Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 131

Leckerer Zaubertrank

Proc-Dateien sind nicht nur nützlich, sondern auch einfach zu erstellen. Damit eignen sie sich gut als Einstieg in die Kernel-Programmierung. Eva-Katharina Kunst, Jürgen Quade

Es ist kein Geheimnis, dass virtuelle Dateisysteme eine riesige Erfolgsgeschichte sind. Dateien, bei denen es sich nicht umDateien handelt, und statische Daten, die dynamisch während des Zugriffs generiert werden, gehören als essenzielle Zutaten zum Zaubertrank.

Die damit verbundene Abbildung auf die Operationen *Datei lesen* und *Datei schreiben* macht den Zugriff auf Systeminformationen beinahe zum Kinderspiel. Weil die Daten als ASCII-Zeichenketten verarbeitet werden, ist es möglich, mithilfe von `echo` und `cat` das System über eine Terminalverbindung ohne spezielle Software und ohne eine komplexe GUI überwachen und sogar konfigurieren.

Der Inhalt einer virtuellen Datei liegt nicht statisch auf einem Hintergrundspeicher wie einer SSD, einer altertümlichen Festplatte oder einem Flash-Speicher. Darum könnte man ihn beispielsweise mit einem zeichenorientierten Gerätetreiber zur Verfügung stellen. Das System generiert den Dateiinhalt dann dynamisch beim Lesezugriff beziehungsweise stößt bei einem Schreibzugriff hinterlegte Aktionen an.

Ordnung halten

Statt die bereits vorhandene Gerätetreibertechnik direkt zu verwenden, hat Chefentwickler Linus Torvalds mit Procfs ein virtuelles Dateisystem eingeführt. Der Clou bei diesem Vorgehen: Die darin abgelegten Dateien sind bereits intern

Die Autoren

Eva-Katharina Kunst ist bereits seit den Anfängen von Linux ein Fan von freier Software. **Jürgen Quade**, Professor an der Hochschule Niederrhein, gibt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux.

Verzeichnissen zugeordnet, was für eine saubere immanente Strukturierung sorgt.

Die entsprechenden Verzeichnisse entstehen intern durch Aufruf der Kernel-Funktion `proc_mkdir()`. Sie erhält als Parameter lediglich den Namen des neuen Unterordners und eine Referenz auf das übergeordnete Verzeichnis, in dem der Unterordner angelegt werden soll. Der Wert `NULL` repräsentiert dabei die Wurzel des Proc-Verzeichnisses. `proc_mkdir()` gibt eine Referenz auf das neu erstellte Verzeichnis zurück, um mit dieser Info innerhalb des neuen Ordners Proc-Dateien anzulegen oder gegebenenfalls weitere Unterverzeichnisse zu erstellen.

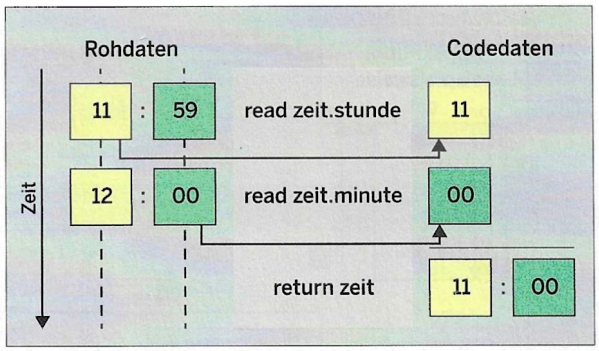
Häppchenweise lesen

Während diese Vorgehensweise für eine rundum positive User-Experience sorgt, müssen sich die Programmiererinnen und Programmierer mit den Problemen herumschlagen, die sich aus ständig alternierenden Daten ergeben: Aus einer virtuellen Datei nur häppchenweise konsumierte Daten können sich zwischen zwei Zugriffen bereits geändert haben. In dem Fall passt dann der erste Teil nicht mehr zum zweiten.

Ein Beispiel: Eine virtuelle Datei gibt beim Lesezugriff die aktuelle Uhrzeit in Stunden und Minuten zurück. Liest nun eine Applikation um 11:59 Uhr in einem ersten Happen die Stunden und kurz darauf – aber erst nach dem Vorrücken des Stundenzeigers auf die 12 – die Minuten, dann ist es für sie erst 11:00 Uhr 1. Ein ähnliches Problem ergibt sich, wenn die ASCII-Repräsentierung von Daten 2 zu unterschiedlichen Zeitpunkten unterschiedliche Längen aufweist. Auch dann passt Teil eins nicht mehr zu Teil zwei.

