Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 14

Kern-Technik

Das Betriebssystem stellt seine Systemaufrufe nur Anwendungen zur Verfügung. Doch manchmal will man solche Services im Kernel selbst verwenden. Das geht, erfordert aber einige Klimmzüge. Eva-Katharina Kunst, Jürgen Quade

Für den Kernel selbst sind Systemcalls in der Regel tabu. Trotzdem ist es in einigen Fällen erforderlich, innerhalb des Kernels Dateien zu lesen, Zugriffsrechte zu ändern oder Programme zu starten. Will der Kernel etwa einen Coredump schreiben oder will sein Hotplug-Subsystem ein Modul laden, sind Systemcalls auch für ihn selbst sehr praktisch. Grundsätzlich stehen dazu drei Möglichkeiten zur Verfügung, siehe Abbildungen 1 bis 3.

Anwendungen helfen

Lösungsansatz 1: Ein Rechenprozess wartet darauf, die Aufträge stellvertretend für den Kernel anzustoßen. Das erfordert aber nicht nur eine geeignete Applikation, sondern auch ein virtuelles Gerät, das der Kern bereitstellt. Beim Zugriff auf dieses Device, etwa über »read()«, legt sich die Anwendung nämlich schlafen. Dass sie vor Verwendung des Moduls schon laufen muss, macht die Sache nicht einfacher.

Etwas eleganter und verbreiteter ist der zweite Lösungsansatz. Hier startet der Kernel selbst die Applikation. Dem Programmierer steht dafür die Funktion »int call_usermodehelper(char *path, char **argv, char **envp, int wait)« zur Verfügung. Sie startet die Userspace-Applikation, die sich hinter »path« verbirgt. Die Funktion bekommt die mit »NULL« terminierte Parameterliste »argv« und die ebenfalls mit »NULL« abgeschlossene Liste des Environments »envp« übergeben.

Der vierte Parameter legt fest, ob »call_usermodehelper()« die Userspace-Applikation im Hintergrund startet oder ob der Kernel auf ihr Ende wartet. Lässt sich die Applikation nicht starten, zum Beispiel wegen fehlender Zugriffsrechte, gibt die Funktion denselben Fehlercode zurück, den sonst der Systemcall »execve()« liefert. Startet der Kernel die Applikation im Hintergrund, ist der Rückgabewert »0«. Wartet er, bis die Anwendung fertig ist, liefert »call_usermodehelper()« deren Exit-Status zurück. Der gestartete Prozess läuft übrigens als Kind des »keventd« (siehe dazu [1]) und mit Superuser-Rechten.

Listing 1 zeigt eine Anwendung, die im Auftrag des Kernels die Datei »/etc/motd« nach »/tmp/foo« kopiert. Wer einen Kernel 2.6 benutzt und die zugehörigen Kernelquellen installiert und konfiguriert hat, kann den Quellcode mit dem Makefile von der Website [2] übersetzen. Da dieses Beispielmodul ohnehin im Kontext von »insmod« abläuft, verhindert »return -EIO« (Listing 1, Zeile 17), dass sich das Modul im Kernel fest-

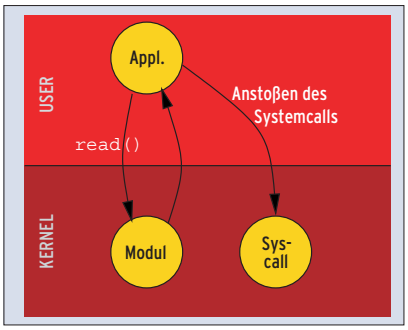

Abbildung 1: Erster Lösungsansatz: Eine laufende Anwendung nimmt Anfragen des Kernels entgegen und ruft für ihn die angefragten Systemcalls auf.

