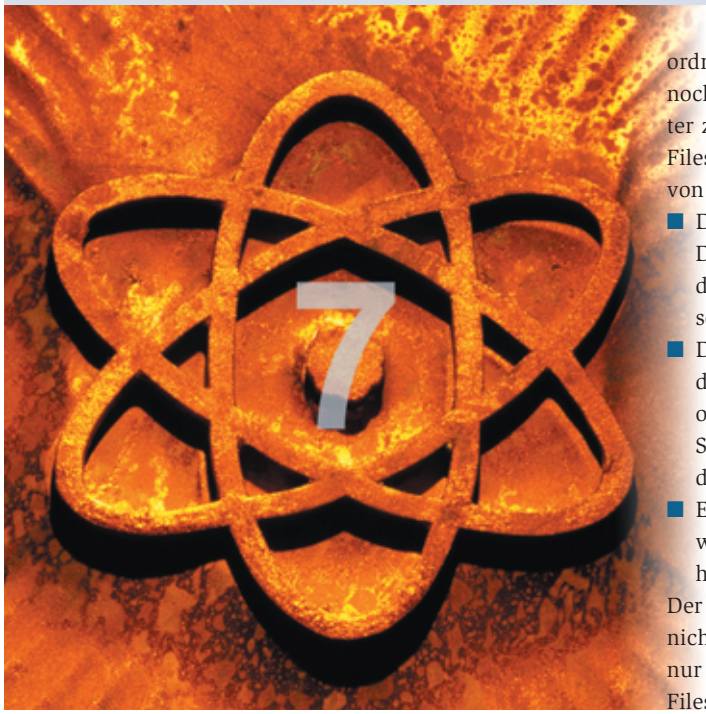
Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 7

Kern-Technik

Laufzeit-Informationen des Kernels lassen sich über das Proc-Filesystem auslesen und verändern. Die siebente Kern-Technik-Folge zeigt, wie man in seinem eigenen Code dieses virtuelle Dateisystem unterstützt. Im Kernel 2.6 vereinfachen Sequence Files die Programmierung. Eva-Katharina Kunst, Jürgen Quade



Das Proc-Filesystem (ProcFS) ist ungewöhnlich, denn es existiert nur virtuell. Es besteht aus Verzeichnissen und Dateien, die sich nicht auf einer Festplatte befinden. Der Kernel erzeugt deren Inhalte stattdessen dynamisch zur Laufzeit. Der Vorteil dabei: Statt alter Kamellen liefert es neueste Informationen.

Simple Schnittstelle

Das Proc-Filesystem ist nicht nur eines der ersten, sondern auch das bekannteste virtuelle Linux-Dateisystem. Anfänglich dafür gedacht, Informationen zu Rechenprozessen zu liefern, ist es mittlerweile zu einem regelrechten Sammelsurium von Verzeichnissen und Dateien geworden. Das in der letzten Kern-Technik-Folge vorgestellte Sys-Filesystem soll zwar in Zukunft mit dieser Unordnung aufräumen, aber noch ist das ProcFS das Fenster zum Kernel. Das virtuelle Filesystem weist eine Reihe von Vorteilen auf:

- Der Benutzer kann über Dateien die Informationen des Kernels lesen und schreiben.
- Dank Ascii-Ausgabe ist diese Schnittstelle auch ohne spezielle Software in Shellskripten vom Anwender nutzbar.
- Es ist für die Kernelentwickler recht einfach zu handhaben.

Der letzte Aspekt gilt aber nicht uneingeschränkt. Es ist nur dann einfach, das Proc-Filesystems zu verwenden, wenn Userspace und Kernel geringe Datenmengen austauschen. Mit steigender Informationsmenge sowie bei sich ändernden Daten wird es schwieriger, das System in eigenen Kernelmodulen zu verwenden. Muss eine Kernelkomponente nur wenige Daten ausgeben, legt der Entwickler ein virtuelles Verzeichnis einfach dadurch an, dass er eine einzige Funktion aufruft:

```
struct proc_dir_entry *mk_procdir( const char
   *name, struct proc_dir_entry *parent )
```