Dieses Problem lösen die meisten Implementierungen dadurch, dass sie beim ersten (lesenden) Zugriff den kompletten Datensatz in einen Puffer kopieren und bei nachfolgenden Zugriffen daraus die – jetzt eventuell veralteten – Daten auslesen. Das funktioniert, solange die Datenmenge überschaubar bleibt und sich nicht zu rasch inhaltlich ändert.

Um das Rad nicht zweimal zu erfinden, fehlerhaften Code zu vermeiden und es den Programmiererinnen und Programmierern etwas leichter zu machen, stellt Linux mit dem Singlefile eine Implementierung bereit, die dieses Problem löst 3. Damit erspart man sich, beim ersten Zugriff selbst per `kmalloc()` ausreichend Speicher zu reservieren, ihn mit den Daten zu füllen, den Speicher mit dem File Pointer zu verknüpfen, die Lese- und eventuell Schreibfunktion zu implementieren und nach den erfolgten Zugriffen den Speicher wieder freizugeben.



1 Greifen Applikationen häppchenweise zu, resultieren daraus möglicherweise falsche Daten.

Alles easy

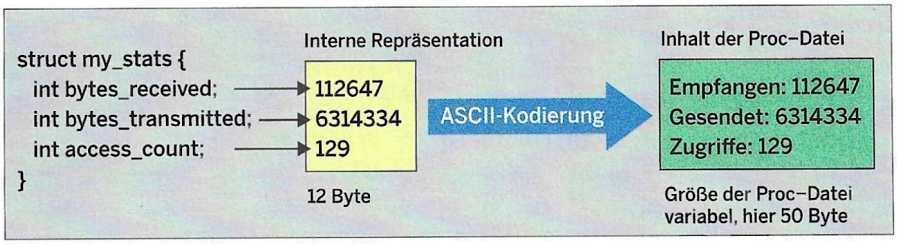
Proc-Filesystem und Singlefile gehören somit zusammen wie Topf und Deckel. Dank dieser Kombination sowie einiger Hilfsfunktionen lässt sich mit ungefähr 40 Zeilen Code eine Proc-Datei realisieren, die per Lesezugriff Daten aus dem Kernel ausschleust. Das bietet Ihnen einen idealen Einstieg in die Kernel-Programmierung, denn Sie implementieren neben der Modulinitialisierung mit dem Einklinken in das Proc-Filesystem nur eine Lesefunktion. Wer den Umgang mit `printf()` gewohnt ist, hat damit keinerlei Probleme, denn mit `seq_printf()` stellt der Kernel eine verwandte Funktion zur Verfügung. Sie schreibt die auszugebenden Daten in einen Puffer, aus dem sich die Applikationen bei ihren Lesezugriffen bedienen.

Listing 1 zeigt den Kernel-Code für die Proc-Datei `/proc/task_struct`, die ausgewählte Parameter der zentralen Datenstruktur `struct task_struct` ausgibt. Zur Erinnerung: `struct task_struct` repräsentiert den Task-Kontrollblock, also sämtliche für das Scheduling interessanten Informationen eines Tasks. Im Beispiel haben wir dazu den Namen des Tasks (Feld `comm`), die PID und die bislang verbrauchte Rechenzeit im Userland (`utime`) ausgesucht. Hinzu kommen die Rechenzeit, die der Kernel mit den Aufträgen der Task verbraucht hat (`stime`), die Realzeitpriorität und die generelle Priorität.