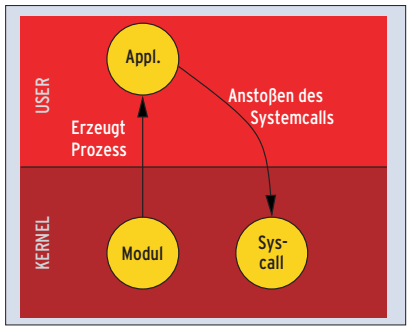

Abbildung 2: Zweiter Lösungsansatz: Die Funktion »call_usermodehelper()« startet eine Applikation, die stellvertretend den Systemcall aufruft.

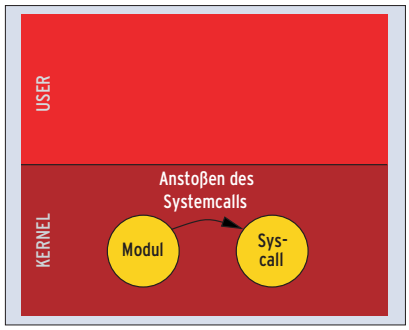

Abbildung 3: Dritter Lösungsansatz: Der Kernel ruft den Systemcall direkt auf. Bei dieser Methode ist keine Hilfe aus dem Userspace nötig.

setzt. Beim Laden des Moduls erscheint dann zwar eine Fehlermeldung, doch diese Methode verkürzt den Testlauf. Immerhin entfällt dadurch das Entladen des Moduls.

Alleingang des Kernels

Es gibt auch Aufgaben, für die das beschriebene Verfahren weniger gut geeignet ist. Sei es aus Performance- oder Design-Gründen: Manchmal soll ein Systemcall direkt vom Kernel, ohne Umweg über eine Applikation, aufgerufen werden. Problematisch ist dabei, dass Systemcalls unter anderem zwei Dinge voraussetzen: Ressourcen im Userspace und einen Prozesskontext.

Kritisch sind aus diesem Grund die Routinen »copy_to_user()« und »copy_from_user()«, die von vielen Systemcalls verwendet werden. Diese Funktionen können nicht nur Daten zwischen User- und Kernelspace, sondern auch innerhalb des Kernelspace austauschen. Zuvor muss allerdings der Sicherheits-Check überwunden werden. Der verhindert nämlich, dass eine Applikation auf Kernelspeicher zugreifen kann.

Mit Hilfe des Makros »get_fs()« lässt sich zunächst jener Ort im Speicher retten, der die Grenze zwischen User- und Kernelspace markiert. Mit »set_fs(KERNEL_DS)« setzt der Programmierer diese Grenze auf einen Maximalwert, sodass der anschließende Test immer positiv ausfällt. Nach dem Aufruf des Systemcalls empfiehlt es sich jedoch, die ursprüngliche Testgrenze wiederherzustellen. Mit diesen beiden Funktionen ergibt sich typischerweise die folgende Codesequenz:

```
mm_segment_t oldfs;
...
oldfs = get_fs();
set_fs(KERNEL_DS);
... // Aufruf des Systemcalls
set_fs(oldfs);
```

Ein Prozesskontext ist nötig, weil Systemcalls häufig die aufrufende Instanz schlafen legen. Wird der Systemcall aber von einer beliebigen Kernelfunktion aufgerufen, legt er die gerade zufällig aktive Anwendung schlafen. Er hat sich praktisch ihre Umgebung geraubt. Daher muss man in den meisten Fällen – wie es auch »call_usermodehelper()« praktiziert – einen eigenen Kernel-Thread erzeugen, siehe [1].

Aufruf des Systemcalls

Um innerhalb des Kernels einen Systemcall zu nutzen, kann nicht – wie in [3] beschrieben – ein Software-Interrupt ausgelöst werden. Stattdessen ruft man die Handler-Funktionen direkt auf. Diese sind meist nach dem Schema »sys_*SysCallName*()« benannt. So heißt die Kernelfunktion zu »open()« »sys_open()«, die zu »read()« »sys_read()«. Sie besitzen im Regelfall dieselben Parameter wie die entsprechenden Funktionen auf Applikationsebene.