Analog erzeugt folgende Funktion eine Proc-Datei:

```
struct proc_dir_entry *create_proc_entry(
   char *name, mode_t mode, struct proc_dir_
   entry *parent )
```

Der Parameter »name« stellt den Namen des Verzeichnisses respektive der Datei dar. »parent« kennzeichnet das übergeordnete Verzeichnis, unter dem Verzeichnis oder Datei erscheinen. Ein Null-Pointer an dieser Position steht für das »/proc«-Verzeichnis selbst. Die beiden Funktionen geben einen Zeiger auf eine Struktur »struct proc_dir_entry« zurück, die den neuen Eintrag repräsentiert (siehe Listing 1, Zeilen 24 und 25).

Rechte setzen

Mit dem zusätzlichen Parameter »mode« legt man die Zugriffsrechte für die Proc-Datei fest. Die Bit-Kodierung der Rechte befindet sich in der Headerdatei »linux/stat.h«. Wird die Funktion mit dem Wert »0« oder dem Symbol »S_IRUGO« als Parameter aufgerufen, können alle User die Proc-Datei lesen. Eine wesentliche Komponente fehlt bislang – eine Funktion, die die eigentliche Information aus dem Kernel aufbereitet und dem User übergibt. Da sie vom jeweiligen Einsatzzweck abhängt, muss der Kernelprogrammierer sie selbst schreiben (im Folgenden »ProcRead()«).
Um dem Kernel die Adresse dieser Funktion bekannt zu geben, trägt der Programmierer sie in die Struktur »struct proc_dir_entry« ein, die der Kernel beim Aufruf von »create_proc_entry()« zurückgegeben hat. Diesen Vorgang zeigt Zeile 27 in Listing 1. Wer schreibfaul ist, kann statt »create_proc_entry()« auch die Inline-Funktion »create_proc_read_entry()« aufrufen:

```
create_proc_read_entry( "ProcExample",
   S_IRUGO, ProcDir, ProcRead, NULL );
```

Die Funktion schreibt die Adresse von »ProcRead()« direkt in die entsprechende Datenstruktur und ersetzt damit die Zeilen 26 bis 29 von Listing 1.

Ähnlich wie »DriverRead()« (siehe [1]) ruft der Kernel »ProcRead()« auf, sobald eine Applikation über den Systemaufruf »read()« die zugehörige Proc-Datei liest. Um der Applikation die internen Informationen lesbar – also in Ascii kodiert – zur Verfügung zu stellen, sind die in Abbildung 1 dargestellten Schritte erforderlich: Speicher reservieren, Daten aufbereiten, Daten kopieren und den Speicher wieder freigeben, sobald die Applikation bedient ist.

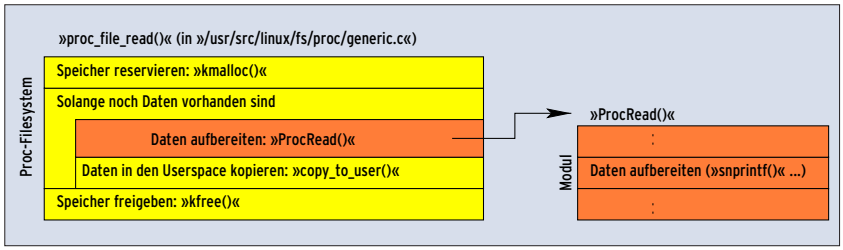


Abbildung 1: Um ein Proc-Filesystem einzurichten, muss der Kernelprogrammierer nur eine »ProcRead()«-Funktion schreiben. Diese wird vom Kernel aufgerufen und schreibt die angefragten Daten in einen Puffer.