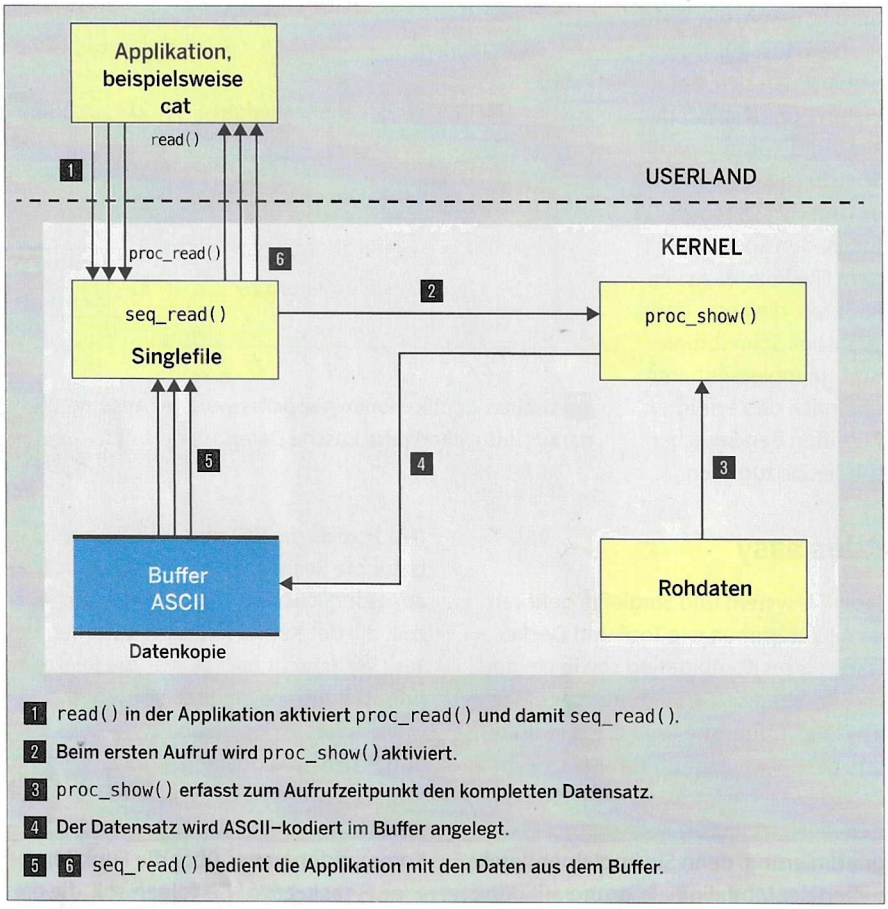
Die Adresse des gerade aktiven Tasks findet sich im globalen Zeiger `current` (Listing 1, Zeile 7). Der verwendete Buffer wird benötigt, weil das Auslesen des Kommandonamens über die Funktion `__get_task_comm()` erfolgen soll, die einen solchen Speicher erwartet. Die Funktion `seq_printf()` erhält als ersten Parameter die Referenz auf das sogenannte Sequence-File-Objekt. Das Sequence File dient als Basistechnologie, auf die das Singlefile aufsetzt.

Selbst machen

Die Funktion `ts_init()` ab Zeile 16 wird beim Laden des Moduls aufgerufen. Diese Information steht dem Kernel durch das Makro `module_init()` in Zeile 33 zur Verfügung. Wir verwenden die Initialisierungsroutine dazu, um unsere Proc-Datei über den Aufruf der Funktion `proc_create_single_data()` in das Proc-Filesystem einzuklinken (Zeile 19). Sie erhält



2 Bei Strings kann der häppchenweise Zugriff unschöne Formatierungen verursachen.



3 Dank Singlefile erhalten Applikationen konsistente Daten.

Listing 1: ts.c

```
01 #include <linux/module.h>
02 #include <linux/proc_fs.h>
03 #include <linux/seq_file.h>
04
05 static int ts_show(struct seq_file *m, void *v) {
06   char buffer[128];
07   seq_printf(m, "     comm: \"%s\"\n", __get_task_comm(buffer,
   sizeof(buffer), current));
08   seq_printf(m, "        pid: %d\n", current->pid);
09   seq_printf(m, "      stime: %llu\n", current->stime);
10   seq_printf(m, "      utime: %llu\n", current->utime);
11   seq_printf(m, "static_prio: %d\n", current->static_prio);
12   seq_printf(m, "rt_priority: %d\n", current->rt_priority);
13   return 0;
14 }
15
16 static int __init ts_init(void) {
17   static struct proc_dir_entry *procdirentry;
18   pr_debug("ts_init");
19   procdirentry=proc_create_single_data("task_struct", 0444, NULL,
   ts_show, NULL);
```

als ersten Parameter den Namen der neuen Datei. Es folgen die Zugriffsrechte in Oktalnotation, die Referenz auf das Verzeichnis, in dem die Datei eingebunden werden soll, sowie die Adresse der für die Datenausgabe zuständigen Funktion. Ein optionaler Parameter, der an die Proc-Datei-Referenz gebunden wird, den wir jedoch nicht weiter verwenden, beendet den Reigen.

