseek) (struct file \*, loff\_t, int);

Die Methode *llseek* wird verwendet, um die aktuelle Lese/Schreib-Position in einer Datei zu verändern; die neue Position wird als (positiver) Rückgabewert zurückgegeben. loff\_t ist ein “langer Offset” und selbst auf 32-Bit-Plattformen mindestens 64 Bits breit. Fehler werden durch einen negativen Rückgabewert gemeldet. Wenn die Funktion für diesen Treiber nicht existiert, schlagen Positionierungen relativ zum Dateiende fehl, andere Positionierungen funktionieren trotzdem; in diesem Fall wird der Positionszeiger in der file-Struktur (beschrieben in [the Section called *Die Struktur file*](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charfile.html)) verändert.

ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);

Wird verwendet, um Daten vom Gerät zu lesen. Steht an dieser Position ein NULL-Zeiger, dann schlägt der *read*-Systemaufruf mit -EINVAL (“Invalid argument”) fehl. Ein nicht-negativer Rückgabewert gibt die Anzahl der erfolgreich eingelesenen Zeichen an (der Rückgabewert ist ein “vorzeichenbehafteter Größentyp”, normalerweise der native Integer-Typ der Zielplattform).

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \* int);

Schickt Daten an das Gerät. Wenn diese Funktion nicht existiert, wird vom Systemaufruf *write* -EINVAL zurückgegeben. Wenn der Rückgabewert nicht negativ ist, enthält er die Anzahl der erfolgreich geschriebenen Bytes.

int (\*readdir) (struct file \*, void \*, filldir\_t);

Dieses Feld sollte bei Geräteknoten NULL sein; es wird nur zum Lesen von Verzeichnissen verwendet und ist nur bei Dateisystemen sinnvoll.

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

Die Methode *poll* ist das Backend zweier Systemaufrufe, *poll* und *select*, die dazu verwendet werden, um abzufragen, ob ein Gerät lesbar oder beschreibbar ist oder sich in einem besonderen Zustand befindet. Beide Systemaufrufe können blockieren, bis ein Gerät lesbar oder beschreibbar wird. Wenn ein Treiber keine *poll*-Methode definiert, dann wird angenommen, daß das Gerät sowohl lesbar als auch beschreibbar ist und sich in keinem besonderen Zustand befindet. Der Rückgabewert ist eine Bitmaske, die den Zustand des Geräts beschreibt.

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

Der *ioctl*-Systemaufruf ermöglicht es Programmen, gerätespezifische Befehle abzusetzen (wie das Formatieren einer Diskettenspur, wobei es sich ja weder um Lesen noch um Schreiben handelt). Einige weitere *ioctl*-Aufrufe werden vom Kernel erkannt, ohne auf die fops-Tabelle zuzugreifen. Wenn das Gerät keinen Einsprungpunkt für *ioctl* enthält, gibt der Systemaufruf bei jeder nicht vordefinierten Anfrage einen Fehler (-ENOTTY, “No such ioctl for device”) zurück. Ein nicht-negativer Rückgabewert wird an das aufrufende Programm zurückgegeben, um eine erfolgreiche Ausführung mitzuteilen.

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

*mmap* wird verwendet, um eine Abbildung von Gerätespeicher auf den Adreßraum des Prozesses anzufordern. Wenn das Gerät diese Methode nicht implementiert, gibt der Systemaufruf *mmap* -ENODEV zurück.

int (\*open) (struct inode \*, struct file \*);

Obwohl dies immer die erste Operation ist, die auf einer Gerätedatei ausgeführt wird, muß ein Treiber nicht unbedingt eine entsprechende Methode deklarieren. Wenn dieser Eintrag NULL ist, dann gelingt das Öffnen des Gerätes immer; das wird Ihrem Treiber allerdings nicht mitgeteilt.

int (\*flush) (struct file \*);

Die *flush*-Operation wird aufgerufen, wenn ein Prozeß seine Kopie eines Dateideskriptors für ein Gerät schließt; die Methode sollte dann alle ausstehenden Operationen auf dem Gerät ausführen (und auf deren Beendigung warten). Verwechseln Sie dies nicht mit der *fsync*-Operation, die von Anwenderprogrammen angefordert wird. *flush* wird derzeit nur vom NFS-Code (dem Network File System) verwendet. Wenn *flush* NULL ist, wird die Methode einfach nicht aufgerufen.

int (\*release) (struct inode \*, struct file \*);

Diese Operation wird aufgerufen, wenn die file-Struktur freigegeben wird. Wie *open* kann auch *release* fehlen.[[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charfops.html" \l "FTN.AEN2766)

int (\*fsync) (struct inode \*, struct dentry \*, int);

Diese Methode ist das Backend des Systemaufrufs *fsync*, den ein Anwender verwendet, um ausstehende Daten herauszuschreiben. Wenn diese Methode vom Treiber nicht implementiert wird, gibt dieser -EINVAL zurück.

int (\*fasync) (int, struct file \*, int);

Diese Operation wird benutzt, um dem Gerät eine Änderung seines FASYNC-Schalters mitzuteilen. Die asynchrone Benachrichtigung ist ein fortgeschrittenes Thema und wird in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html) behandelt. Wenn der Treiber die asynchrone Benachrichtigung nicht unterstützt, kann dieses Feld NULL sein.

ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);, ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);

Diese beiden Methoden wurden spät im 2.3-Entwicklungszyklus hinzugefügt und implementieren sogenannte scatter/gather-Lese- und Schreiboperationen. Applikationen müssen von Zeit zu Zeit einzelne Lese- oder Schreib-Operationen durchführen, bei denen mehrere Speicherbereiche betroffen sind. Diese Systemaufrufe erlauben dies, ohne die Daten zusätzlich kopieren zu müssen.

struct module \*owner;

Dieses Feld ist keine Methode wie alles andere in der Struktur file\_operations. Es handelt sich vielmehr um einen Zeiger auf das Modul, dem diese Struktur “gehört”. Dies wird vom Kernel verwendet, um den Verwendungszähler des Moduls zu pflegen.

Der Gerätetreiber scull implementiert nur die wichtigsten Methoden und verwendet das Tag-Format, um seine file\_operations-Struktur zu deklarieren:

|  |
| --- |
| struct file\_operations scull\_fops = {  llseek: scull\_llseek,  read: scull\_read,  write: scull\_write,  ioctl: scull\_ioctl,  open: scull\_open,  release: scull\_release,  }; |

Diese Deklaration verwendet die Tag-Initialisierungssyntax, die wir oben beschrieben haben. Diese Syntax ist vorzuziehen, weil sie Treiber portabler in Hinblick auf Änderungen in der Strukturdefinition macht, und sie macht den Code auch kompakter und besser lesbar. Die Tag-Initialisierung erlaubt das Umstellen von Struktur-Mitgliedern; in manchen Fällen konnten maßgebliche Performance-Verbesserungen erreicht werden, indem häufig verwendete Struktur-Mitglieder in die gleiche Zeile des Hardware-Caches gestellt wurden.

Außerdem muß das owner-Feld der file\_operations-Struktur gesetzt werden. Oftmals werden Sie dies zusammen mit der restlichen Initialisierung in der folgenden Tag-Syntax vorfinden:

|  |
| --- |
| owner: THIS\_MODULE, |

Dieser Ansatz funktioniert aber nur auf 2.4-Kerneln. Portabler ist die Verwendung des Makros SET\_MODULE\_OWNER, das in <linux/module.h> definiert ist. scull führt diese Initialisierung folgendermaßen durch:

|  |
| --- |
| SET\_MODULE\_OWNER(&scull\_fops); |

Dieses Makro funktioniert auf allen Strukturen, die ein owner-Feld haben; wir werden auf dieses Feld noch in anderem Zusammenhang zurückkommen.

### Fußnoten

|  |  |
| --- | --- |
| [[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charfops.html" \l "AEN2766) | Beachten Sie, daß *release* nicht jedesmal aufgerufen wird, wenn ein Prozeß *close* aufruft. Immer wenn eine file-Struktur gemeinsam genutzt wird (etwa nach einem *fork* oder *dup*), wird *release* nicht aufgerufen, bis nicht alle Kopien geschlossen sind. Wenn Sie ausstehende Daten herausschreiben müssen, wenn eine der Kopien geschlossen wird, sollten Sie die *flush*-Methode verwenden. |

**Die Struktur file**

struct file, definiert in <linux/fs.h>, ist die zweitwichtigste Datenstruktur, die in Gerätetreibern verwendet wird. Beachten Sie, daß file nichts mit der Struktur FILE in Anwendungsprogrammen zu tun hat. FILE wird in der C-Bibliothek definiert und taucht nie im Kernel-Code auf. struct file dagegen ist eine Kernel-Struktur, die nie in Anwendungsprogrammen auftaucht.

Die Struktur file repräsentiert eine *offene Datei*. (Sie ist nichts gerätetreiberspezifisches; jede offene Datei hat so eine struct file im Kernel-Space.) Sie wird vom Kernel beim *open* erzeugt und wird bis zum *close* an jede Funktion übergeben, die auf der Datei operiert. Nachdem die Datei geschlossen worden ist, gibt der Kernel die Datenstruktur frei. Eine *offene Datei* unterscheidet sich von einer *Plattendatei*, die durch struct inode repräsentiert wird.

In den Kernel-Quellen heißen Zeiger auf struct file üblicherweise entweder file oder filp (*File Pointer*). Wir werden den Zeiger durchgehend filp nennen, um Verwechslungen mit der Struktur selbst zu vermeiden: filp ist ein Zeiger (und als solcher eines der Argumente der Geräte-Methoden), file die Struktur selbst.

Die wichtigsten Felder in struct file werden unten aufgeführt. Wie der letzte Abschnitt kann auch diese Liste beim ersten Lesen übersprungen werden. Im nächsten Abschnitt werden wir dann aber auf echten C-Code stoßen, und wir werden dort auf einige dieser Felder eingehen, so daß Sie dann eventuell hier nachschlagen möchten.

mode\_t f\_mode;

Der Dateimodus identifiziert die Datei durch die Bits FMODE\_READ und FMODE\_WRITE als entweder lesbar oder schreibbar (oder beides). Es kann nützlich sein, dieses Feld auf Lese/Schreib-Erlaubnis in Ihren *ioctl*-Funktionen zu überprüfen, aber Sie müssen die Zugriffsrechte nicht beim *read* und *write* überprüfen, weil der Kernel das schon macht, bevor Ihr Treiber aufgerufen wird. Ein Versuch, ohne Schreibrechte zu schreiben, wird zurückgewiesen, bevor der Treiber überhaupt davon erfährt.

loff\_t f\_pos;

Die aktuelle Lese/Schreib-Position. loff\_t ist ein 64-Bit-Wert (in gcc-Terminologie ein long long). Der Treiber kann diesen Wert auslesen, wenn er die aktuelle Position in der Datei wissen muß, sollte diesen Wert aber nie verändern (*read* und *write* sollten die Position durch den Zeiger aktualisieren, den sie als letztes Argument bekommen, anstatt direkt auf filp->f\_pos zuzugreifen).

unsigned int f\_flags;

Dies sind die Datei-Schalter wie O\_RDONLY, O\_NONBLOCK und O\_SYNC. Ein Treiber muß den Schalter für das Nicht-Blockieren überprüfen; die anderen Schalter werden selten verwendet. Insbesondere sollten Lese/Schreib-Rechte mit f\_mode anstelle von f\_flags überprüft werden. Alle Flags werden in der Header-Datei <linux/fcntl.h> definiert.

struct file\_operations \*f\_op;

Die Operationen, die mit der Datei assoziiert sind. Der Kernel weist die Zeiger im Zuge der Implementation von *open* zu und liest sie immer, wenn er eine Operation weiterleiten muß. Der Wert in filp->f\_op wird nie für später abgespeichert; das bedeutet, daß Sie die Datei-Operationen auf Ihrer Datei ändern können, wann immer Sie wollen. Die neuen Methoden gelten dann unmittelbar nach dem Rücksprung zum Aufrufer. Beispielsweise ersetzt der Code für *open* für die Major-Nummer 1 (/dev/null, /dev/zero usw.) die Operationen in filp->f\_op je nachdem, welche Minor-Nummer geöffnet wird. Dieses Verfahren erlaubt die Implementation unterschiedlichen Verhaltens für die gleiche Major-Nummer, ohne daß ein zusätzlicher Systemaufruf eingeführt werden muß. Die Möglichkeit, die Datei-Operationen zu ersetzen, ist das Kernel-Äquivalent zu der Technik, die in der objektorientierten Programmierung *Überschreiben von Methoden* genannt wird.

void \*private\_data;

Der Systemaufruf *open* setzt diesen Zeiger auf NULL, bevor die *open*-Methode des Treibers aufgerufen wird. Der Treiber kann dieses Feld für seine eigenen Zwecke verwenden oder ignorieren. Er kann das Feld auch verwenden, um auf allozierte Daten zu verweisen, muß es dann aber in der *release*-Methode wieder freigeben, bevor die file-Struktur vom Kernel zerstört wird. private\_data ist eine gute Möglichkeit, um Zustandsinformationen zwischen Systemaufrufen abzuspeichern und wird von den meisten unserer Beispielmodule verwendet.

struct dentry \*f\_dentry;

Die Verzeichniseintrag-Struktur (*dentry*, von *directory entry*), die zur Datei gehört. Dentries sind eine in der 2.1-Serie eingeführte Optimierung. Autoren von Gerätetreibern müssen sich — abgesehen vom Zugriff auf die inode-Struktur als filp->f\_dentry->d\_inode — normalerweise nicht mit Dentry-Strukturen befassen.