Die Rückgabewerte der Handlerfunktionen entsprechen der Konvention für Rückgabewerte im Kernel. Ein Wert zwischen -1 und -1000 zeigt einen Fehler an, alles andere signalisiert Erfolg. Für die Fehlerauswertung verwendet der geübte Kernelhacker das Makro »long IS_ERR (const void *ptr)«.

Wer ein dynamisches Kernelmodul schreibt, kann allerdings nur auf die Handlerfunktionen »sys_open()«, »sys_close()«, »sys_lseek()« und »sys_read()« zurückgreifen. Nur diese Funktionen exportiert der Kernel 2.6.7 bei einer x86-Architektur – das sind nicht gerade viele im Vergleich zu den Anwendungen, de-

Listing 1: Kernel startet Applikation »umh.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 static int __init mod_init(void)
07 {
08   int ret;
09   char *argv[4] = {"/bin/cp", "/etc/motd", "/tmp/
   foo", NULL};
10    char *envp[3] = {"HOME=/", "PATH=/sbin:/bin:/
   usr/sbin:/usr/bin", NULL};
11
12   ret=call_usermodehelper(argv[0],argv,envp,1);
13    if( ret < 0 ) {
14     printk("call_usermodehelper failed ...\n");
15     return -EIO;
16   }
17   return -EIO; // nur zum Test
18 // return 0;
19 }
20
21 static void __exit mod_exit(void)
22 {
23 }
24
25 module_init( mod_init );
26 module_exit( mod_exit );
```

Listing 2: »readfile.c«

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/completion.h>
05 #include <linux/fs.h>
06
07 MODULE_LICENSE("GPL");
08
09 static int ThreadID = 0;
10 static DECLARE_COMPLETION(OnExit);
11 static char buf[128];
12
13 static int ThreadCode(void *data)
        //  Prozessorkontext
14 {
15   struct file *config_file;
16   daemonize("MyKThread");
17   allow_signal( SIGTERM );
18
19   // Dateizugriffe im Kernel
20   config_file = filp_open( "/etc/motd", O_RDONLY, 0 );
21   if( IS_ERR( config_file ) )
22      return -EIO;
23   kernel_read( config_file, 0, buf, sizeof(buf) );
24   printk( buf );
25   filp_close( config_file, NULL );
26
27   // Hier sind Zugriffe auf Systemcall-Handler
        möglich
28   complete_and_exit( &OnExit, 0 );
29 }
30
31 static int __init kthreadInit(void)
32 {
33   ThreadID=kernel_thread( ThreadCode, NULL,
       CLONE_KERNEL );
34   if( ThreadID==0 )
35     return -EIO;
36   return 0;
37 }
38
39 static void __exit kthreadExit(void)
40 {
41   kill_proc( ThreadID, SIGTERM, 1 );
42   wait_for_completion( &OnExit );
43 }
44 module_init( kthreadInit );
45 module_exit( kthreadExit ); // Modul-Handling
```

nen fast 280 Systemcalls zur Verfügung stehen. Aber das bedeutet nicht, dass der Kernelprogrammierer auf die übrigen Syscalls verzichten muss.

Überhaupt lassen sich wenige Systemaufrufe im Kernel sinnvoll einsetzen. Schließlich dienen viele dazu, Informationen aus dem Kernel in den Applikationsbereich zu transportieren, zum Beispiel die UID eines Rechenprozesses. Innerhalb des Betriebssystemkerns lassen sich solche Information ohne Umweg abfragen. Die Handlerfunktionen sind vielfach nichts weiter als Wrapper um die Routinen im Kernel, die die eigentliche Arbeit ausführen.

Dateizugriff aus dem Kernel

Auch beim Zugriff auf Dateien sind die wichtigen Funktionen im Kernel versteckt. Die Handlerfunktion »sys_open()« ruft nämlich die Kernelroutine »struct file *filp_open(const char * filename, int flags, int mode)« auf. Zuvor erzeugt »sys_open()« den Filedeskriptor, den aber normalerweise nur die Applikation benötigt.