Falls die Funktion `proc_create_single_data()` den Wert `NULL` zurückgibt, ist ein Fehler aufgetreten. Das kommt beispielsweise vor, wenn bereits eine Datei unter dem angeforderten Namen existiert. Ansonsten signalisiert ein Rückgabewert der Routine `ts_init()` von `0` die erfolgreiche Initialisierung. Die beim Entladen des Moduls aufgerufene Funktion `ts_exit()` sorgt via `remove_proc_entry()` (Zeile 30) für das Entfernen der als Parameter angegebenen Proc-Datei. Geben Sie hier ein Verzeichnis anstelle einer Datei an, löscht das Proc-Filesystem rekursiv erst den Verzeichnisinhalt und dann das Verzeichnis selbst. Die hie und da auftauchenden Funktionen `pr_err()` und `pr_info()` dienen der Ausgabe von (Debug-)Informationen im Syslog. Das können Sie in einem Terminal mittels des Kommandos `sudo tail -f /var/log/kern.log` fortwährend elegant einsehen.

Ein Makefile (Listing 2) steuert die Generierung des Kernel-Moduls `ts.ko` aus dem Quellcode `ts.c`. Achten Sie hier auf die Schreibweise `Makefile`, und erstellen Sie die Einrückungen per Tabulator.

Bringen Sie vor dem Erstellen des Moduls Ihr System auf den aktuellen Stand, und installieren Sie gegebenenfalls noch das Paket *build_essential* (Listing 3). Liegen das Makefile und der Quellcode `ts.c` im selben Verzeichnis, generiert ein simpler Aufruf von `make` das Modul 4. Mit Root-Rechten ausgestattet, laden Sie es per `insmod ts.ko` in den Kernel.

Nun steht im Verzeichnis `/proc/` die neue Datei `/proc/task_struct` zur Verfügung, die Sie beispielsweise per `cat /proc/task_struct` auslesen können.

Schreiben lernen

Wollen Sie neben dem Auslesen von Daten über eine Proc-Datei auch Konfigurationsarbeiten vornehmen, müssen Sie eine Schreibfunktion implementieren.

In diesem Fall lässt sich `proc_create_single_data()` nicht mehr ohne etwas Trickserei einsetzen. Sie erlaubt keinen direkten Zugriff auf die Struktur `struct proc_ops`, den Sie benötigen, um die Adresse der Schreibfunktion zu hinterlegen.

Die Struktur `struct proc_ops` speichert Adressen von Zugriffsfunktionen, die unter anderem mit den Syscalls `open()`, `close()`, `read()` und `write()` korrespondieren. Ruft eine Applikation `open()` mit dem Namen der Proc-Datei auf, aktiviert das im Modul die korrespondierende und zu implementierende Funktion `proc_open()`. Das gilt analog auch für `close()` (`proc_close()`), `read` (`proc_read`), `write()` (`proc_write()`) und noch ein paar weitere, in diesem Kontext irrelevante Funktionen.

Die Liste mit den Funktionsadressen (`struct proc_ops`) übergeben Sie dem Kernel per `proc_create()` oder `proc_create_data()` nebst dem Namen der neuen Proc-Datei, den Zugriffsrechten und dem Ort, an dem die Datei im Proc-Filesystem auftauchen soll.

Provokateure

Um per Proc-File eine Konfiguration vorzunehmen, müssen Sie im Modul die Methode `proc_write()` implementieren, die der Kernel aufruft, sobald eine Applikation auf die Proc-Datei schreibt. Sie kopiert die vom Programm beim Aufruf im Applikationsspeicher (Userspace) abgelegten Daten in den Kernel – ein direkter Zugriff ist hier ja nicht erlaubt. Dann interpretiert sie die Daten und führt die innerhalb der Funktion implementierte Aktion aus. Das alles muss aber sehr sorgsam passieren, damit nicht versehentlich (oder durch eine bösartige Applikation provoziert) mehr Daten in den Kernel wandern, als der dort dafür vorgesehene Speicherplatz aufnehmen kann. Zum Datentransfer selbst dient die Funktion `copy_from_user()`, die Sie mit dem Wohin, Woher und Wie viel parametrieren.