Die tatsächliche Struktur hat noch einige weitere Felder, die aber für Gerätetreiber nicht nützlich sind. Wir können diese Felder problemlos ignorieren, weil Treiber file-Strukturen nie füllen; sie greifen nur auf andernorts erzeugte Strukturen zu.

# open und release

Nachdem wir uns jetzt die Felder kurz angeschaut haben, benutzen wir sie in echten scull-Funktionen.

## Die Methode open

Die Methode *open* wird von Treibern bereitgestellt, um mit Initialisierungen spätere Operationen vorzubereiten. Außerdem erhöht *open* normalerweise den Verwendungszähler des Gerätes, damit das Modul nicht entladen wird, bevor die Datei wieder geschlossen wird. Der Zähler, der in Abschnitt in [Kapitel 2](http://www.oreilly.de/german/freebooks/linuxdrive2ger/building.html) beschrieben wird, wird dann von der Methode *release* wieder heruntergezählt.

In den meisten Treibern macht *open* folgendes:

* Es inkrementiert den Verwendungszähler.
* Es fragt gerätespezifische Fehler ab (beispielsweise könnte das Gerät nicht bereit sein oder andere Hardware-Fehler könnten auftreten).
* Es initialisiert das Gerät, wenn es das erste Mal geöffnet wird.
* Es identifiziert die Minor-Nummer und aktualisiert gegebenenfalls den f\_op-Zeiger.
* Es alloziert und füllt Datenstrukturen, die in filp->private\_data untergebracht werden sollen.

ark=dash>

In scull hängen die meisten der obengenannten Aufgaben von der Minor-Nummer des geöffneten Gerätes ab. Daher muß als erstes das Gerät, um das es geht, identifiziert werden. Dazu sehen wir uns inode->i\_rdev an.

Wir haben bereits darüber gesprochen, daß der Kernel die Minor-Nummer des Gerätes selbst nicht verwendet, so daß der Treiber diese nach Belieben verwenden kann. In der Praxis werden verschiedene Minor-Nummern verwendet, um auf verschiedene Geräte zuzugreifen oder um ein und dasselbe Gerät auf verschiedene Arten zu öffnen. Beispielsweise verweisen /dev/st0 (Minor-Nummer 0) und /dev/st1 (Minor-Nummer 1) auf verschiedene SCSI-Bandlaufwerke, aber */dev/nst0* (Minor-Nummer 128) ist das gleiche physikalische Gerät wie */dev/st0*, verhält sich aber anders (es spult das Band nicht zurück, wenn das Gerät geschlossen wird). Alle Gerätedateien für Bandlaufwerke haben unterschiedliche Minor-Nummern, damit der Treiber sie auseinanderhalten kann.

Ein Treiber kennt selbst nie den Namen des Gerätes, das gerade geöffnet wird, nur die Gerätenummer. Benutzer können sich das zunutze machen, indem sie zur Bequemlichkeit Aliasnamen für Geräte anlegen. Wenn Sie zwei Gerätedateien mit dem gleichen Paar aus Minor- und Major-Nummer anlegen, dann sind das die gleichen Geräte, die Dateien können nicht unterschieden werden. Den gleichen Effekt erreichen Sie mit einem symbolischen oder harten Link; das ist auch die empfohlene Vorgehensweise.

Der scull-Treiber verwendet Minor-Nummern folgendermaßen: Das höherwertige Nibble (vier Bits) identifiziert den Typ (die Personality) des Gerätes, und das niederwertige Nibble dient zur Unterscheidung zwischen den einzelnen Geräten, wenn der Typ mehr als eine Geräteinstanz unterstützt (scull0-3 und scullpipe0-3). scull0 unterscheidet sich also von scullpipe0 im höherwertigen Nibble, scull0 und scull1 im niederwertigen Nibble.[[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/x2969.html" \l "FTN.AEN3024) Zwei Makros (TYPE und NUM) werden definiert, um die Bits aus der Gerätenummer zu gewinnen:

|  |
| --- |
| #define TYPE(dev) (MINOR(dev) >> 4) /\* oberes Nibble \*/  #define NUM(dev) (MINOR(dev) & 0xf) /\* unteres Nibble \*/ |

Zu jedem Gerätetyp definiert scull eine spezifische Struktur file\_operations, die beim Öffnen in filp->f\_op eingesetzt wird. Der folgende Code zeigt, wie das Aufteilen in Bits und mehrfache fops implementiert werden:

|  |
| --- |
| struct file\_operations \*scull\_fop\_array[]={  &scull\_fops, /\* type 0 \*/  &scull\_priv\_fops, /\* type 1 \*/  &scull\_pipe\_fops, /\* type 2 \*/  &scull\_sngl\_fops, /\* type 3 \*/  &scull\_user\_fops, /\* type 4 \*/  &scull\_wusr\_fops /\* type 5 \*/  };  #define SCULL\_MAX\_TYPE 5  /\* In scull\_open wird fop\_array entsprechend TYPE(dev) verwendet \*/  int type = TYPE(inode->i\_rdev);  if (type > SCULL\_MAX\_TYPE) return -ENODEV;  filp->f\_op = scull\_fop\_array[type]; |

Der Kernel ruft *open* entsprechend der Major-Nummer auf, und scull verwendet dann die Minor-Nummer in den oben definierten Makros. TYPE wird als Index in das Array scull\_fop\_array verwendet, um den richtigen Satz von Methoden für den gerade geöffneten Gerätetyp zu bekommen.

In scull haben wir die korrekte file\_operations-Struktur entsprechend des Typs der Minor-Nummer an filp->f\_op zugewiesen. Dann wird die in den neuen fops definierte *open*-Methode aufgerufen. Normalerweise ruft ein Treiber seine eigenen fops nicht auf, diese werden vom Kernel verwendet, um die richtige Treiber-Methode aufzurufen. Aber wenn die *open*-Methode mit verschiedenen Gerätetypen umgehen können muß, dann kann es doch sinnvoll sein, fops->open aufzurufen, nachdem der fops-Zeiger entsprechend der Minor-Nummer des geöffneten Gerätes modifiziert wurde.

Der eigentliche Code von *scull\_open* steht hier. Er verwendet die oben definierten Makros TYPE und NUM, um die Minor-Nummer aufzuteilen:

|  |
| --- |
| int scull\_open(struct inode \*inode, struct file \*filp)  {  Scull\_Dev \*dev; /\* Geraeteinformation \*/  int num = NUM(inode->i\_rdev);  int type = TYPE(inode->i\_rdev);  /\*  \* Wenn private\_data nicht gueltig ist, dann verwenden wir kein devfs  \* und benutzen deswegen den Typ (aus der Minor-Nr.), um ein f\_op  \* auszuwaehlen  \*/  if (!filp->private\_data && type) {  if (type > SCULL\_MAX\_TYPE) return -ENODEV;  filp->f\_op = scull\_fop\_array[type];  return filp->f\_op->open(inode, filp); /\* an spezifisches open  \* weiterleiten \*/  }  /\* Typ 0, Geraetenummer ueberpruefen (sofern private\_data nicht gueltig ist) \*/  dev = (Scull\_Dev \*)filp->private\_data;  if (!dev) {  if (num >= scull\_nr\_devs) return -ENODEV;  dev = &scull\_devices[num];  filp->private\_data = dev; /\* for other methods \*/  }  MOD\_INC\_USE\_COUNT; /\* Moeglicherweise vor dem Schlafen \*/  /\* jetzt die Laenge des Geraets auf 0 setzen, wenn es nur zum Schreiben  geöffnet wurde \*/  if ( (filp->f\_flags & O\_ACCMODE) == O\_WRONLY) {  if (down\_interruptible(&dev->sem)) {  MOD\_DEC\_USE\_COUNT;  return -ERESTARTSYS;  }  scull\_trim(dev); /\* Fehler ignorieren \*/  up(&dev->sem);  }  return 0; /\* Erfolg \*/  } |

Hier sind wahrscheinlich noch einige Erklärungen angebracht. Die Datenstruktur, in der der Speicherbereich untergebracht wird, heißt Scull\_dev und wird gleich erläutert. Die globalen Variablen scull\_nr\_devs und scull\_devices[] (alle in Kleinbuchstaben) geben die Anzahl der verfügbaren Geräte und das eigentliche Array von Zeigern in Scull\_dev an.

Die Aufrufe von *down\_interruptible* und *up* können Sie im Moment ignorieren, wir kommen darauf in Kürze zurück.

Der Code sieht ziemlich dürftig aus, weil er beim *open* nicht auf bestimmte Geräte eingeht. Das muß auch nicht sein, weil die Geräte scull0-3 vom Design her global und persistent sind. Insbesondere gibt es kein “Initialisieren des Gerätes beim ersten Öffnen”, weil wir keinen Zähler der geöffneten *sculls* mitführen, nur den Verwendungszähler des Moduls.

In Anbetracht der Tatsache, daß der Kernel den Verwendungszähler des Moduls über das owner-Feld in der Struktur file\_operations verwalten kann, wundern Sie sich vielleicht, warum wir das hier manuell machen. Das liegt daran, daß Module in älteren Kerneln die gesamte Verwaltungsarbeit selbst machen mußten; es gab den owner-Mechanismus damals noch nicht. Um zu älteren Kerneln portabel zu sein, implementiert scull seinen eigenen Verwendungszähler. Dieses Verhalten führt dazu, daß der Verwendungszähler auf 2.4-Systemen zu groß ist, was aber kein Problem ist, denn er fällt gleichwohl auf Null zurück, wenn das Modul nicht verwendet wird.

Die einzige reale Operation, die hier auf dem Gerät vorgenommen wird, ist das Verkürzen auf eine Länge von Null, wenn das Gerät zum Schreiben geöffnet wird. Das Überschreiben des Gerätes mit einer kürzeren Datei führt zu einem kleineren Gerätedatenbereich, ähnlich wie das Öffnen einer normalen Datei zum Schreiben die Datei auf Null verkürzt. Die Operation tut nichts, wenn das Gerät zum Lesen geöffnet ist.

Später, wenn wir uns die anderen Personalities von scull anschauen, werden wir noch sehen, wie richtige Initialisierungen funktionieren.