Kernelcode greift effizienter direkt über die von »filp_open()« zurückgegebene Instanz vom Typ »struct file« zu, siehe [4]. Die Parameter »flags« und »mode« der Funktion »filp_open()« geben Zugriffsart und -rechte an, wie vom Systemcall »open« bekannt ist. So öffnet die Codezeile

```
log_file = filp_open( "/tmp/foo",
   O_RDWR | O_CREAT, 0644 );
```

die Datei »/tmp/foo« zum Lesen und Schreiben. Existiert die Datei nicht, legt der Kernel sie mit den Zugriffsrechten 644 an. Die Funktion »filp_open()« gibt einen Zeiger auf »struct file *« zurück oder einen Fehlerwert. Die entsprechende Funktion zur Freigabe lautet »int filp_close(struct file *filp, fl_owner_t id)«. Der Parameter »id« vom Typ »fl_owner_t« (»struct files_struct«) ist ein Zeiger auf die zum Prozess gehörigen Dateien, wenn »filp_close()« indirekt über einen Syscall aufgerufen wurde. Im Kernelfall trägt der Programmierer entweder »0« oder »current->files« ein. Beide Funktionen sind in der Headerdatei »linux/fs.h« deklariert.

Statt »sys_read()« zum Lesen zu verwenden (was ohne Filedeskriptor auch nicht geht), bietet sich die Funktion »ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)« an (Abbildung 4). Zuvor muss aber noch wie beschrieben der Speicher-Check ausgehebelt werden. Eine Kernelfunktion macht das selbst und kopiert auch gleich die Daten: »int kernel_read(struct file *file, unsigned long offset, char *addr, unsigned long count)«.

Im Fall des File- oder Gerätezugriffs ließen sich auch direkt die Lese- und Schreibfunktionen benutzen, die in der »struct file_operations« abgelegt sind, siehe [4]. Doch ist das mit mehr Programmieraufwand verbunden. So ist vorher zu prüfen, ob die Zugriffsfunktion überhaupt existiert. Schließlich gibt es kein Gesetz, dass jeder Treiber jede Funktion implementieren muss.

Listing 2 zeigt einen Programmausschnitt, der innerhalb des Kernels die Datei »/etc/motd« einliest und über den Syslog-Daemon wieder ausgibt. Zur Übersetzung des Moduls ist im Makefile

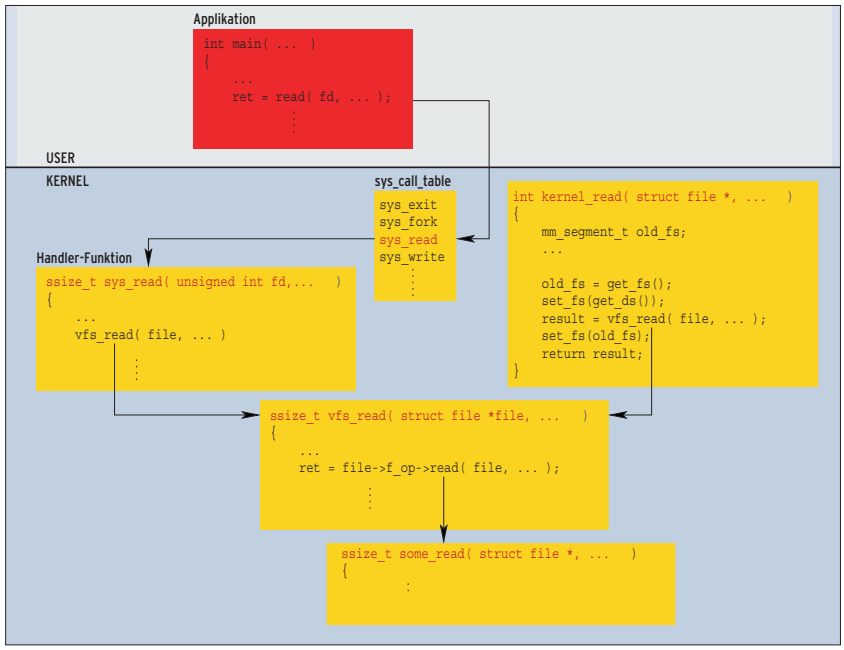

Abbildung 4: Im Kernel sind einige Funktionen beteiligt, wenn aus einer Datei gelesen wird. Die Userspace-Funktion »read()« ruft die Handler-Funktion »sys_read()« auf, diese wiederum »vfs_read()«.