Ein Beispiel soll die Zusammenhänge klarer machen. Listing 4 zeigt den Quellcode für das Modul `procfiles.ko`, mit dem sich dynamisch Proc-Dateien unterhalb des Proc-Verzeichnisses `/proc/linux-magazin/` anlegen lassen. Dazu erzeugt das Modul zuerst einmal die Proc-Datei `create`. Für jede dort hineingeschriebene ASCII-codierte Zahl erzeugt das Modul eine weitere Proc-Datei, deren Name der Nummer entspricht. Beim Entladen des Moduls werden sämtliche Proc-Dateien und ebenso das erstellte Unterverzeichnis wieder entfernt.

Um das zu realisieren, müssen Sie neben der Initialisierungs- und Deinitiali-

Listing 1: ts.c (Fortsetzung von S. 68)

```
20   if (procdirentry==NULL) {
21     pr_err("proc_create_single_data() failed\n");
22     return -EIO;
23   }
24   // proc_set_user(procdirentry, KUIDT_INIT(1000), KGIDT_
   INIT(1000));
25   return 0;
26 }
27
28 static void __exit ts_exit(void) {
29   pr_debug("ts_exit");
30   remove_proc_entry("task_struct", NULL);
31 }
32
33 module_init(ts_init);
34 module_exit(ts_exit);
35 MODULE_LICENSE("GPL");
```

Listing 2: Makefile

```
obj-m += ts.o
KDIR = /lib/modules/$(shell uname -r)/build
all:
        make -C $(KDIR) M=$(shell pwd) modules
clean:
        make -C $(KDIR) M=$(shell pwd) clean
```

```
quade@ezs-cocka:~/lm$ ls
Makefile  ts.c
quade@ezs-cocka:~/lm$ make
make -C /lib/modules/5.15.0-88-generic/build  M=/home/quade/lm modules
make[1]: Verzeichnis „/usr/src/linux-headers-5.15.0-88-generic“ wird betreten
  CC [M]  /home/quade/lm/ts.o
  MODPOST /home/quade/lm/Module.symvers
  CC [M]  /home/quade/lm/ts.mod.o
  LD [M]  /home/quade/lm/ts.ko
  BTF [M] /home/quade/lm/ts.ko
Skipping BTF generation for /home/quade/lm/ts.ko due to unavailability of vmlinux
make[1]: Verzeichnis „/usr/src/linux-headers-5.15.0-88-generic“ wird verlassen
quade@ezs-cocka:~/lm$ sudo insmod ts.ko
quade@ezs-cocka:~/lm$ ls -l /proc/task_struct
-r--r--r-- 1 root root 0 Nov  2 20:03 /proc/task_struct
quade@ezs-cocka:~/lm$ cat /proc/task_struct
       comm: "cat"
        pid: 81202
      stime: 0
      utime: 4000000
static_prio: 120
rt_priority: 0
quade@ezs-cocka:~/lm$
```

4 **Das Generieren, Laden und Einsetzen des Moduls.**

Listing 3: Vorbereitungen

```
$ sudo apt update
$ sudo apt dist-upgrade
$ sudo apt install build-essential
```

sierungsroutine des Moduls (`procfiles_init()` und `procfiles_exit()`) die Methoden `procfiles_seq_open()`, `proc_show()` und nicht zuletzt `proc_write()` implementieren. Die Methode `proc_show()` (Zeile 9) ist prinzipiell identisch mit der bereits vorgestellten Funktion `ts_show()`. Neben Informationen aus dem Task-Control-Block liefert sie Informationen zum Sequence File, insbesondere die Größe und die Anzahl der gespeicherten Bytes. Der Ausgabe lässt sich