## Die Methode release

Die Aufgabe der Methode *release* ist das Gegenstück zu *open*. Manchmal wird die Methodenimplementation auch *device*\_close anstelle von *device*\_release genannt. In jedem Falle sollte sie aber folgende Aufgaben erfüllen:

* deallozieren von allem, das *open* in filp->private\_data alloziert hat
* beim letzten Schließen das Gerät herunterfahren
* Verwendungszähler dekrementieren

ark=dash>

Die grundlegende Form von scull muß das Gerät nicht herunterfahren, so daß der benötigte Code minimal ist:[[2]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/x2969.html" \l "FTN.AEN3116)

|  |
| --- |
| int scull\_release (struct inode \*inode, struct file \*filp)  {  MOD\_DEC\_USE\_COUNT;  return 0;  } |

Das Dekrementieren des Verwendungszählers ist wichtig, weil der Kernel das Modul nicht entladen kann, solange der Zähler nicht auf Null zurückgeht.

Wie kann der Zähler konsistent bleiben, wenn eine Datei manchmal geschlossen wird, ohne vorher geöffnet worden zu sein? Wir wissen alle, daß *dup* und *fork* aus einer geöffneten Datei zwei machen, ohne *open* aufzurufen, aber jede dieser Dateien wird beim Beenden des Programms wieder geschlossen. Beispielsweise öffnen die meisten Programme ihre stdin-Datei (oder ihr stdin-Gerät) nicht, aber alle schließen sie letztendlich.

Die Antwort ist einfach: Nicht jeder *close*-Systemaufruf verursacht einen Aufruf der Methode release. Nur diejenigen, die wirklich die Gerätedatenstruktur freigeben, rufen auch die Methode auf — daher auch der Name. Der Kernel verwaltet einen Zähler, der registriert, wie oft eine file-Struktur verwendet wird. Weder *fork* noch *dup* erzeugen eine neue file-Struktur (das tut nur *open*); sie inkrementieren nur den Zähler in der existierenden Struktur.

Der Systemaufruf *close* führt die Methode *release* nur dann aus, wenn der Zähler der Struktur auf Null fällt, was genau dann passiert, wenn die Struktur zerstört wird. Dieser Zusammenhang zwischen der Methode release und dem Systemaufruf *close* garantiert, daß der Verwendungszähler von Modulen immer konsistent ist.

Beachten Sie, daß die Methode *flush* immer dann aufgerufen wird, wenn eine Applikation *close* aufruft. Nur wenige Treiber implementieren *flush* aber, weil es normalerweise nichts beim Schließen zu tun gibt, solange *release* nicht im Spiel ist.

Wie Sie sich vielleicht vorstellen können, gilt all das hier Gesagte auch, wenn die Applikation beendet wird, ohne explizit ihre offenen Dateien zu schließen: Der Kernel schließt automatisch alle offenen Dateien beim Beenden des Prozesses, indem er den Systemaufruf *close* intern aufruft.

### Fußnoten

|  |  |
| --- | --- |
| [[1]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/x2969.html" \l "AEN3024) | Die Aufteilung in verschiedene Bits ist eine typische Anwendungsweise für Minor-Nummern. Der IDE-Treiber verwendet beispielsweise die oberen beiden Bits für die Plattennummer und die unteren sechs Bits für die Partitionsnummer. |
| [[2]](http://www.oreilly.de/german/freebooks/linuxdrive2ger/x2969.html" \l "AEN3116) | Die anderen Arten des Gerätes werden von anderen Funktionen geschlossen, weil *scull\_open* andere Werte in filp->f\_op eingesetzt hat. Diese Funktionen werden wir später sehen. |

**Die Verwendung von Speicher in scull**

Bevor wir die *read*- und *write*-Operationen einführen, sollten wir besser noch einen Blick darauf werfen, wie und warum scull Speicher alloziert. Das “Wie” müssen wir wissen, um den Code gründlich zu verstehen, und das “Warum” demonstriert, welche Entscheidungen ein Treiber-Programmierer treffen muß, obwohl scull sicherlich nicht typisch für ein Gerät ist.

In diesem Abschnitt behandeln wir nur, wie Speicher in scull alloziert wird, und gehen nicht auf die anderen Bereiche der Hardware-Verwaltung ein, die Sie kennen müssen, um richtige Treiber zu schreiben. Diese Kenntnisse erwerben Sie in [Kapitel 8](http://www.oreilly.de/german/freebooks/linuxdrive2ger/hard.html) und [Kapitel 9](http://www.oreilly.de/german/freebooks/linuxdrive2ger/irq.html). Daher können Sie diesen Abschnitt überspringen, wenn Sie nicht daran interessiert sind, wie der speicherorientierte scull-Treiber arbeitet.

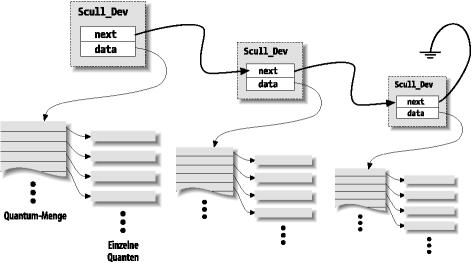
Die Region im Speicher, die von scull verwendet wird und die wir hier auch ein *Gerät* nennen, hat eine variable Länge. Je mehr Sie schreiben, um so größer wird sie. Der Speicherbereich kann wieder verkürzt werden, indem das Gerät mit einer kürzeren Datei überschrieben wird.

Die Implementation, die wir für scull gewählt haben, ist nicht so besonders intelligent. Aber der Quelltext für eine intelligentere Implementation wäre schwieriger zu lesen, und in diesem Kapitel geht es darum, sich mit *read* und *write* zu beschäftigen, und nicht mit der Speicherverwaltung. Darum verwendet der Code nur *kmalloc* und *kfree*, ohne darauf zurückzugreifen, ganze Seiten zu allozieren, auch wenn das effizienter wäre.

Andererseits wollten wir aber auch die Größe des *Geräte*-Bereiches nicht beschränken, sowohl aus prinzipiellen als auch aus praktischen Gründen. Prinzipiell betrachtet, sind willkürliche Beschränkungen bezüglich der Größe der verwalteten Daten immer schlecht, und praktisch gesehen kann scull dazu verwendet werden, um nach und nach den gesamten Systemspeicher zu verbrauchen, um zu testen, wie sich Ihre Programme bei geringem Speicher verhalten. Mit solchen Tests können Sie auch die Interna des Systems besser verstehen. Sie können den Befehl *cp /dev/zero /dev/scull0* verwenden, um das gesamte RAM von scull verbrauchen zu lassen, oder mit dem Hilfsprogramm dd selbst entscheiden, wie viele Dateien in das scull-Gerät kopiert werden sollen.

In scull ist jedes Gerät eine verkette Liste von Zeigern, die alle auf ein Scull\_Dev zeigen. Jede dieser Strukturen kann defaultmäßig über ein Array von Zwischenzeigern auf maximal vier Millionen Bytes verweisen. Die Quellen verwenden ein Array von 1000 Zeigern auf Bereiche von jeweils 4000 Bytes. Wir nennen jeden Speicherbereich ein *Quantum* und das Array (oder seine Länge) eine *Quantum-Menge*. Ein scull-Gerät und seine Speicherbereiche sind in [Abbildung 3-1](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charscullmem.html#FIG3-QUANTA) zu sehen.

**Abbildung 3-1. Das Layout eines scull-Geräts**



Die Zahlen sind so gewählt, daß das Schreiben eines einzelnen Bytes in scull acht- oder zwölftausend Bytes im Speicher verbraucht: viertausend für das Quantum und vier- oder achttausend für die Quantum-Menge (je nachdem, ob ein Zeiger auf der Zielplattform 32 oder 64 Bits groß ist). Auf der anderen Seite ist der Verwaltungsaufwand beim Durchlaufen der verketteten Liste durchaus nicht so schlecht, wenn große Datenmengen geschrieben werden, weil es nur ein Listenelement für vier Megabyte Daten gibt und die maximale Größe des Gerätes auf einige wenige Megabytes beschränkt ist, da es nicht größer als der Computerspeicher werden kann.

Die Wahl passender Werte für das Quantum und die Quantum-Menge ist eine Frage der Policy und nicht des Mechanismus; die optimalen Werte hängen davon ab, wie das Gerät verwendet wird. Der scull-Treiber sollte also keine bestimmten Werte für das Quantum und die Quantum-Menge erzwingen. In scull kann der Benutzer die Werte auf verschiedene Art und Weise verändern: durch das Ändern der Makros SCULL\_QUANTUM und SCULL\_QSET in scull.h beim Kompilieren, durch das Setzen der Integer-Werte scull\_quantum und scull\_qset beim Laden des Moduls oder durch das Ändern der aktuellen und der Default-Werte mittels *ioctl*-Aufrufen zur Laufzeit.

Die Verwendung eines Makros und eines Integer-Wertes, um eine Konfiguration sowohl zur Kompilierungs- als auch zur Ladezeit zu ermöglichen, erinnert daran, wie wir die Major-Nummer ausgewählt haben. Wir verwenden diese Technik immer, wenn ein Wert im Treiber willkürlich oder Policy-abhängig ist.

Die einzige verbleibende Frage ist, welche Default-Werte gewählt werden sollten. In diesem speziellen Fall besteht das Problem darin, die beste Balance zwischen der Speicherverschwendung halbgefüllter Quanta und Quantum-Mengen und dem Verwaltungsaufwand beim Allozieren, Deallozieren und Verwalten der Zeiger-Verkettung zu finden, der auftritt, wenn die Quanta und die Quantum-Mengen klein sind.

Darüber hinaus muß auch noch das interne Design von *kmalloc* berücksichtigt werden. Wir werden hier aber nicht auf die Details eingehen. Die Innereien von *kmalloc* werden im Abschnitt [the Section called *Die ganze Wahrheit über kmalloc* in Kapitel 7](http://www.oreilly.de/german/freebooks/linuxdrive2ger/get.html#GETKMAL) in [Kapitel 7](http://www.oreilly.de/german/freebooks/linuxdrive2ger/get.html) seziert.

Die Default-Werte wurden in der Annahme gewählt, daß die meisten Programmierer mehr als vier Megabyte RAM haben und daß wahrscheinlich sehr große Datenmengen auf scull geschrieben werden. Die Besitzer großer Computer werden zum Testen mehrere zehn Megabyte auf das Gerät schreiben, auch wenn normalerweise nur ein paar Kilobyte transportiert werden werden.

Die Datenstruktur, die die Geräteinformation enthält, sieht folgendermaßen aus:

|  |
| --- |
| typedef struct Scull\_Dev {  void \*\*data;  struct Scull\_Dev \*next; /\* naechstes Listenelement \*/  int quantum; /\* die aktuelle Quantum-Groesse \*/  int qset; /\* die aktuelle Array-Groesse \*/  unsigned long size;  devfs\_handle\_t handle; /\* nur bei devfs verwendet \*/  unsigned int access\_key; /\* wird von sculluid und scullpriv verwendet \*/  struct semaphore sem; /\* Semaphor zum gegenseitigen Ausschluss \*/  } Scull\_Dev; |

Der folgende Code zeigt, wie Scull\_Dev in der Praxis verwendet wird, um Daten zu speichern. Die Funktion *scull\_trim* ist dafür zuständig, den gesamten Datenbereich freizugeben, und wird von *scull\_open* aufgerufen, wenn die Datei zum Schreiben geöffnet wird. Sie traversiert einfach die Liste und gibt alle vorgefundenen Quanta und Quantum-Mengen frei.

|  |
| --- |
| int scull\_trim(Scull\_Dev \*dev)  {  Scull\_Dev \*next, \*dptr;  int qset = dev->qset; /\* "dev" ist nicht null \*/  int i;  for (dptr = dev; dptr; dptr = next) { /\* alle Listenelemente \*/  if (dptr->data) {  for (i = 0; i < qset; i++)  if (dptr->data[i])  kfree(dptr->data[i]);  kfree(dptr->data);  dptr->data=NULL;  }  next=dptr->next;  if (dptr != dev) kfree(dptr); /\* alle außer dem ersten \*/  }  dev->size = 0;  dev->quantum = scull\_quantum;  dev->qset = scull\_qset;  dev->next = NULL;  return 0;  } |

**Ein kurze Einführung in Race Conditions**

Da Sie jetzt verstehen, wie die Speicherverwaltung von scull funktioniert, sollten Sie sich einmal das folgende Szenario anschauen: Zwei Prozesse, A und B, haben beide das gleiche scull-Gerät zum Schreiben geöffnet. Beide versuchen gleichzeitig, Daten auf das Gerät zu schreiben. Damit diese Operation durchgeführt werden kann, ist ein neues Quantum notwendig, also alloziert jeder Prozeß den benötigten Speicher und legt einen Zeiger darauf in der Quantum-Menge ab.