Der Kernel hilft

Das Reservieren des Speichers sowie die Kopieraktion zum Userspace und das spätere Freigeben des Speichers übernimmt das Proc-Subsystem automatisch. Dem Programmierer bleibt einzig die Aufgabe, die Daten in dem allozierten Speicher aufzubereiten. Aus diesem Grund besitzen die Funktionen »DriverRead()« und »ProcRead()« – obgleich funktional sehr verwandt – unterschiedliche Aufrufparameter. Ein Funktionskopf sieht folgendermaßen aus:

```
static int ProcRead( char *mempage,
   char **start, off_t offset,
   int size, int *eof, void *data )
```

Der Parameter »mempage« bezeichnet die Adresse der Speicherseite, die das Proc-Subsystem bereits reserviert hat, während »size« die Größe dieses Speicherbereichs nennt. Die Variable »offset« gibt an, ab welcher Stelle in der virtuellen Datei der Userprozess lesen will (Abbildung 5). Die beiden übrigen Parameter »start« und »eof« sind Zeiger auf Variablen, die Rückgabewerte für das Proc-Subsystem aufnehmen.

Über »eof« teilt »ProcRead()« mit, dass keine weiteren Daten zur Ausgabe vorhanden sind. »start« nimmt üblicherweise die Adresse innerhalb der übergebenen Speicherseite auf, ab der »Proc Read()« die Daten abgelegt hat. Falls die auszugebenden Daten am Beginn der Speicherseite stehen, trägt »ProcRead()« hier »NULL« ein. Dann wertet der Kernel den Parameter »offset« aus und findet selbst die Daten.

Eine »ProcRead()«-Funktion ist also leicht zu programmieren, wenn alle Daten in die Speicherseite passen. Die Daten selbst kopiert man am einfachsten mit Hilfe der Funktionen »snprintf()« und »strncat()« in die Speicherseite. Die Zeilen 15 und 16 in Listing 1 zeigen, wie man den Rückgabewert dieser Funktionen nutzt, um sicherzustellen, dass »ProcRead()« nicht mehr Bytes schreibt, als Speicher verfügbar ist. Außerdem gibt die Funktion die Gesamtlänge der geschriebenen Daten zurück.

Vorsicht: Buffer Overflow

Theoretisch stehen auch die Funktionen »sprintf()« und »strcat()« bereit. Man sollte sie aber auch im Kernel vermeiden, denn sie könnten unter Umständen mehr Bytes kopieren, als überhaupt in der Speicherseite Platz haben. Etwas komplizierter wird es, wenn die gut 3000 Bytes einer Speicherseite nicht ausreichen, um alle Daten aufzunehmen. Dann nämlich ruft der Kernel die Funktion »ProcRead()« mehrfach auf. Die wiederum erkennt anhand des Parameters »offset«, welche Daten sie als nächste in der Speicherseite ablegen muss (siehe Abbildung 2).

In diesem Fall kodiert »ProcRead()« einen Datensatz nach Ascii und überprüft, ob dabei der Offset im Ausgabestrom erreicht ist. Wenn nicht, verwirft die Funktion den ersten Datensatz und bereitet den darauf folgenden auf. Diesen Vorgang wiederholt sie so lange, bis der Offset schließlich überschritten ist. Nun muss die Funktion nur noch den Wert von »start« auf den Beginn der aktuellen Ausgabe setzen.

Listing 2 zeigt eine »ProcRead()«-Funktion, die immer wieder den String »Hallo Welt\n« zurückgibt. Wenn etwa 4000 Zeichen ausgegeben sind, setzt »Proc Read()« den Parameter »eof« auf »1« und liefert den Wert »NULL« zurück. Reale