Listing 3: Schreibzugriff aus dem Kernel

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03 #include <linux/fs.h>
04 #include <asm/uaccess.h>
05
06 MODULE_LICENSE("GPL");
07
08 static char filename[255];
09 module_param_string( filename, filename,
   sizeof(filename), 666 );
10 struct file *log_file;
11
12 static int __init mod_init(void)
13 {
14    mm_segment_t oldfs;
15
16    if( filename[0]=='\0' )
17       strncpy( filename, "/tmp/kernel_file",
   sizeof(filename) );
18    printk("opening filename: %s\n", filename);
19    log_file = filp_open( filename,
   O_WRONLY|O_CREAT, 0 );
20    if( IS_ERR( log_file ) )
21       return -EIO;
22
23    oldfs = get_fs();
24    set_fs( KERNEL_DS );
25    vfs_write( log_file, "Hello World\n", 12,
   &log_file->f_pos );
26    set_fs( oldfs );
27    filp_close( log_file, NULL );
28    return 0;
29 }
30
31 static void __exit mod_exit(void)
32 {
33 }
34 module_init( mod_init );
35 module_exit( mod_exit );
```

nur der Modulname durch den Namen des Moduls aus Listing 2 zu ersetzen: Aus »umh.o« wird »readfile.o«. Nach dem Laden des Moduls taucht im Syslog, zum Beispiel in »/var/log/messages«, der Inhalt von »/etc/motd« auf.

Schreiben von Dateien

Eine Datei direkt vom Kernel schreiben funktioniert ähnlich wie das Lesen. Auch hier bietet ein Aufruf von

```
Instanz->f_op->write(log_file,
    "hallo\n", 7,  &offset );
```

direkten Zugriff, falls die Schreibfunktion »write()« implementiert ist. Die Funktion »ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)« nimmt dem Programmierer derartige Überprüfungen ab. Eine Funktion, die analog zu »kernel_read()« die Überprüfung selbst aushebelt, gibt es allerdings nicht. Beim Aufruf der Funktion ist noch darauf zu achten, dass der Parameter »pos« auch wirklich ein Zeiger ist.

Der in Listing 3 vorgestellte Kernelcode zeigt ein einfaches Beispiel, das den eben beschriebenen Weg geht. Es schreibt den String »Hello World« in eine Datei. Zum Übersetzen ist wiederum das Makefile anzupassen. Ohne Angabe des Modulparameters schreibt das Modul seinen String in die Datei »/tmp/kernel_file«. Dieses Beispiel erzeugt keinen eigenen Thread, es borgt sich den Prozesskontext von »insmod«.

Ausblick

Mit einigen Tricks lassen sich also auch im Kernel Systemaufrufe verwenden. Man sollte dabei aber auch an Sicherheitsaspekte denken, denn wo mehr Code im Kernelmodus abläuft, dort sind auch mehr potenziell gefährliche Fehler. Dem Programmierer steht durch die vorgestellten Techniken im Kernel eine Vielzahl von Funktionen zur Verfügung, mit denen er zum Beispiel Dateien lesen und schreiben kann. Wie man im Kernel Netzwerkfunktionen nutzt, zeigt die nächste Folge dieser Reihe. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 4: Linux-Magazin 11/03, S. 96
[2] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/09/Kern-Technik]
[3] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 13: Linux-Magazin 8/04, S. 92
[4] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 2: Linux-Magazin 9/03, S. 86

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, haben unter dem Titel »Linux-Treiber entwickeln« ein Buch zum Kernel 2.6 veröffentlicht.