Das gibt Ärger. Weil beide Prozesse das gleiche scull-Gerät sehen, legen sie auch beide den neuen Speicher an der gleichen Stelle in der Quantum-Menge ab. Wenn A seinen Zeiger zuerst ablegt, dann überschreibt B diesen Zeiger danach. Der von A allozierte Speicher und die darin enthaltenen Daten gehen also verloren.

Dies ist eine klassische *Race Condition*: Das Ergebnis hängt davon ab, wer als erstes zum Zug kommt; und normalerweise passiert etwas Unerfreuliches. Auf Einprozessor-Linux-Systemen muß sich der scull-Code nicht mit solchen Problemen auseinandersetzen, weil Prozesse, die Kernel-Code ausführen, nicht gezwungen werden können, die CPU abzugeben. Auf SMP-Systemen ist die Sache leider komplizierter. Die beiden Prozesse A und B können gut und gern auf verschiedenen Prozessoren laufen und sich so wie beschrieben ins Gehege kommen.

Der Linux-Kernel stellt mehrere Mechanismen bereit, um Race Conditions zu vermeiden und mit ihnen umzugehen. Eine vollständige Beschreibung dieser Mechanismen müssen wir auf [Kapitel 9](http://www.oreilly.de/german/freebooks/linuxdrive2ger/irq.html)> verschieben, aber ein wenig können wir hier schon darauf eingehen.

Ein *Semaphor* ist ein allgemeiner Mechanismus zur Steuerung des Zugriffs auf Ressourcen. In seiner einfachsten Form kann ein Semaphor für *gegenseitigen Ausschluß* (*mutual exclusion*) verwendet werden; Prozesse, die Semaphore im Mutual Exclusion-Modus verwenden, werden daran gehindert, gleichzeitig den gleichen Code auszuführen oder auf die gleichen Daten zuzugreifen. Solche Semaphore nennt man oft *mutex*, was von “mutual exclusion” kommt.

Semaphore in Linux werden in <asm/semaphore.h> definiert. Ihr Typ ist struct semaphore, und Treiber sollten auf sie nur über das definierte Interface zugreifen. In scull wird ein Semaphor pro Gerät in der Struktur Scull\_Dev alloziert. Weil die Geräte vollständig unabhängig voneinander sind, ist es nicht notwendig, einen gegenseitigen Ausschluß über Gerätegrenzen hinweg zu erzwingen.

Semaphore müssen vor der ersten Verwendung durch Übergabe eines numerischen Arguments an *sema\_init* initialisiert werden. Wenn es um einen gegenseitigen Ausschluß geht (also darum, Threads vom gleichzeitigen Zugriff auf die gleichen Daten abzuhalten), dann kann das Semaphor mit dem Wert 1 initialisiert werden, was bedeutet, daß das Semaphor zur Verfügung steht. Der folgende Code aus der Initialisierungsfunktion des scull-Moduls (*scull\_init*) zeigt, wie die Semaphore im Zuge der Einrichtung des Geräts initialisiert werden.

|  |
| --- |
| for (i=0; i < scull\_nr\_devs; i++) {  scull\_devices[i].quantum = scull\_quantum;  scull\_devices[i].qset = scull\_qset;  sema\_init(&scull\_devices[i].sem, 1);  } |

Ein Prozeß, der in einen von einem Semaphor geschützten Code-Bereich eintreten möchte, muß zunächst sicherstellen, daß sich darin noch kein anderer Prozeß befindet. In der klassischen Informatik heißt die Funktion zum Erwerb eines Semaphors oft *P*, aber unter Linux müssen Sie *down* oder *down\_interruptible* aufrufen. Diese Funktionen fragen den Wert des Semaphors ab, um festzustellen, ob er größer als 0 ist; wenn das der Fall ist, dann wird das Semaphor dekrementiert, und die Funktion kehrt zurück. Ist das Semaphor dagegen 0, dann legen sich die Funktionen schlafen und versuchen es noch einmal, nachdem ein anderer Prozeß, der vermutlich das Semaphor freigegeben hat, sie aufgeweckt hat.

Die Funktion *down\_interruptible* kann von einem Signal unterbrochen werden, während *down* es nicht zuläßt, daß zwischendrin Signale ausgeliefert werden. In fast allen Fällen sollten Sie Signale zulassen; ansonsten riskieren Sie nicht-beendbare Prozesse und anderes unerwünschtes Verhalten. Das Zulassen von Signalen verkompliziert allerdings die Lage dahingehend, daß Sie immer abfragen müssen, ob die Funktion (hier *down\_interruptible*) unterbrochen wurde. Wie üblich gibt die Funktion 0 im Erfolgsfall und einen von Null verschiedenen Wert im Fehlerfall zurück. Wenn der Prozeß unterbrochen wurde, dann hat die Funktion auch keine Semaphore erworben, weswegen Sie auch nicht *up* aufrufen müssen. Ein typischer Aufruf zum Erwerben eines Semaphors sieht daher normalerweise so aus:

|  |
| --- |
| if (down\_interruptible (&sem))  return -ERESTARTSYS; |

Der Rückgabewert -ERESTARTSYS teilt dem System mit, daß die Operation von einem Signal unterbrochen wurde. Die Kernel-Funktion, die die Gerätemethode aufgerufen hat, wird entweder einen erneuten Aufruf versuchen oder -EINTR an die Applikation zurückgeben — je nachdem, wie die Applikation den Umgang mit Signalen konfiguriert hat. Natürlich kann es sein, daß Ihr Code vor dem Rücksprung erst etwas aufräumen muß, wenn er in diesem Modus unterbrochen wurde.

Ein Prozeß, der einen Semaphor erwirbt, muß diesen danach immer wieder freigeben. Die dafür zuständige Funktion heißt in der Informatik *V*; Linux nennt sie aber *up*. Ein einfacher Aufruf wie

|  |
| --- |
| up (&sem); |

inkrementiert den Wert des Semaphors und weckt alle Prozesse auf, die darauf warten, daß das Semaphor verfügbar wird.

Bei der Arbeit mit Semaphoren muß man sich vorsehen. Die vom Semaphor geschützten Daten müssen genau definiert sein, und *sämtlicher* Code, der auf diese Daten zugreift, muß zuerst das Semaphor erwerben. Code, der *down\_interruptible* verwendet, um das Semaphor zu erwerben, darf keine andere Funktion aufrufen, die ebenfalls versucht, diesen Semaphor zu erwerben; ansonsten kommt es zu einem Deadlock. Wenn es eine Routine in Ihrem Treiber versäumt, ein gehaltenes Semaphor wieder freizugeben (z.B. als Folge eines Rücksprungs nach einem Fehler), dann werden alle weiteren Versuche, das Semaphor zu erwerben, blockieren. Gegenseitiger Ausschluß ist immer ein kniffliges Problem und bedarf einer wohldefinierten und methodischen Herangehensweise.

In scull wird der gerätespezifische Semaphor dazu verwendet, die gespeicherten Daten vor unangemessenem Zugriff zu schützen. Sämtlicher Code, der auf das Feld data der Struktur Scull\_Dev zugreift, muß zunächst das Semaphor erworben haben. Um Deadlocks zu vermeiden, sollten nur Funktionen, die Geräte-Methoden implementieren, das Semaphor anfordern. Interne Routinen wie das oben gezeigte *scull\_trim* gehen davon aus, daß das Semaphor bereits erworben wurde. Solange dies gewährleistet ist, ist der Zugriff auf die Datenstruktur Scull\_Dev vor Race Conditions geschützt.

# read und write

> > > >

Die Methoden *read* und *write* führen eine ähnliche Aufgabe durch, sie kopieren also Daten aus dem und in den Applikationscode. Ihre Prototypen sind daher sehr ähnlich und können gleichzeitig eingeführt werden:

|  |
| --- |
| ssize\_t read(struct file \*filp, char \*buff,  size\_t count, loff\_t \*offp);  ssize\_t write(struct file \*filp, const char \*buff,  size\_t count, loff\_t \*offp); |

Bei beiden Methoden ist filp der file-Zeiger und count die Größe der angeforderten Datenübertragung. Das Argument buff zeigt auf den Benutzer-Puffer, der die zu schreibenden Daten enthält, oder auf den leeren Puffer, in den die gelesenen Daten hineingeschrieben werden sollen. offp ist schließlich ein Zeiger auf ein Objekt mit “langem Offset”, das die Dateiposition angibt, auf die der Benutzer zugreifen will. Der Rückgabewert ist ein “vorzeichenbehafteter Größentyp”, dessen Verwendung wir später behandeln.

Was den Datentransfer angeht, besteht die Hauptschwierigkeit der beiden Geräte-Methoden darin, die Daten zwischen dem Adreßraum des Kernels und dem des Benutzers hin- und herzutransportieren. Die Operation kann nicht auf die übliche Weise mittels Zeigern oder *memcpy* durchgeführt werden. User-Space-Adressen können aus einer Reihe von Gründen nicht direkt im Kernel-Space verwendet werden.

Ein großer Unterschied zwischen Adressen im Kernel-Space und Adressen im User-Space besteht darin, daß Speicher im User-Space ausgelagert werden kann. Wenn der Kernel auf einen Zeiger im User-Space zugreift, ist die zugehörige Seite möglicherweise nicht im Speicher vorhanden, und es wird ein Seitenfehler(*Page Fault*) erzeugt. Die in diesem Abschnitt und im Abschnitt [the Section called *Das Argument von ioctl benutzen* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html#EXIOCTLSW) in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html) eingeführten Funktionen verwenden ein paar versteckte Zaubertricks, um auch dann noch korrekt mit Seitenfehlern umzugehen, wenn die CPU sich gerade im Kernel-Space befindet.

Interessanterweise verwendete die x86-Version von Linux 2.0 völlig unterschiedliche Speichertabellen im User-Space und im Kernel-Space. User Space-Zeiger konnten überhaupt nicht vom Kernel-Space aus dereferenziert werden.

Wenn das Ziel-Gerät eine Erweiterungskarte anstelle von RAM ist, entsteht das gleiche Problem, weil der Treiber trotzdem noch Daten zwischen Benutzer-Puffern und dem Kernel-Space (sowie möglicherweise zwischen dem Kernel-Space und dem I/O-Speicher) übertragen muß.

Das Kopieren zwischen Spaces geschieht in Linux mit speziellen Funktionen, die in <asm/uaccess.h> definiert sind. Eine solche Kopie wird entweder durch eine generische (*memcpy* -artige) Funktion oder durch Funktionen, die für eine bestimmte Datengröße (char, short, int, long) optimiert sind, durchgeführt; die meisten davon werden im Abschnitt [the Section called *Das Argument von ioctl benutzen* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html#EXIOCTLSW)> in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html)> erklärt.

Der Code für *read* und *write* in scull muß ein gesamtes Datensegment in den Benutzer-Adreßraum oder aus ihm heraus kopieren. Diese Fähigkeit wird durch die folgenden Kernel-Funktionen bereitgestellt, die ein beliebiges Array von Bytes kopieren und das Herzstück jeder *read*- und *write*-Implementation sind:

|  |
| --- |
| unsigned long copy\_to\_user(void \*to, const void \*from,  unsigned long count);  unsigned long copy\_from\_user(void \*to, const void \*from,  unsigned long count); |

Obwohl sich diese Funktionen wie normale *memcpy*-Funktionen verhalten, müssen Sie ein wenig zusätzliche Vorsicht walten lassen, wenn Sie von Kernel-Code aus auf den User-Space zugreifen wollen. Die angesprochenen Seiten im User Space sind möglicherweise nicht im Speicher vorhanden, und der Page Fault-Handler kann den Prozeß schlafen legen, während die Seite geholt wird. Dies passiert beispielsweise, wenn die Seite aus dem Swap-Space geholt werden muß. Daraus folgt für den Treiberautor, daß jede Funktion, die auf den User-Space zugreift, reentrant sein muß und gleichzeitig mit anderen Treiberfunktionen laufen können muß (siehe auch Abschnitt [the Section called *Reentranten Code schreiben* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exblocking.html#EXREENTER)> in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html)>). Deswegen verwenden wir Semaphore, um den nebenläufigen Zugriff zu steuern.