Listing 1: Implementierung einer Proc-Datei

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/proc_fs.h>
04 #include <linux/init.h>
05 #include <linux/stat.h>
06
07 MODULE_LICENSE("GPL");
08
09 static struct proc_dir_entry *ProcDir,
   *ProcFile;
10
11 static int ProcRead( char *buf, char **start,
   off_t offset, int size, int *eof, void *data)
12 {
13    int BytesWritten=0;
14
15    BytesWritten += snprintf( buf, size,
   "ProcRead wurde\n" );
16    BytesWritten += snprintf( buf+BytesWritten,
17         size-BytesWritten, "aufgerufen.\n" );
18    *eof = 1;
19    return BytesWritten;
20 }
21
22 static int __init ProcInit(void)
23 {
24    ProcDir = proc_mkdir( "ExampleDir", NULL );
25    ProcFile = create_proc_entry( "ProcExample",
   S_IRUGO, ProcDir );
26    if ( ProcFile ) {
27         ProcFile->read_proc = ProcRead;
28         ProcFile->data = NULL;
29    }
30    return 0;
31 }
32
33 static void __exit ProcExit(void)
34 {
35    if ( ProcFile ) remove_proc_entry(
   "ProcExample", ProcDir );
36    if ( ProcDir )  remove_proc_entry(
   "ExampleDir", NULL );
37 }
38
39 module_init( ProcInit );
40 module_exit( ProcExit );
```

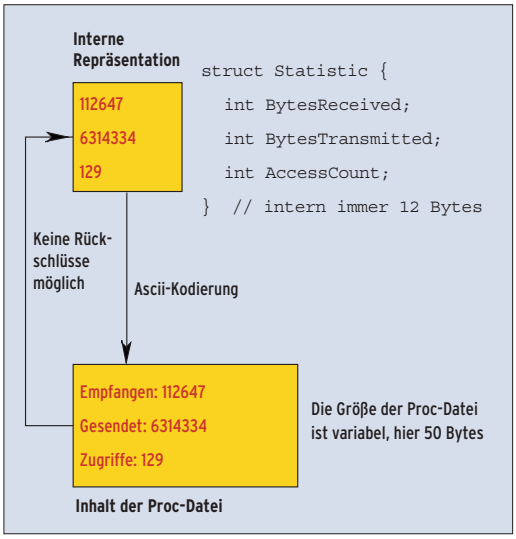


Abbildung 2: Die Größe der Proc-Datei unterscheidet sich vom internen Speicherbedarf der Daten. Auch wenn die Position in den Ausgabedaten (einem Offset entsprechend) bekannt ist, lässt sie keine direkten Rückschlüsse auf den internen Speicherort zu.

»ProcRead()«

»offset« beim Aufruf	Returnwert[1]	»*start« am Ende	neuer Inhalt von »offset«	
0 14	14 23	0 0	14 37	»offset« zählt ausgegebene Ascii-Zeichen
0 1	111 89	1 1	1 2	»offset« zählt in Einheiten, die »ProcRead()« selbst festleg

[1] der Returnwert ist gleich der Anzahl der kopierten Bytes

Abbildung 3: Der Inhalt des Aufrufarguments »start« entscheidet, ob »ProcRead()« den Returnwert oder den Inhalt von »start« selbst zur Berechnung des neuen Offsets verwendet.

Aufgabenstellungen verursachen mehr Kopfzerbrechen. Dass der Ausgabedatenstrom im Regelfall umfangreicher ist als die Repräsentation der internen Daten, stellt an sich noch keine wirkliche Schwierigkeit dar.

Zeitvariante Daten

Dass sich die internen Daten zwischen zwei oder mehr notwendigen Zugriffen auf die »ProcRead()«-Funktion verändern können – also dynamisch sind –, ist problematisch. Soll eine Proc-Datei einen Integerwert ausgeben, der bei einem x86-Kernel grundsätzlich 4 Bytes belegt, benötigt sie für dessen dezimale Ausgabe zwischen 1 (Wert »0«) und 10 Bytes (Wert »4294967296«). Ändert sich der Wert der zugehörigen Variablen zwischen den beiden Aufrufen von »Proc Read()«, kommt es zu einer Verschiebung im Ausgabestrom, also zu einem fehlerhaften Ergebnis. Regel Nummer eins lautet daher, solche Werte nur „ganz oder gar nicht“ auszugeben.

Bei dynamischen Daten – also Daten, die sich mit der Zeit ändern – ist es unmöglich, Rückschlüsse auf den internen Speicherort zu ziehen. Der Parameter »offset« spiegelt nur die bisherige Position im Ausgabedatenstrom wider. Trotzdem muss »ProcRead()« bei wiederholtem Aufruf wissen, wo es beim letzten Mal aufgehört hat zu lesen.

Die Lösung des Problems besteht darin, die aktuelle, interne Position zwischenzuspeichern. Dazu haben die Kernelentwickler auf einen Trick zurückgegriffen. »ProcRead()« darf nämlich in der Variablen »start« auch einen Wert ablegen, der kleiner ist als die Anfangsadresse der Speicherseite. In diesem Fall addiert »ProcRead()« den in »start« abgelegten Wert zum Offset, nicht wie sonst üblich die Anzahl der kopierten Zeichen (siehe Abbildung 3).

In welcher Einheit die interne Position abgelegt wird, bleibt dem Kernelhacker selbst überlassen. Sinnvoll ist es beispielsweise, die Anzahl der bisher ausgegebenen Datensätze mitzuzählen. Wenn also »ProcRead()« beim ersten Aufruf einen Datensatz aufbereitet und diese Daten 10 Bytes belegen, liefert die Funktion zwar – wie bisher auch – »10« zurück. Sie setzt allerdings »start« auf »1«. Beim nachfolgenden Aufruf von »ProcRead()« enthält »offset« nicht den Wert »10«, sondern »1«. »ProcRead()« muss nun den Datensatz mit der Nummer »1« ausgeben, also den zweiten Datensatz. Mit diesem Marker findet die Funktion die Position innerhalb der internen Daten, hier den Index. Listing 3 zeigt, wie man »start« nutzt, um damit die interne Position festzuhalten.

Listing 2: Komplexere Lesefunktion