Listing 4: procfiles.c

```
01 #include <linux/module.h>
02 #include <linux/proc_fs.h>
03 #include <linux/seq_file.h>
04
05 static struct proc_dir_entry *parent;
06 static struct proc_dir_entry *procdirentry;
07 static ssize_t proc_write(struct file *filp,
   const char *ubuf, size_t len, loff_t *off);
08
09 static int proc_show(struct seq_file *m, void *v)
   {
10   char buffer[128];
11   seq_printf(m, " comm: \"%s\"\n", __get_task_
   comm(buffer, sizeof(buffer), current));
12   seq_printf(m, "  pid: %d\n", current->pid);
13   seq_printf(m, "    m: %px\n", m);
14   seq_printf(m, " size: %lu\n", m->size);
15   seq_printf(m, "count: %lu\n", m->count);
16   return 0;
17 }
18
19 static int procfiles_seq_open(struct inode
   *devfile, struct file *instance) {
20   return single_open(instance, proc_show, NULL);
21 }
22
23 static struct proc_ops proc_fops = {
24   .proc_open    = procfiles_seq_open,
25   .proc_read    = seq_read,       // predefined
26   .proc_write   = proc_write,
27   .proc_lseek   = seq_lseek,      // predefined
28   .proc_release = single_release, // predefined
29 };
30
31 static ssize_t proc_write(struct file *filp,
   const char *ubuf, size_t len, loff_t *off) {
32   char buffer[128];
33   int ret, to_copy, not_copied;
34   long number;
35   memset(buffer, 0, sizeof(buffer));
36   to_copy = min(len, sizeof(buffer));
37   not_copied = copy_from_user(buffer,ubuf,to_
   copy);
38   ret = kstrtol(buffer, 0, &number);
39   pr_info("profile_write: %s - number: %ld\n",
   buffer, number);
40   if (number <= 0) {
41     pr_err("number not valid");
42     return -EFAULT;
43   }
44   snprintf(buffer, sizeof(buffer), "%ld",
   number);
45   procdirentry=proc_create_data(buffer, 0444 ,
   parent, &proc_fops, NULL);
46   *off += to_copy - not_copied;
47   return to_copy-not_copied;
48 }
49
50 static int __init procfiles_init(void) {
51   parent = proc_mkdir("linux-magazin", NULL);
52   if (parent==NULL) {
53     pr_err("proc_mkdir failed");
54     return -EFAULT;
55   }
56   procdirentry=proc_create_data("create", 0666,
   parent, &proc_fops, NULL);
57   if (procdirentry==NULL) {
58     proc_remove(parent);
59     pr_err("proc_create_data failed");
60     return -EFAULT;
61   }
62   return 0;
63 }
64
65 static void __exit procfiles_exit(void) {
66   proc_remove(parent);
67 }
68
69 module_init(procfiles_init);
70 module_exit(procfiles_exit);
71 MODULE_LICENSE("GPL");
```

entnehmen, dass man in einem Singlefile maximal 4 KByte Daten ablegen kann.

Bei der Methode `procfiles_seq_open()` (Zeile 19) handelt es sich um einen Einzeiler, der die Funktion `single_open()` aufruft. Die Modulinitialisierung (`procfiles_init()`, ab Zeile 50) legt in unserem Beispiel ein Unterverzeichnis `/proc/linux_magazin/` an und erzeugt darin unter dem Namen `create` die erste Proc-Datei. Beim Aufruf von `proc_create_data()` übergeben Sie die Liste der Methoden (`proc_fops` vom Typ `struct proc_ops` ab Zeile 23). Dabei fällt auf, dass die Liste nicht nur die Open- und die Write-Funktionsadressen umfasst, sondern noch die Adressen der Funktionen `seq_read()`, `seq_lseek()` und `single_release`. Die sind im Sequence-File-Subsystem des Kernels definiert.

Eine Wahl lassen

Der Eintrag für `seq_lseek()` deutet schon an, dass das Singlefile auch mit dem wahlfreien Zugriff zurechtkommt. Statt Byte für Byte zu lesen, springen Applikationen mithilfe des Systemaufrufs `lseek()` beliebige Positionen innerhalb der Datei an. Sie starten in der Mitte, lesen dann am Anfang und genehmigen sich schließlich die letzten Bytes der Proc-Datei.

Die für uns jetzt interessante Funktion `proc_write()` ab Zeile 31 definiert einen Puffer, der später die Daten aus der Applikation aufnimmt. Darüber hinaus werden einige Variablen definiert, um in jedem Fall die korrekte Menge an Bytes zu kopieren – nicht zu wenig und nicht zu viel. Das stellt das Makro `min()` sicher.

Nach dem Aufruf von `copy_from_user()` stehen die Daten im Puffer zur Verfügung. Per `kstrtoul()` konvertieren Sie den String in eine Nummer, die Sie in der Variablen `number` ablegen. Anschließend konvertieren Sie diese wieder zurück in einen String. Diese zunächst unsinnig wirkende Aktion stellt sicher, dass am Ende ein verwertbarer String vorliegt und kein Binärmüll die Sicherheit gefährdet. Ein negativer Wert oder Null werden nicht akzeptiert.