Die Rolle der beiden Funktionen ist nicht darauf beschränkt, Daten in den oder aus dem User-Space zu kopieren: Sie überprüfen auch, ob der Zeiger in den User-Space gültig ist. Wenn das nicht der Fall ist, wird auch nicht kopiert; wenn aber während des Kopierens eine ungültige Adresse vorgefunden wird, werden nur Teile der Daten kopiert. In beiden Fällen ist der Rückgabewert die verbleibende Menge zu kopierender Daten. Der Code in scull überprüft diesen Wert und gibt -EFAULT zurück, wenn es sich nicht um 0 handelt.

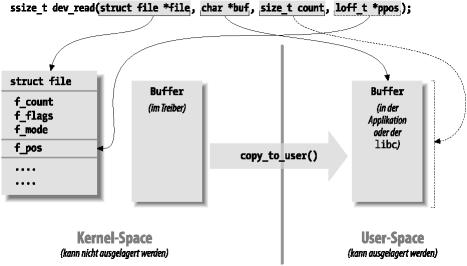
Das Thema des Zugriffs auf den User-Space sowie ungültiger User Space-Zeiger ist etwas komplizierter und wird im Abschnitt [the Section called *Das Argument von ioctl benutzen* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html#EXIOCTLSW)> in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html)> besprochen. Es lohnt sich aber zu wissen, daß Sie *\_ \_copy\_to\_user* und *\_ \_copy\_from\_user* verwenden können, wenn Sie sicher sind, den User-Space-Zeiger nicht überprüfen zu müssen, etwa, wenn Sie diesen bereits überprüft haben.

Was die eigentlichen Gerätemethoden angeht, so ist es die Aufgabe der *read*-Methode, Daten vom Gerät (unter Verwendung von *copy\_to\_user*) in den User-Space zu kopieren, während *write* Daten vom User-Space auf das Gerät kopieren muß (mittels *copy\_from\_user*). Jeder *read*- und *write*-Systemaufruf überträgt eine bestimmte Anzahl von Bytes, aber es ist dem Treiber freigestellt, weniger Daten zu übertragen — die genauen Regeln dafür unterscheiden sich beim Lesen und beim Schreiben etwas und werden weiter hinten in diesem Kapitel beschrieben.

Wie viele Daten die Methoden auch immer transportieren; sie sollten im allgemeinen die Dateiposition an \*offp aktualisieren, so daß diese die aktuelle Dateiposition nach einer erfolgreichen Beendigung des Systemaufrufs wiedergibt. Meistens ist das Argument offp nur ein Zeiger auf filp->f\_pos, aber zur Unterstützung der Systemaufrufe *pread* und *pwrite*, die das Gegenstück zu *lseek* und *read* oder *write* in einer einzigen atomaren Operation ausführen, wird ein anderer Zeiger verwendet.

Die Aufgabe der verschiedenen Argumente von *read* ist in [Abbildung 3-2](http://www.oreilly.de/german/freebooks/linuxdrive2ger/charrdwr.html#FIG3-READ) wiedergegeben.

**Abbildung 3-2. Die Argumente von read**



Sowohl die Methode *read* als auch die Methode *write* gibt einen negativen Wert zurück, wenn ein Fehler auftritt. Ein Rückgabewert größer oder gleich 0 teilt dem aufrufenden Programm mit, wie viele Bytes erfolgreich übertragen worden sind. Wenn einige Daten korrekt transportiert wurden und dann ein Fehler auftritt, ist der Rückgabewert trotzdem noch die Anzahl der erfolgreich übertragenen Bytes; der Fehler wird nicht vor dem nächsten Funktionsaufruf gemeldet.

Obwohl Kernel-Funktionen eine negative Zahl zurückgeben, um einen Fehler zu melden, und der Betrag dieser Zahl angibt, welcher Fehler aufgetreten ist (wie in [Kapitel 2](http://www.oreilly.de/german/freebooks/linuxdrive2ger/building.html) im Abschnitt [the Section called *Fehlerbehandlung in init\_module* in Kapitel 2](http://www.oreilly.de/german/freebooks/linuxdrive2ger/buiini.html#BUIERR) beschrieben), sehen Programme im User-Space immer nur -1 als Fehlercode. Diese Programme müssen auf die Variable errno zugreifen, um herauszufinden, was passiert ist. Dieser Unterschied liegt zum einen am POSIX-Aufrufstandard für Systemaufrufe und zum anderen am Vorteil, im Kernel nicht mit errno umgehen zu müssen.

## Die Methode read

Der Rückgabewert von *read* wird vom aufrufenden Programm folgendermaßen interpretiert:

* Wenn der Wert gleich dem Argument count ist, dann ist die angeforderte Zahl von Bytes übertragen worden. Dies ist der optimale Fall.
* Wenn der Wert positiv, aber kleiner als count ist, ist nur ein Teil der Daten übertragen worden. Das kann aus einer Reihe von Gründen passieren, die vom jeweiligen Gerät abhängen. Meistens wird das Programm noch einmal versuchen, die Daten zu lesen. Wenn Sie beispielsweise die Funktion *fread* verwenden, dann ruft die Bibliotheksfunktion den Systemaufruf noch einmal auf, um die angeforderte Datenübertragung noch einmal durchzuführen.
* Wenn der Wert Null ist, wird das als Erreichen des Dateiendes interpretiert.
* Ein negativer Wert weist auf einen Fehler hin. Der Wert gibt an, was für ein Fehler das war; die Werte stehen in <linux/errno.h>. Diese Fehler sehen aus wie -EINTR (unterbrochener Systemaufruf) oder -EFAULT (fehlerhafte Adresse).

ark=bullet>

In der obenstehenden Liste fehlt noch der Fall, daß keine Daten vorliegen, aber später eintreffen könnten. In diesem Fall sollte der Systemaufruf *read* blockiert werden. Wir behandeln blockierendes Lesen erst im Abschnitt [the Section called *Blockierende I/O* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exblocking.html) in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html).

Der Code von scull macht von diesen Regeln Gebrauch, insbesondere vom partiellen Lesen. Jeder Aufruf von *scull\_read* behandelt nur ein einziges Daten-Quantum; es gibt keine Schleife, um alle Daten auszulesen. Damit ist der Code kürzer und leichter zu lesen. Wenn das lesende Programm mehr Daten benötigt, kann es den Systemaufruf wiederholen. Wenn die Standardbibliothek (also *fread* und Konsorten) zum Lesen verwendet wird, wird die Applikation nicht einmal bemerken, daß der Datentransfer in Schüben erfolgt.

Wenn die aktuelle Leseposition größer als die Größe des Gerätes ist, gibt die *read*-Methode von scull 0 zurück, um mitzuteilen, daß keine Daten vorliegen (daß wir also mit anderen Worten am Dateiende sind). Diese Situation kann auftreten, wenn ein Prozeß A von dem Gerät liest, während Prozeß B das Gerät zum Schreiben öffnet und damit das Gerät auf eine Länge von 0 verkürzt. Prozeß A findet sich selbst auf einmal jenseits des Dateiendes wieder, und der nächste *read*-Aufruf gibt 0 zurück.

Der Code von *read* sieht wie folgt aus:

|  |
| --- |
| ssize\_t scull\_read(struct file \*filp, char \*buf, size\_t count,  loff\_t \*f\_pos)  {  Scull\_Dev \*dev = filp->private\_data; /\* das erste Listenelement \*/  Scull\_Dev \*dptr;  int quantum = dev->quantum;  int qset = dev->qset;  int itemsize = quantum \* qset; /\* wie viele Bytes sind im Listenelement \*/  int item, s\_pos, q\_pos, rest;  ssize\_t ret = 0;  if (down\_interruptible(&dev->sem))  return -ERESTARTSYS;  if (\*f\_pos >= dev->size)  goto out;  if (\*f\_pos + count > dev->size)  count = dev->size - \*f\_pos;  /\* Listenelement, qset-Index und Offset im Quantum suchen \*/  item = (long)\*f\_pos / itemsize;  rest = (long)\*f\_pos % itemsize;  s\_pos = rest / quantum; q\_pos = rest % quantum;  /\* der Liste bis zur richtigen Position (anderenorts definiert) folgen \*/  dptr = scull\_follow(dev, item);  if (!dptr->data)  goto out; /\* Loecher nicht füllen \*/  if (!dptr->data[s\_pos])  goto out;  /\* nur bis zum Ende dieses Quantums lesen \*/  if (count > quantum - q\_pos)  count = quantum - q\_pos;  if (copy\_to\_user(buf, dptr->data[s\_pos]+q\_pos, count)) {  ret = -EFAULT;  goto out;  }  \*f\_pos += count;  ret = count;  out:  up(&dev->sem);  return ret;  } |

## Die Methode write

*write* kann wie *read* weniger Daten übertragen, als angefordert waren. Die folgenden Regeln gelten für den Rückgabewert:

* Wenn der Wert gleich count ist, ist die angeforderte Anzahl von Bytes übertragen worden.
* Wenn der Wert positiv, aber kleiner als count ist, dann wurde nur ein Teil der Daten übertragen. Das Anwendungsprogramm wird wahrscheinlich versuchen, auch den Rest der Daten zu schreiben.
* Wenn der Wert 0 ist, ist nichts geschrieben worden. Dieses Ergebnis ist kein Fehler, weswegen auch kein Fehlercode zurückgegeben wird. Auch hier wird die Standardbibliothek *write* erneut aufrufen. Wir werden die Bedeutung dieses Falls im Abschnitt [the Section called *Blockierende I/O* in Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/exblocking.html) in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html) untersuchen, wenn wir auf blockierendes Schreiben eingehen.
* Ein negativer Wert weist auf einen aufgetretenen Fehler hin. Wie bei *read* sind die möglichen Fehlerwerte in <linux/errno.h> definiert.

ark=bullet>

Unglücklicherweise gibt es einige schlecht geschriebene Programme, die eine Fehlermeldung ausgeben und abbrechen, wenn Daten nur partiell übertragen worden sind. Das passiert, weil es manche Programmierer gewohnt sind, *write*-Aufrufe entweder ganz oder gar nicht gelingen zu sehen, was auch meistens passiert und von den Geräten ebenfalls unterstützt werden sollte. Diese Einschränkung in der Implementation von scull könnte behoben werden, aber wir wollten den Code nicht komplizierter als notwendig machen.

Der Code von *write* in scull behandelt wie der von read jeweils nur ein Quantum:

|  |
| --- |
| ssize\_t scull\_write(struct file \*filp, const char \*buf, size\_t count,  loff\_t \*f\_pos)  {  Scull\_Dev \*dev = filp->private\_data;  Scull\_Dev \*dptr;  int quantum = dev->quantum;  int qset = dev->qset;  int itemsize = quantum \* qset;  int item, s\_pos, q\_pos, rest;  ssize\_t ret = -ENOMEM; /\* der Wert wird in "goto out"-Anweisungen verwendet \*/  if (down\_interruptible(&dev->sem))  return -ERESTARTSYS;  /\* Listenelement, qset-Index und Offset suchen \*/  item = (long)\*f\_pos / itemsize;  rest = (long)\*f\_pos % itemsize;  s\_pos = rest / quantum; q\_pos = rest % quantum;  /\* der Liste bis zur richtigen Position folgen \*/  dptr = scull\_follow(dev, item);  if (!dptr->data) {  dptr->data = kmalloc(qset \* sizeof(char \*), GFP\_KERNEL);  if (!dptr->data)  goto out;  memset(dptr->data, 0, qset \* sizeof(char \*));  }  if (!dptr->data[s\_pos]) {  dptr->data[s\_pos] = kmalloc(quantum, GFP\_KERNEL);  if (!dptr->data[s\_pos])  goto out;  }  /\* nur bis zum Ende dieses Quantums schreiben \*/  if (count > quantum - q\_pos)  count = quantum - q\_pos;  if (copy\_from\_user(dptr->data[s\_pos]+q\_pos, buf, count)) {  ret = -EFAULT;  goto out;  }  \*f\_pos += count;  ret = count;  /\* die Groeße aktualisieren \*/  if (dev->size < \*f\_pos)  dev-> size = \*f\_pos;  out:  up(&dev->sem);  return ret;  } |

## readv und writev

Unix-Systeme unterstützen seit langem zwei alternative Systemaufrufe namens *readv* und *writev*. Diese “Vektorversionen” erwarten ein Array von Strukturen, die jeweils einen Zeiger auf einen Puffer und einen Längenwert enthalten. Ein *readv*-Aufruf liest dann der Reihe nach die angegebene Menge in jeden Puffer. *writev* dagegen sammelt die Inhalte der Puffer und schreibt sie in einer einzigen Schreiboperation heraus.