```
01 static int ProcRead( char *buf, char **start,
   off_t offset,
02                  int size, int *eof, void *data
   )
03 {
04    int ActualWritten=0, BytesWritten=0,
   NewByteCount;
05
06    while( BytesWritten < (long) offset ) { //
   Aktuellen Datensatz suchen.
07       ActualWritten = snprintf( buf, size,
   "Hallo Welt\n" );
08       BytesWritten += ActualWritten;
09    }
10    // In buf befinden sich jetzt alte _und_ neue
   Daten.
11    // Start wird auf die neuen Daten gesetzt.
12    NewByteCount=BytesWritten - offset;
13    *start = buf + ActualWritten -
   NewByteCount;
14    size -= ActualWritten - NewByteCount; // Auf
   neue Startadresse anpassen.
15
16    NewByteCount += snprintf(
   *start+NewByteCount,
17       size - NewByteCount, "Hallo Welt\n");
18    if( offset > 4000 ) {
19       *eof = 1;
20       return 0;
21    }
22    return NewByteCount;
23 }
```

Schon aufgeräumt?

Entfernt der Benutzer das Kernelmodul, sollen die zugehörigen Einträge im Proc-Filesystem natürlich ebenfalls verschwinden. Dazu steht dem Programmierer die Funktion »void remove_proc_entry(const char *name, struct proc_dir_entry *parent)« zur Verfügung.

Eine Proc-Datei kann nicht nur lesen, sondern auch schreiben – ein überaus nützliches Feature. So konfiguriert der Benutzer zur Laufzeit Kernelkomponenten oder Treiber, indem er einfach den entsprechenden Wert in die Proc-Datei schreibt:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Auch für den schreibenden Zugriff steht eine Funktion zur Verfügung, im Folgenden als »ProcWrite()« bezeichnet. Die Adresse dieser Funktion trägt der Kernel-

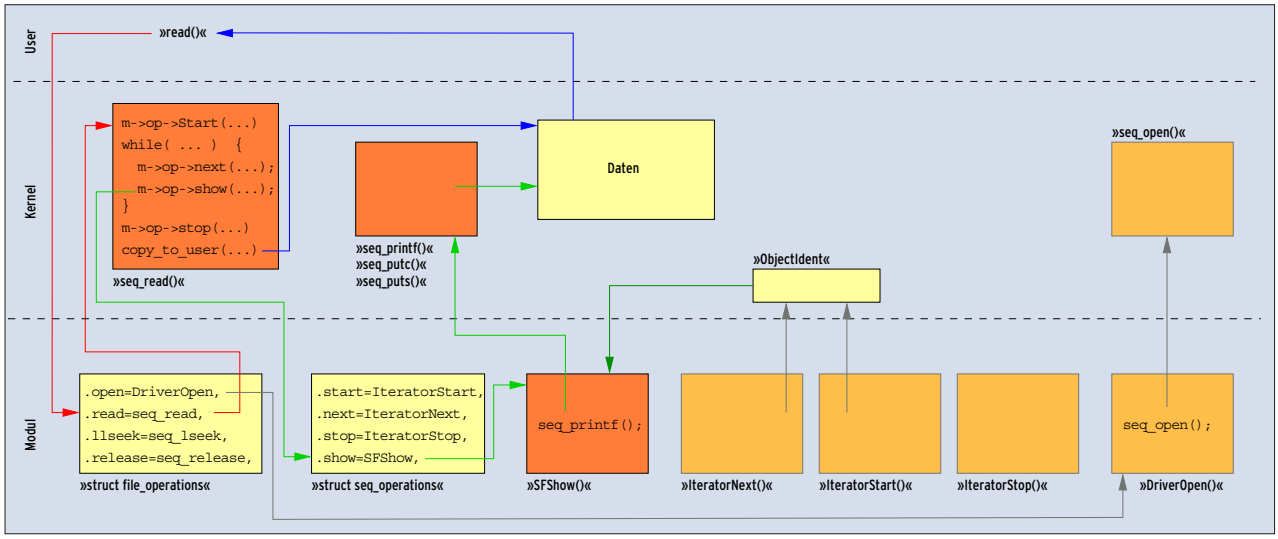


Abbildung 4: Um ein Sequence File zu unterstützen, muss der Programmierer die dargestellten Komponenten realisieren: Drei Iteratoren-Funktionen sowie »SFShow()« benötigt der Kernel, um ein Sequence File anzusprechen. Die Lese- und Schreibfunktionen gleichen dann denen der Standard-C-Bibliothek.

Programmierer in die nach dem Anlegen einer Proc-Datei zurückgegebene Struktur »struct proc_dir_entry« ein. Wichtig ist es dabei, beim Anlegen der Datei die Rechte für den Schreibzugriff (»S_IWUGO«) zu setzen.

Der Prototyp der Funktion (deklariert in »linux/proc_fs.h«) zeigt, dass die Funktion nicht über eine Speicherseite mit der Anwendung kommuniziert, sondern direkt:

```
int ProcWrite( struct file *file, const
   char *buffer, unsigned long count,
   void *data )
```

Einzig der Parameter »data« statt eines Offsets unterscheidet die Funktion von der bekannten »DriverWrite()« (siehe [1]). »ProcWrite()« transferiert die Applikationsdaten selbst aus dem Userspace in den Kernelspace, indem es »copy_from_user()« aufruft. Die meist in Ascii gehaltenen Daten dekodiert man am besten mit Hilfe der Funktionen »strncmp()« oder »strstr()«. Listing 4 zeigt die Implementierung einer Funktion »ProcWrite()«.

Sequence Files

Interne, zeitvariante Informationen Ascii-kodiert ausgeben ist ein Problem aller virtuellen Dateien. Kernel 2.6 bietet mit den so genannten Sequence Files [2] (»linux/seq_file.h«) einen alternativen Lösungsansatz. Ist ein Sequence File (SF) erst mal angelegt, lässt es sich einfach verwenden. Der Entwickler hat im Wesentlichen nur eine Funktion zu schreiben, die einen Datensatz in die Ascii-kodierte Form konvertiert (im Folgenden »SFShow()«):

```
static int SFShow(struct seq_file
   *SFObject, void *ObjectIdent)
```

Der Kernel übergibt dieser Funktion zwei Parameter: erstens die Adresse eines SF-Objekts (»SFObject«), über das der Kernel das Sequence File verwaltet, und zweitens »ObjectIndent«, über den er das auszugebende Element identifiziert. Wie »ObjectIndent« zu interpretieren ist, ist Sache des Programmierers von »SFShow()«. Im Regelfall wird man wohl einen Zeiger auf das auszugebende Element ablegen.

Zur Konvertierung der internen Daten in eine Ascii-kodierte Darstellung bieten sich die folgenden drei Funktionen an: »seq_printf()«, »seq_putc()« und »seq_puts()«. Sie entsprechen den aus dem Applikationsbereich bekannten Funktionen »printf()«, »putc()« und »puts()«. Der einzige Unterschied besteht darin, dass das erste Argument der »seq_«-

Listing 3: Zustandsinformationen in »start«