Buchmacher

In Zeile 45 legen Sie durch Aufruf von `proc_create_data()` die neue, zusätzliche Proc-Datei mit dem numerischen Namen an. Dann gilt es noch, die Anzahl der geschriebenen Bytes im Offset zu verbuchen (Zeile 46) und sie auch als Wert zurückzugeben.

Abbildung 5 zeigt das angepasste Makefile (hier steht in Zeile 1 `procfiles.o` statt `ts.o`), die Generierung des Moduls, das Laden und schließlich den schreibenden und lesenden Zugriff auf die Proc-Dateien. Der Schreibzugriff auf die Datei `create` erzeugt die beiden Proc-Dateien 42 und 99. Beim Entladen des Moduls entfernt der Kernel die Proc-Dateien und das Proc-Verzeichnis wieder.

Ausblick

Das Studium der Funktionsprototypen und der Struktur `struct proc_ops` in der Kernel-Header-Datei `proc_fs.h` offenbart, dass das Proc-Filesystem noch erheblich mehr Funktionalität zu bieten hat. Vor allem erwähnenswert ist dabei die Funktion `proc_set_size()`, die es ermöglicht, einer Proc-Datei eine Größe zuzuordnen. Im Dateimanager beziehungsweise beim Aufruf von `ls -l` tauchen virtuelle Dateien ja zumeist mit einer Größe von 0 Bytes auf. Auch die Ownership lässt sich leicht anpassen: Dazu genügt ein Aufruf von `proc_set_user()` mit den entsprechenden Parametern. Und falls die Proc-Datei generell frische Daten liefern soll, greift man auf die hier nur am Rand gestreiften Sequence Files zurück. Aber das ist ein Thema für eine andere Folge der Kern-Technik. (*jlu*)

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/48908

Weitere Infos und interessante Links
www.lm-online.de/qr/48908

```
quade@ezs-cocka:~/lm$ cat Makefile
obj-m += procfiles.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
	make -C $(KDIR)  M=$(shell pwd) modules

clean:
	make -C $(KDIR)  M=$(shell pwd) clean
quade@ezs-cocka:~/lm$ make
make -C /lib/modules/5.15.0-88-generic/build  M=/home/quade/lm modules
make[1]: Verzeichnis „/usr/src/linux-headers-5.15.0-88-generic“ wird betreten
  CC [M]  /home/quade/lm/procfiles.o
  MODPOST /home/quade/lm/Module.symvers
  CC [M]  /home/quade/lm/procfiles.mod.o
  LD [M]  /home/quade/lm/procfiles.ko
  BTF [M] /home/quade/lm/procfiles.ko
Skipping BTF generation for /home/quade/lm/procfiles.ko due to unavailability
of vmlinux
make[1]: Verzeichnis „/usr/src/linux-headers-5.15.0-88-generic“ wird verlassen
quade@ezs-cocka:~/lm$ sudo insmod procfiles.ko
quade@ezs-cocka:~/lm$ ls -lrt /proc/linux-magazin/
insgesamt 0
-rw-rw-rw- 1 root root 0 Nov  2 20:12 create
quade@ezs-cocka:~/lm$ echo "42" >/proc/linux-magazin/create
quade@ezs-cocka:~/lm$ echo "99" >/proc/linux-magazin/create
quade@ezs-cocka:~/lm$ ls -lrt /proc/linux-magazin/
insgesamt 0
-rw-rw-rw- 1 root root 0 Nov  2 20:12 create
-r--r--r-- 1 root root 0 Nov  2 20:12 99
-r--r--r-- 1 root root 0 Nov  2 20:12 42
quade@ezs-cocka:~/lm$ cat /proc/linux-magazin/99
 comm: "cat"
  pid: 82773
    m: ffff9d5043e78e88
 size: 4096
count: 62
quade@ezs-cocka:~/lm$ cat /proc/linux-magazin/42
 comm: "cat"
  pid: 82774
    m: ffff9d50411d8d98
 size: 4096
count: 62
quade@ezs-cocka:~/lm$
```

5 **Proc-Files bieten eine intuitiv bedienbare Konfigurationsschnittstelle.**