Bis zur Kernel-Version 2.3.44 emulierte Linux allerdings *readv* und *writev* immer durch mehrfache *read*- und *write*-Aufrufe. Wenn Ihr Treiber keine Methoden für Vektoroperationen bereitstellt, dann ist das auch immer noch der Fall. In vielen Situationen erreicht man aber eine größere Effizienz, wenn man *readv* und *writev* direkt im Treiber implementiert.

Die Prototypen der Vektoroperationen lauten:

|  |
| --- |
| ssize\_t (\*readv) (struct file \*filp, const struct iovec \*iov,  unsigned long count, loff\_t \*ppos);  ssize\_t (\*writev) (struct file \*filp, const struct iovec \*iov,  unsigned long count, loff\_t \*ppos); |

Hier sind die Argumente filp und ppos die gleichen wie bei *read* und *write*. Die in <linux/uio.h> definierte Struktur iovec sieht folgendermaßen aus:

|  |
| --- |
| struct iovec  {  void \*iov\_base;  \_&thinsp;\_kernel\_size\_t iov\_len;  }; |

Jedes iovec beschreibt einen zu übertragenden Datenblock, der bei iov\_base (im User-Space) beginnt und iov\_len Bytes lang ist. Der Parameter count gibt an, wie viele iovec-Strukturen es gibt. Diese Strukturen werden von der Applikation erzeugt, aber der Kernel kopiert sie vor dem Aufruf des Treibers in den Kernel-Space.

Die einfachste Implementation der Vektoroperationen ist eine einfache Schleife, die einfach die Adresse und Länge jedes iovec-Blocks an die *read*- oder *write*-Funktion des Treibers übergibt. Oft muß der Treiber für ein effizientes und korrektes Verhalten etwas schlauer sein. Beispielsweise sollte ein *writev* auf einem Bandlaufwerk sämtliche Daten aus den iovec-Strukturen in einem einzigen Datensatz auf das Band schreiben.

Viele Treiber haben aber keinen Vorteil darin, diese Methoden selbst zu implementieren. Daher werden sie in scull einfach weggelassen. Der Kernel emuliert sie mit *read* und *write*, und das Ergebnis ist das gleiche.

# Die neuen Geräte ausprobieren

Wenn Sie die vier gerade beschriebenen Methoden haben, kann der Treiber kompiliert und getestet werden; er speichert alle Daten, die Sie in ihn schreiben, bis sie mit neuen Daten überschrieben werden. Das Gerät arbeitet wie ein Daten-Puffer, dessen Länge nur durch die tatsächlich verfügbare RAM-Menge beschränkt ist. Sie können cp, dd und die Ein-/Ausgabeumleitung verwenden, um den Treiber zu testen.

Sie können den Befehl free benutzen, um zu sehen, wie die Menge freien Speichers sinkt und wieder ansteigt, wenn Daten in scull geschrieben werden.

Um sich besser daran zu gewöhnen, daß nur jeweils ein Quantum geschrieben wird, können Sie *printk*-Aufrufe an passenden Stellen in den Kernel einfügen und beobachten, was passiert, wenn eine Applikation große Datenmengen liest oder schreibt. Alternativ dazu können Sie auch das Hilfsprogramm strace benutzen, um die vom Programm durchgeführten Systemaufrufe zu beobachten. Wenn Sie damit cp oder auch ls -l > /dev/scull0 verfolgen, werden Sie die aufgeteilten Lese- und Schreib-Anweisungen verfolgen können. Im nächsten Kapitel werden wir auf Monitoring- und Debugging-Techniken etwas detaillierter eingehen.

# Das Device-Dateisystem

Wie bereits am Anfang des Kapitels angedeutet, enthalten die neueren Versionen des Linux-Kernels ein spezielles Dateisystem für Geräte-Einsprungpunkte. Dieses Dateisystem gibt es seit einiger Zeit als unoffiziellen Patch. In den offiziellen Quell-Baum wurde es mit Version 2.3.46 aufgenommen. Eine Rückportierung auf 2.2 steht ebenfalls zur Verfügung, ist aber nicht in den offiziellen 2.2-Kerneln enthalten.

Auch wenn die Verwendung dieses speziellen Dateisystems derzeit noch nicht besonders weit verbreitet ist, bieten die neuen Features doch einige Vorteile für Autoren von Gerätetreibern. Deswegen verwendet unsere Version von scull das devfs, wenn dieses auf dem Zielsystem vorhanden ist. Das Modul verwendet Konfigurationsinformationen des Kernels zur Compile-Zeit, um herauszufinden, ob bestimmte Features eingeschaltet worden sind; in diesem Fall geht es darum festzustellen, ob CONFIG\_DEVFS\_FS definiert ist oder nicht.

Die wichtigsten Vorteile von devfs sind:

* Geräte-Einsprungpunkte in /dev werden bei der Initialisierung des Geräts erzeugt und beim Entfernen des Geräts ebenfalls wieder entfernt.
* Der Gerätetreiber kann Gerätenamen, Besitzer und Zugriffsrechte angeben, aber Programme im User-Space können trotzdem noch den Besitzer und die Zugriffsrechte ändern (aber nicht den Dateinamen).
* Man muß keine Major-Nummer für den Treiber anfordern und sich nicht mit Minor-Nummern auseinandersetzen.

Als Folge daraus ist es nicht notwendig, beim Laden oder Entladen eines Moduls ein Skript auszuführen, das die Gerätedateien erzeugt, weil der Treiber durch die Verwaltung seiner eigenen Gerätedateien autonom ist.

Um Geräte zu erzeugen und zu entfernen, sollte der Treiber die folgenden Funktionen aufrufen:

|  |
| --- |
| #include <linux/devfs\_fs\_kernel.h>  devfs\_handle\_t devfs\_mk\_dir (devfs\_handle\_t dir,  const char \*name, void \*info);  devfs\_handle\_t devfs\_register (devfs\_handle\_t dir,  const char \*name, unsigned int flags,  unsigned int major, unsigned int minor,  umode\_t mode, void \*ops, void \*info);  void devfs\_unregister (devfs\_handle\_t de); |

Die devfs-Implementation enthält noch weitere Funktionen zur Verwendung durch Kernel-Code. Diese ermöglichen die Erzeugung symbolischer Links, den Zugriff auf die internen Datenstrukturen, um devfs\_handle\_t-Elemente aus Inodes zu bekommen, sowie weitere Aufgaben. Wir behandeln diese Funktionen hier nicht, weil sie entweder nicht wichtig oder aber schwer zu verstehen sind. Neugierige Leser finden weitere Informationen in der Header-Datei.

Die diversen Argumente der Funktionen zum Registrieren und Deregistrieren lauten:

dir

Das Eltern-Verzeichnis, in dem die neue Gerätedatei erzeugt werden soll. Die meisten Treiber werden hier NULL angeben, um die Dateien im Verzeichnis /dev zu erzeugen. Um ein eigenes Verzeichnis zu erzeugen, sollte der Treiber *devfs\_mk\_dir* verwenden.

name

Der Name des Geräts ohne das führende /dev. Der Name kann Schrägstriche enthalten, wenn das Gerät in einem Unterverzeichnis stehen soll; das Unterverzeichnis wird dann während der Registrierung erzeugt. Alternativ dazu können Sie auch einen gültigen dir-Zeiger auf das gewünschte Unterverzeichnis angeben.

flags

Eine Bitmaske mit devfs-Flags. DEVFS\_FL\_DEFAULT kann eine gute Wahl sein; DEVFS\_FL\_AUTO\_DEVNUM ist das Flag für die automatische Zuweisung von Major- und Minor-Nummern. Welche Flags es gibt und was sie bedeuten, beschreiben wir später.

major, minor

Die Major- und Minor-Nummern des Geräts. Sie werden nicht verwendet, wenn DEVFS\_FL\_AUTO\_DEVNUM in den Flags angegeben worden ist.

mode

Die Zugriffsrechte auf das neue Gerät.

ops

Ein Zeiger auf die Dateioperationsstruktur des Geräts.

info

Ein Default-Wert für filp->private\_data. Das Dateisystem initialisiert den Zeiger mit diesem Wert, wenn das Gerät geöffnet wird. Der an devfs\_mk\_dir übergebene info-Zeiger wird nicht von devfs verwendet und dient als “Benutzerdaten”-Zeiger.

de

Ein “*devfs*-Eintrag”, der aus einem früheren Aufruf von *devfs\_register* stammt.

Diese Flags werden verwendet, um bestimmte Eigenschaften der zu erzeugenden Gerätedatei auszuwählen. Sie sind zwar kurz und deutlich in <linux/devfs\_fs\_kernel.h> dokumentiert, aber es lohnt sich trotzdem, sie hier zu beschreiben.

DEVFS\_FL\_NONE, DEVFS\_FL\_DEFAULT

Das zuerst genannte Symbol ist einfach 0 und sollte verwendet werden, um den Code besser lesbar zu machen. Das zweite Makro ist derzeit als DEVFS\_FL\_NONE definiert, ist aber eine gute Wahl, um aufwärtskompatibel mit zukünftigen Implementationen des Dateisystems zu sein.

DEVFS\_FL\_AUTO\_OWNER

Dieses Flag läßt das Gerät so aussehen, als gehörte es der letzten Benutzer-/Gruppen-ID, die es geöffnet hat; außerdem ist es les- und schreibbar für alle, wenn es derzeit von keinem Prozeß geöffnet ist. Dieses Feature ist für TTY-Gerätedateien nützlich, aber auch für Gerätetreiber interessant, die den gleichzeitigen Zugriff auf ein Gerät verhindern müssen, das nicht gemeinsam genutzt werden kann. Um solche Fragen kümmern wir uns in [Kapitel 5](http://www.oreilly.de/german/freebooks/linuxdrive2ger/extra.html)>.

DEVFS\_FL\_SHOW\_UNREG, DEVFS\_FL\_HIDE

Das erste Flag legt fest, daß die Gerätedatei bei der Deregistrierung nicht aus /dev entfernt wird; das zweite bestimmt, daß die Datei nie in /dev erscheint. Diese Flags sind für normale Geräte üblicherweise nicht notwendig.

DEVFS\_FL\_AUTO\_DEVNUM

Alloziert automatisch eine Gerätenummer für dieses Gerät. Die Nummer gehört auch nach der Deregistrierung des devfs-Eintrags zum Gerät. Wenn der Treiber also erneut geladen wird, bevor das System heruntergefahren worden ist, bekommt er das gleiche Paar aus Major- und Minor-Nummer.

DEVFS\_FL\_NO\_PERSISTENCE

Den Eintrag nicht verfolgen, nachdem er entfernt worden ist. Dieses Flag spart nach dem Entfernen des Moduls etwas Systemspeicher, allerdings auf Kosten der Persistenz von Gerätemerkmalen über das Entladen und erneute Laden hinweg. Die persistenten Features sind die Zugriffsrechte, die Eigentümerschaft und die Major- und Minor-Nummern.