```
01 static int Beispieldaten[ANZAHL_ELEMENTE];
02 ...
03 static int ProcRead( char *buf, char **start,
   off_t offset,
04         int size, int *eof, void *data )
05 {
06    int count;
07
08    if( offset >= ANZAHL_ELEMENTE ) {
09       *eof = 1;
10       return 0;
11    }
12    count = snprintf( buf, size, "Beispieldaten
   %ld: %d\n",
13       offset, Beispieldaten[offset] );
14    *start = 1; // Ein weiteres Element
   verarbeitet.
15    return count;
16 }
```

Listing 4: Proc-Datei schreiben

```
01 static int ProcWrite( struct file *Instanz,
   const char __user *UserBuffer,
02                 unsigned long count, void
   *data )
03 {
04    char *KernelBuffer;
05    int NotCopied;
06
07    KernelBuffer = kmalloc( count, GFP_KERNEL );
08    if( !KernelBuffer ) {
09       return -ENOMEM;
10    }
11    NotCopied = copy_from_user( KernelBuffer,
   UserBuffer, count );
12    if( strncmp( "Konfiguration", KernelBuffer,
   14 - 1 )==0 )
13       configured = 1;
14    kfree( KernelBuffer );
15    return count - NotCopied;
16 }
```

Funktionen ein Zeiger auf das Sequence-File-Objekt ist. Um die im Speicher liegenden Einzelteile des Sequence File nacheinander zu lesen, benötigt der Kernel außer »SFShow()« weitere geeignete Funktionen, die hier »IteratorStart()«, »IteratorNext()« und »IteratorStop()« heißen. Will eine Anwendung Daten lesen, ruft der Kernel zunächst »IteratorStart()« auf und dann mehrfach »IteratorNext()«. »SFShow()« benutzt die Rückgabewerte dieser Funktionen.

Hat der Kernel genügend Daten gesammelt, ruft er »IteratorStop()« auf und übergibt sie der Anwendung. Sobald diese weitere Daten anfordert, starten zunächst »IteratorStart()«, dann wiederholt »IteratorNext()« und schließlich »IteratorStop()« erneut, und zwar so lange, wie noch Daten vorhanden sind:

```
void *IteratorStart( struct seq_file *m,
   loff_t *Index );
void *IteratorNext( struct seq_file *m,
   void *ObjectIdent, loff_t *Index );
void *IteratorStop( struct seq_file *m,
   void *ObjectIdent );
```

»IteratorStart()« erhält als Argument einen Index, der die Anzahl der bisher bearbeiteten Datensätze angibt. Sind die auszugebenden Daten beispielsweise in einer Liste organisiert, wird »IteratorStart()« genau »Index« Listenelemente verwerfen, um das dann folgende Listenelement der aufrufenden Funktion zurückzugeben. »IteratorNext()« benutzt sowohl den Index als auch die Objekt-Identifikation »ObjectIdent«. Diese Funktion gibt ebenfalls das nächste Element zurück. Sind alle Elemente ausgegeben, ist der Rückgabewert »NULL«. Für jedes zurückgegebenen Element inkrementieren die Funktionen darüber hinaus die Variable »*Index«.

Sobald der Kernel genug Elemente gelesen hat, ruft er die Funktion »IteratorStop()« auf. Sie macht die beim Aufruf von »IteratorStart()« durchgeführten Initialisierungen wieder rückgängig. Mit dem Funktionspaar »IteratorStart()« und »IteratorStop()« lassen sich außerdem konkurrierende Zugriffe auf die auszugebenden Daten über einen Semaphor vermeiden.

Funktionen registrieren

Um die Adressen der Funktionen »IteratorStart()«, »IteratorNext()« und »IteratorStop()« dem Kernel bekannt zu geben, trägt der Entwickler sie in eine Instanz der Datenstruktur »struct seq_operations« ein. Die Funktion »seq_open()« wird typischerweise von »DriverOpen()« oder »ProcOpen()« aufgerufen. Statt der im »struct file_operations« vorkommenden Funktionen »DriverRead()«, »DriverLSeek()« und »DriverRelease()« kommen die durch den Kernel bereitgestellten Funktionen »seq_read()«, »seq_lseek()« und »seq_release()« zum Einsatz. Das Sequence File ist damit fertig implementiert. Abbildung 4 zeigt die Modulkomponenten eines Sequence File sowie den Ablauf, wenn eine Applikation das Sequence File liest.

Das komplette Beispiel eines Sequence File (»seqfile.c«) liegt auf dem FTP-Server des Linux-Magazins bereit [4]. Diese Implementierung organisiert die internen Daten in einer verketteten Liste. Liest eine Anwendung »/proc/SeqTestFile«, gibt der Kernel in diesem Beispiel sämtliche Elternprozesse des aufrufenden Prozesses mit ihrer PID aus.

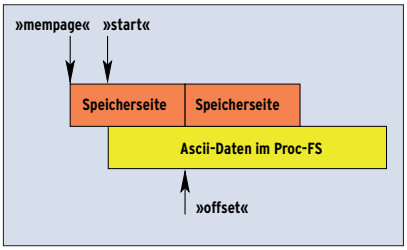


Abbildung 5: »mempage« und »start« verweisen auf die im Speicher liegenden Daten. »offset« bezieht sich dagegen auf die Ausgabe über das Proc-File.

Vorschau

Reale Filesysteme liegen im Gegensatz zu den virtuellen meist auf Festplatten, benutzen also echte Hardware. Der Kernel braucht dafür passende Treiber, um Daten blockweise und im wahlfreien Zugriff zu lesen und zu schreiben. Block-Gerätetreiber implementieren diese Funktionen, sie sind das Thema der nächsten Kern-Technik. *(ofr/fjl)* ■

Single Files – Abwandlung der Sequence Files

Es gibt virtuelle Dateien, denen genau ein Datensatz im Speicher entspricht. Für diesen Fall steht das »Single File«-Interface zur Verfügung. Der Kernel übernimmt das gesamte Handling, insbesondere auch das Seek innerhalb der Daten. Der Entwickler hat letztlich nur eine »Show()«-Funktion zu schreiben. Die »Open()«-Funktion des Moduls (zum Beispiel »DriverOpen()« oder »ProcOpen()«) ruft die Funktion »single_open()« auf. Die Funktion »single_open()« bekommt neben dem Zeiger auf ein »struct file« noch die Adresse der Funktion »SFShow()« übergeben. Da »single_open()« dynamisch Speicher alloziert, kommt statt »DriverRelease()« eine Funktion namens »single_release()« zum Einsatz. Die Lese- und die Seek-Funktionen sind wiederum »seq_read()« und »seq_lseek()«.

Proc-Filesystem aus Benutzersicht

Auch für Anwender bringt der Kernel 2.6 einige Neuerungen mit. Das Format vieler Dateien hat sich geändert und so muss der User die entsprechenden Procps-Programme installieren, um Auskunft über laufende Prozesse oder die Speichernutzung zu bekommen [3]. Wenn, wie schon mehrfach erwähnt, das Sys- viele Aufgaben des Proc-Filesystems übernimmt, ändern sich weitere Files, was für Inkompatibilitäten sorgt. So müssen viele Systemtools angepasst werden, die Kernelinformationen über das Proc-FS lesen und schreiben.

Threads unterstützt das neue ProcFS effizienter, es zeigt erst einmal nur den Original-Thread an und spart damit Suchzeit. Der Kernel schreibt nun über »/proc/config.gz« die Compile- und Modulkonfiguration heraus. Die Benutzer von Multiprozessormaschinen stellen die CPU-Affinität für einzelne Interrupts ein, indem sie eine Bitmaske in »/proc/irq/IRQ#/smp_affinity« schreiben (siehe »Documentation/IRQ-affinity.txt«).

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 9/03, S. 86
[2] Jonathan Corbet, „Driver Porting: The seq_file interface“: [http://lwn.net/Articles/22355/]
[3] Procps: [http://procps.sourceforge.net/]
[4] Listings: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/02/Kern-Technik/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.