Es ist möglich, die zu einem Gerät gehörenden Flags zur Laufzeit abzufragen oder zu verwenden. Dazu dienen die beiden folgenden Funktionen:

|  |
| --- |
| int devfs\_get\_flags (devfs\_handle\_t de, unsigned int \*flags);  int devfs\_set\_flags (devfs\_handle\_t de, unsigned int flags); |

## devfs in der Praxis

Weil devfs zu ernsthaften Inkompatibilitäten hinsichtlich Gerätenamen im User-Space führt, benutzen es nicht alle Systeme. Unabhängig davon, wie dieses neue Feature von den Linux-Benutzern akzeptiert wird, ist es nicht so wahrscheinlich, daß Sie in näherer Zeit reine devfs-Treiber schreiben werden, weswegen Sie auch das “alte” Verfahren mit dem Erzeugen von Dateien und Zugriffsrechten im User-Space und mit der Verwendung von Major- und Minor-Nummern im Kernel-Space unterstützen müssen.

Der Code zur Initialisierung eines Gerätetreibers, der nur funktioniert, wenn devfs installiert ist, ist eine Untermenge des Codes, den Sie für beide Umgebungen brauchen, so daß wir Ihnen hier nur die Dual-Mode-Initialisierung zeigen. Anstatt einen speziellen Treiber zu schreiben, der devfs verwendet, fügen wir einfach devfs-Unterstützung zum scull-Treiber hinzu. Wenn Sie scull in einen Kernel laden, der devfs verwendet, dann müssen Sie insmod direkt aufrufen, anstatt das Skript scull\_load zu verwenden.

Wir haben uns entschieden, ein Verzeichnis zu erzeugen, das alle Gerätedateien von scull enthalten soll, weil die Struktur von devfs in hohem Maße hierarchisch ist und es keinen Grund gibt, sich nicht an diese Konvention zu halten. Außerdem können wir so zeigen, wie ein Verzeichnis erzeugt und entfernt wird.

Der folgende Code in *scull\_init* kümmert sich um die Geräteerzeugung und verwendet ein Feld namens handle in der Gerätestruktur, um sich zu merken, welche Geräte bereits registriert worden sind:

|  |
| --- |
| /\* Wenn wir devfs haben, /dev/scull erzeugen, wo die Dateien hinkommen \*/  scull\_devfs\_dir = devfs\_mk\_dir(NULL, "scull", NULL);  if (!scull\_devfs\_dir) return -EBUSY; /\* Problem \*/  for (i=0; i < scull\_nr\_devs; i++) {  sprintf(devname, "%i", i);  devfs\_register(scull\_devfs\_dir, devname,  DEVFS\_FL\_AUTO\_DEVNUM,  0, 0, S\_IFCHR | S\_IRUGO | S\_IWUGO,  &scull\_fops,  scull\_devices+i);  } |

Dieser Code wird von den folgenden Zeilen aus *scull\_cleanup* vervollständigt:

|  |
| --- |
| if (scull\_devices) {  for (i=0; i<scull\_nr\_devs; i++) {  scull\_trim(scull\_devices+i);  /\* Die folgende Zeile ist nur für devfs \*/  devfs\_unregister(scull\_devices[i].handle);  }  kfree(scull\_devices);  }  /\* auch das ist nur für devfs \*/  devfs\_unregister(scull\_devfs\_dir); |

Ein Teil der obenstehenden Code-Fragmente wird mit #ifdef CONFIG\_DEVFS\_FS umgeben. Wenn das Feature im aktuellen Kernel nicht eingeschaltet ist, verwendet scull weiterhin *register\_chrdev*.

Um beide Umgebungen zu unterstützen, ist jetzt nur noch die Initialisierung von filp->f\_ops und filp->private\_data in der Gerätemethode *open* erforderlich. Der erste Zeiger wird so gelassen, wie er ist, weil die korrekten Datei-Operationen bereits in *devfs\_register* angegeben worden sind. Der zweite Zeiger muß nur von der *open*-Methode initialisiert werden, wenn er NULL ist, denn nur dann wird devfs nicht verwendet.

|  |
| --- |
| /\*  \* Wenn die privaten Daten nicht gueltig sind, dann verwenden wir kein  \* devfs und setzen also den Typ (aus der Minor-Nummer), um einen neuen  \* Wert für f\_op anzugeben.  \*/  if (!filp->private\_data && type) {  if (type > SCULL\_MAX\_TYPE) return -ENODEV;  filp->f\_op = scull\_fop\_array[type];  return filp->f\_op->open(inode, filp); /\* an open weiterleiten \*/  }  /\* Typ 0, Geraetenummer ueberpruefen (wenn private\_data nicht gültig ist) \*/  dev = (Scull\_Dev \*)filp->private\_data;  if (!dev) {  if (num >= scull\_nr\_devs) return -ENODEV;  dev = &scull\_devices[num];  filp->private\_data = dev; /\* für andere Methoden \*/  } |

Mit dem oben gezeigten Code kann das scull-Modul in einem System, das devfs verwendet, geladen werden. *ls -l /dev/scull* zeigt dann die folgende Ausgabe an:

|  |
| --- |
| crw-rw-rw- 1 root root 144, 1 Jan 1 1970 0  crw-rw-rw- 1 root root 144, 2 Jan 1 1970 1  crw-rw-rw- 1 root root 144, 3 Jan 1 1970 2  crw-rw-rw- 1 root root 144, 4 Jan 1 1970 3  crw-rw-rw- 1 root root 144, 5 Jan 1 1970 pipe0  crw-rw-rw- 1 root root 144, 6 Jan 1 1970 pipe1  crw-rw-rw- 1 root root 144, 7 Jan 1 1970 pipe2  crw-rw-rw- 1 root root 144, 8 Jan 1 1970 pipe3  crw-rw-rw- 1 root root 144, 12 Jan 1 1970 priv  crw-rw-rw- 1 root root 144, 9 Jan 1 1970 single  crw-rw-rw- 1 root root 144, 10 Jan 1 1970 user  crw-rw-rw- 1 root root 144, 11 Jan 1 1970 wuser |

Die Funktion der einzelnen Dateien ist die gleiche wie beim “normalen” *scull*-Modul; der einzige Unterschied besteht in den Pfadnamen der Gerätedateien: Was vorher /dev/scull0 war, ist jetzt /dev/scull/0.

## Portabilitätsprobleme mit devfs

Die Quelldateien von scull sind ein wenig komplizierter, weil sie unter Linux 2.0, 2.2 und 2.4 kompilierbar sein und laufen sollen. Diese Portabilitätsanforderungen führen zu mehreren Fällen bedingter Kompilierung auf der Basis von CONFIG\_DEVFS\_FS.

Glücklicherweise sind sich die meisten Entwickler darüber einig, daß #ifdef-Konstrukte von Übel sind, wenn sie im Rumpf von Funktionsdefinitionen (im Gegensatz zu Header-Dateien) stehen. Daher brauchen wir zusätzliche Mechanismen, wenn wir trotz der Verwendung von devfs #ifdefs in unserem Code vermeiden wollen. Gleichwohl findet sich noch bedingte Kompilation in scull, weil ältere Versionen der Kernel-Header-Dateien keine andere Unterstützung enthalten.

Wenn Ihr Code nur mit der Version 2.4 des Kernels verwendet werden soll, können Sie bedingte Kompilierung durch das Aufrufen von Kernel-Funktionen vermeiden, die den Treiber auf beide Arten initialisieren; eine der beiden Initialisierungen macht dabei gar nichts und meldet nur eine erfolgreiche Ausführung zurück. Hier folgt ein Beispiel, wie eine solche Initialisierung aussehen kann:

|  |
| --- |
| #include <devfs\_fs\_kernel.h>  int init\_module()  {  /\* Major-Nummer anfordern, macht nichts, wenn devfs verwendet wird \*/  result = devfs\_register\_chrdev(major, "name", &fops);  if (result < 0) return result;  /\* mit devfs registrieren, macht nichts, wenn devfs nicht verwendet wird \*/  devfs\_register(NULL, "name", /\* .... \*/ );  return 0;  } |

Sie können ähnliche Tricks in Ihren eigenen Header-Dateien verwenden, solange Sie dabei aufpassen, keine Funktionen umzudefinieren, die schon von den Kernel-Header-Dateien verwendet werden. Das Beseitigen von bedingter Kompilierung ist eine gute Sache, weil die Lesbarkeit des Codes verbessert und die Wahrscheinlichkeit von Fehlern verringert wird, denn der Compiler parst die gesamte Eingabedatei. Immer, wenn eine/die bedingte Kompilierung verwendet wird, besteht die Gefahr, daß sich Tipp- oder andere Fehler einschleichen, die unentdeckt bleiben, weil sie an einer Stelle stehen, die vom C-Präprozessor aufgrund eines #ifdef verworfen wird.

Hier zeigen wir als Beispiel, wie scull.h die bedingte Kompilierung im Aufräum-Teil des Programms vermeidet. Dieser Code ist über alle Kernel-Versionen hinweg portabel, weil er sich nicht darauf verläßt, daß devfs in den Header-Dateien bekannt ist:

|  |
| --- |
| #ifdef CONFIG\_DEVFS\_FS /\* nur, wenn verwendet, um Fehler in 2.0 zu vermeiden \*/  #include <linux/devfs\_fs\_kernel.h>  #else  typedef void \* devfs\_handle\_t; /\* #ifdef in der Struktur vermeiden \*/  #endif |

> > > > In sysdep.h wird nichts definiert, weil es sehr schwer ist, solche Hacks generisch genug zu implementieren, um allgemein verwendbar zu sein. Jeder Treiber sollte seine eigenen Bedürfnisse abdecken, um eine exzessive Verwendung von #ifdef im Funktionscode zu vermeiden. Wir haben uns außerdem entschlossen, im Beispiel-Code zu diesem Buch (mit der Ausnahme von scull), devfs nicht zu unterstützen. Wir hoffen, daß dieser Abschnitt hier ausreichend ist, um Ihnen das weitere Entdecken von devfs zu ermöglichen, wenn Sie das wünschen. Wir haben die devfs-Unterstützung aus den weiteren Beispielen herausgelassen, um den Code einfach zu halten.

# Abwärtskompatibilität

Wir haben in diesem Kapitel bisher die Programmierschnittstelle der Version 2.4 des Linux-Kernels beschrieben. Leider hat sich diese Schnittstelle im Laufe der Kernel-Entwicklung deutlich verändert. Diese Änderungen sind für sich genommen alle Verbesserungen, bringen aber wieder einmal Schwierigkeiten für diejenigen mit, die Treiber schreiben wollen, die mit verschiedenen Kernel-Versionen kompatibel sind.

Was dieses Kapitel angeht, so gibt es nur wenige sichtbare Unterschiede zwischen den Versionen 2.4 und 2.2. In Version 2.2 wurden allerdings viele der Prototypen der file\_operations-Methoden im Vergleich zur Version 2.0 geändert; außerdem wurde der Zugriff auf den User-Space deutlich verändert (und vereinfacht). Der Semaphor-Mechanismus war in Linux 2.0 noch nicht so weit entwickelt. Und schließlich wurde in der 2.1-Entwicklungsserie der Verzeichniseintrag-Cache (*dentry cache*) eingeführt.

## Änderungen in der Struktur für die Dateioperationen

Eine Reihe von Faktoren haben die Änderungen in den file\_operations-Methoden vorangetrieben. Die seit langem bestehende Grenze von 2 GByte für die Dateigröße brachte selbst zu Zeiten von Linux 2.0 Probleme mit sich. Daher wurde ab der 2.1-Serie der Typ loff\_t, ein 64-Bit-Wert, verwendet, um Dateipositionen und Längen anzugeben. Die Unterstützung großer Dateien war vor der Version 2.4 noch nicht vollständig in den Kernel integriert, aber ein großer Teil der Basisarbeit wurde schon vorher erledigt und mußte von Treiber-Entwicklern berücksichtigt werden.

Eine weitere in der 2.1-Entwicklung eingeführte Änderung war das Hinzufügen des Zeigerarguments f\_pos zu den Methoden *read* und *write*. Diese Änderung wurde vorgenommen, um die POSIX-Systemaufrufe *pread* und *pwrite* zu unterstützen, die explizit den Datei-Offset angeben, von dem Daten gelesen oder zu dem Daten geschrieben werden sollen. Ohne diese Systemaufrufe kann es bei Multithreading-Programmen zu Race Conditions kommen, wenn Daten in Dateien verschoben werden.

Fast alle Methoden in Linux 2.0 erwarteten ein explizites Zeigerargument inode. In der 2.1-Reihe wurde dieser Parameter aus mehreren Methoden entfernt, weil er selten benötigt wurde. Wenn Sie ihn noch brauchen, können Sie ihn immer noch aus dem filp-Argument gewinnen.

Als Ergebnis daraus sahen die Prototypen der häufig verwendeten Methoden in file\_operations in 2.0 so aus:

int (\*lseek) (struct inode \*, struct file \*, off\_t, int);

Beachten Sie, daß diese Methode in Linux 2.0 nicht *llseek*, sondern *lseek* heißt. Die Namensänderung soll anzeigen, daß Positionierungsoperationen jetzt mit 64-Bit-Offset-Werten vorgenommen werden können.

int (\*read) (struct inode \*, struct file \*, char \*, int);, int (\*write) (struct inode \*, struct file \*, const char \*, int);

Wie erwähnt hatten diese Funktionen in Linux 2.0 den inode-Zeiger als Argument, dafür aber kein Positionsargument.

void (\*release) (struct inode \*, struct file \*);

Im 2.0-Kernel konnte die Methode *release* nicht fehlschlagen und gab deswegen void zurück.

Es hat noch viele weitere Änderungen an der Struktur file\_operations gegeben; wir werden diese in den folgenden Kapiteln behandeln, wenn sie für uns relevant werden. Es lohnt sich aber gleichwohl auch jetzt schon, sich anzuschauen, wie portabler Code geschrieben werden kann, der die bisher geschriebenen Änderungen abdeckt. Die Änderungen an diesen Methoden sind umfassend, und es gibt keine einfache, elegante Möglichkeit, sie zu behandeln.

Der Beispiel-Code berücksichtigt diese Änderungen mit kleinen Wrapper-Funktionen, die zwischen der alten und der neuen API “übersetzen”. Diese Wrapper werden nur verwendet, wenn mit 2.0-Header-Dateien kompiliert wird, und ersetzen die “echten” Geräte-Methoden in der Struktur file\_operations. Der folgende Code implementiert die Wrapper für den scull-Treiber:

|  |
| --- |
| /\*  \* Die folgenden Wrapper bringen den Code mit 2.0-Kerneln zum Laufen  \*/  #ifdef LINUX\_20  int scull\_lseek\_20(struct inode \*ino, struct file \*f,  off\_t offset, int whence)  {  return (int)scull\_llseek(f, offset, whence);  }  int scull\_read\_20(struct inode \*ino, struct file \*f, char \*buf,  int count)  {  return (int)scull\_read(f, buf, count, &f->f\_pos);  }  int scull\_write\_20(struct inode \*ino, struct file \*f, const char \*b,  int c)  {  return (int)scull\_write(f, b, c, &f->f\_pos);  }  void scull\_release\_20(struct inode \*ino, struct file \*f)  {  scull\_release(ino, f);  }  /\* "Echte" Namen in 2.0-Namen umdefinieren \*/  #define scull\_llseek scull\_lseek\_20  #define scull\_read scull\_read\_20  #define scull\_write scull\_write\_20  #define scull\_release scull\_release\_20  #define llseek lseek  #endif /\* LINUX\_20 \*/ |

Das Umdefinieren der Namen auf diese Weise kann auch Struktur-Elemente abdecken, deren Namen sich mit der Zeit geändert haben (wie etwa im Falle der Änderung von *lseek* in *llseek*).

Wir müssen wahrscheinlich nicht extra sagen, daß Sie solche Umdefinitionen von Namen mit Vorsicht vornehmen sollten; diese Zeilen sollten vor der Definition der Struktur file\_operations, aber nach einer Veränderung dieser Namen stehen.

Es gibt noch zwei weitere Inkompatibilitäten im Zusammenhang mit der Struktur file\_operations. So wurde die Methode *flush* während der 2.1-Entwicklung hinzugefügt. Treiber-Entwickler müssen sich fast nie um diese Methode kümmern, aber ihre reine Anwesenheit mitten in der Struktur kann trotzdem noch zu Problemen führen. Sie können es am besten vermeiden, sich mit der *flush*-Methode auseinanderzusetzen, wenn Sie die markierte Initialisierungssyntax verwenden, wie wir das in allen Beispieldateien getan haben.

Der andere Unterschied besteht darin, wie ein inode-Zeiger aus der filp-Struktur geholt wird. Während moderne Kernel eine dentry-Struktur (directory entry, Verzeichniseintrag) verwenden, gab es eine solche Struktur im Kernel 2.0 nicht. Daher definiert sysdep.h ein Makro, das für den portablen Zugriff auf inode von einem filp verwendet werden soll.

|  |
| --- |
| #ifdef LINUX\_20  # define INODE\_FROM\_F(filp) ((filp)->f\_inode)  #else  # define INODE\_FROM\_F(filp) ((filp)->f\_dentry->d\_inode)  #endif |

## Der Verwendungszähler von Modulen

> > In 2.2 und früheren Kerneln gab es von seiten des Linux-Kernels keine Unterstützung für Module bei der Verwaltung des Verwendungszählers. Die Module mußten diese Arbeit selbst erledigen. Dieser Ansatz war fehlerträchtig und machte viel doppelte Arbeit notwendig.

Code, der portabel sein soll, muß aber auch mit dieser alten Vorgehensweise zurechtkommen. Das bedeutet, daß der Verwendungszähler weiterhin inkrementiert werden muß, wenn eine neue Referenz auf das Modul angelegt wird, und daß er dekrementiert werden muß, wenn diese Referenz wieder verschwindet. Portabler Code muß auch das Problem umgehen, daß das Feld owner in der Struktur file\_operations in früheren Kernel-Versionen nicht existiert. Das ist am einfachsten möglich, wenn Sie SET\_MODULE\_OWNER verwenden, anstatt auf das Feld owner zuzugreifen. In sysdep.h stellen wir eine leere Version von SET\_FILE\_OWNER für Kernel bereit, die nicht über dieses Feld verfügen.

## Änderungen in der Unterstützung von Semaphoren

Die Semaphor-Unterstützung war im 2.0-Kernel noch weniger weit entwickelt; ganz allgemein war die Unterstützung für SMP-Systeme zu diesem Zeitpunkt noch recht primitiv. Treiber, die nur für diese Kernel-Version geschrieben wurden, müssen gar keine Semaphore verwenden, weil nur jeweils eine CPU Kernel-Code ausführen konnte. Dennoch mag es hier die Notwendigkeit für Semaphore geben, und es schadet allemal nicht, den vollständigen Schutz späterer Kernel-Versionen zu haben.

Die meisten der in diesem Kapitel behandelten Semaphor-Funktionen existierten auch schon im Kernel 2.0. Die einzige Ausnahme bildet die Funktion *sema\_init*; im Kernel 2.0 mußten Entwickler die Semaphore manuell initialisieren. Die Header-Datei sysdep.h deckt dieses Problem durch Definition einer Version von *sema\_init* ab, die bei der Verwendung eines 2.0-Kernels benutzt wird:

|  |
| --- |
| #ifdef LINUX\_20  # ifdef MUTEX\_LOCKED /\* Nur, wenn semaphore.h eingebunden ist \*/  extern inline void sema\_init (struct semaphore \*sem, int val)  {  sem->count = val;  sem->waking = sem->lock = 0;  sem->wait = NULL;  }  # endif  #endif /\* LINUX\_20 \*/ |

## Änderungen im Zugriff auf den User-Space

Schließlich änderte sich am Anfang der 2.1-Reihe der Zugriff auf den User-Space total. Die neue Schnittstelle hat ein besseres Design und nutzt die Hardware für den sicheren Zugriff auf User Space-Speicher sehr viel besser aus. Aber natürlich hat sich die Schnittstelle geändert. Im 2.0-Kernel sahen die Funktionen für den Speicherzugriff folgendermaßen aus:

|  |
| --- |
| void memcpy\_fromfs(void \*to, const void \*from, unsigned long count);  void memcpy\_tofs(void \*to, const void \*from, unsigned long count); |

Die Namen dieser Funktionen stammen aus der historischen Verwendung des FS-Segment-Registers auf dem i386. Beachten Sie, daß diese Funktionen keinen Rückgabewert haben; wenn der Benutzer eine ungültige Adresse angibt, dann schlägt das Kopieren der Daten stillschweigend fehl. sysdep.h versteckt die Umbenennung und erlaubt Ihnen das portable Aufrufen von *copy\_to\_user* und *copy\_from\_user*.

**Schnellreferenz**

In diesem Kapitel wurden die folgenden Symbole und Header-Dateien eingeführt. Die Liste der Felder in struct file\_operations und struct file wird hier nicht wiederholt.

#include <linux/fs.h>

Die *Dateisystem*-Header-Datei wird zum Schreiben von Gerätetreibern benötigt. Alle wichtigen Funktionen sind hier definiert.

int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);

Registriert einen Zeichen-Gerätetreiber. Wenn die Major-Nummer nicht Null ist, wird diese unverändert übernommen, wenn die Nummer Null ist, wird eine dynamische Nummer zugewiesen.

int unregister\_chrdev(unsigned int major, const char \*name);

Deregistriert einen Treiber beim Entladen. Sowohl major als auch name müssen die gleichen Werte wie beim Registrieren enthalten.

kdev\_t inode->i\_rdev;

Die Geräte-Nummer des aktuellen Geräts steht über das inode-Argument zur Verfügung.

int MAJOR(kdev\_t dev);, int MINOR(kdev\_t dev);

Diese Makros extrahieren die Major- und Minor-Nummern aus einem Geräte-Wert.

kdev\_t MKDEV(int major, int minor);

Dieses Makro erzeugt einen kdev\_t-Wert aus den Major- und Minor-Nummern.

SET\_MODULE\_OWNER(struct file\_operations \*fops)

Dieses Makro setzt das owner-Feld in der angegebenen file\_operations-Struktur.

#include <asm/semaphore.h>

Definiert Funktionen und Typen rund um Semaphore.

void sema\_init (struct semaphore \*sem, int val);

Initialisiert einen Semaphor mit einem bekannten Wert. Mutexe werden normalerweise mit dem Wert 1 initialisiert.

int down\_interruptible (struct semaphore \*sem);, void up (struct semaphore \*sem);

Holt einen (wenn notwendig schlafenden) Semaphor bzw. gibt ihn wieder frei.

#include <asm/segment.h>, #include <asm/uaccess.h>

segment.h definiert Funktionen, die zum Kopieren von Daten zwischen User-Space und Kernel-Space in allen Kerneln bis einschließlich 2.0 verwendet werden. In Version 2.1 wurde der Name der Header-Datei in uaccess.h geändert.

unsigned long \_ \_copy\_from\_user (void \*to, const void \*from, unsigned long count);, unsigned long \_ \_copy\_to\_user (void \*to, const void \*from, unsigned long count);

Kopiert Daten zwischen User-Space und Kernel-Space.

void memcpy\_fromfs(void \*to, const void \*from, unsigned long count);, void memcpy\_tofs(void \*to, const void \*from, unsigned long count);

Diese Funktionen wurden in der Kernel-Version 2.0 dazu benutzt, ein Array von Bytes zwischen dem User-Space und dem Kernel-Space hin- und herzukopieren.

#include <linux/devfs\_fs\_kernel.h>, devfs\_handle\_t devfs\_mk\_dir (devfs\_handle\_t dir, const char \*name, void \*info);, devfs\_handle\_t devfs\_register (devfs\_handle\_t dir, const char \*name, unsigned int flags,, unsigned int major, unsigned int minor, umode\_t mode, void \*ops, void \*info);, void devfs\_unregister (devfs\_handle\_t de);

Die grundlegenden Funktionen zum Registrieren von Geräten im Geräte-Dateisystem (*devfs*).