Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 13

Kern-Technik

Systemaufrufe bilden die Schnittstelle zwischen Userspace und Kernel. Dieser Artikel zeigt, wie sie funktionieren und wie man sogar eigene Systemcalls implementiert. Eva-Katharina Kunst, Jürgen Quade

Das Betriebssystem verwaltet Ressourcen wie Rechenzeit, Speicher oder Geräte und stellt Dienste dafür zur Verfügung: Will eine Anwendung auf Hardware zugreifen, tut sie das nicht direkt, sondern vermittelt durch den Kernel, der im Prozessormodus mit der dafür nötigen Priorität läuft. Für die meisten Zwecke bringt das Betriebssystem passende Interface-Funktionen mit, die so genannten Systemaufrufe, die einen definierten Übergang vom User- in den Kernelmodus darstellen. Mehr als 280 Systemcalls finden sich in Kernel 2.6.6.

Durchnummeriert

Jeder Systemcall besitzt eine eindeutige Nummer, die ihm in der Headerdatei »asm/unistd.h« (siehe Listing 1) zugeordnet wird. Der Systemcall »fork«, der einen neuen Prozess erzeugt, trägt beispielsweise die Nummer 2, »write« entspricht 4 und »gettimeofday« der 78.

Normalerweise ruft die C-Bibliothek Glibc hinter den Kulissen solche Systemcalls auf, wenn eine Anwendung entsprechende Funktionen benutzt. Um einen Systemcall direkt aufzurufen, muss manuell ein Software-Interrupt ausgelöst werden. Da die Programmiersprache C hierzu keine Funktion anbietet, braucht der Entwickler dafür Assembler.

Es gibt mehrere Software-Interrupts. Für Systemcalls kommt traditionell die Nummer 0x80 zum Einsatz, auch wenn es noch andere Wege gibt, um in den Kernel zu wechseln, siehe Kasten „Intel kann es schneller“. Der Interrupt löst den Übergang in den privilegierten Kernelmodus aus. Der Kernel selbst erwartet die Nummer, die den auszuführenden Dienst kennzeichnet (also beispielsweise 4 für »write«), in einem Register, auf einer x86-Plattform im EAX. Die zum Systemcall gehörigen Parameter werden ebenfalls in Registern abgelegt. Abbildung 1 zeigt die Register eines x86-Prozessors, weitere Informationen finden sich unter [1].

Parameter in Registern

Listing 2 ruft in Assembler den Systemcall »write« auf. Der Filedeskriptor, die Adresse für die Ausgabe und die Anzahl der schreibende Bytes liegen nacheinander in den Registern EBX, ECX und EDX. Um aus dem Assembler-Code ein Programm zu erzeugen, reicht der Aufruf von »make hello« aus. Zum selben Ergebnis führt »gcc -o hello hello.S«.

In den seltensten Fällen schreibt der Anwendungsentwickler seinen Assembler-Code selbst. Für die wichtigsten Systemcalls gibt es in der Standard-C-Library entsprechende Funktionen, die er nur aufzurufen braucht. Aber auch wenn der Systemcall nicht über eine Bibliotheksfunktion zur Verfügung steht, ist Assembler-Programmierung kein Muss. Vielmehr stellt der Kernel einen Satz Makros bereit und die Standardbibliothek hat eine Systemcall-Funktion.

Die »_syscall*X*()«-Makros versehen einen Systemcall mit einem Funktionsnamen, den der Programmierer später verwenden kann. Das »*X*« steht dabei für die Anzahl der Parameter. Um beispielsweise den Systemcall »write« mit seinen drei Parametern zu nutzen, kommt das Makro »_syscall3()« zum Einsatz. Listing 4 zeigt den Aufruf.

Da die »_syscall*X*()«-Makros intern auf die globale Variable »errno« zugreifen, muss der entsprechende Header eingebunden sein (Zeile 2). Außerdem ist sicherzustellen, dass der Compiler die

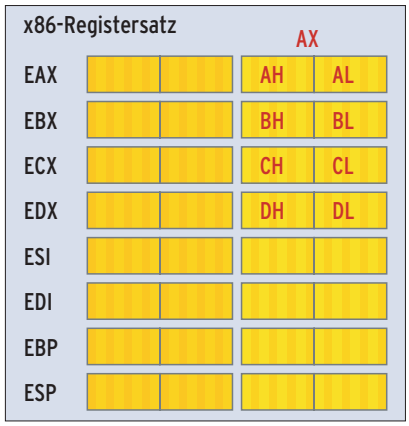

Abbildung 1: Die Parameterübergabe zwischen User- und Kernelspace läuft über die Register des x86-Prozessors: Im EAX steht die Nummer des Syscall, die anderen nehmen dessen Parameter auf.

Headerdateien des Kernels 2.6 verwendet, zum Beispiel durch den Aufruf:

```
cc -Wall -I/usr/src/linux-2.6.6/include -o ➥
syscallX syscallX.c
```

Der erste Parameter des »_syscall*X*()«-Makros steht für den Typ des Systemcall-Rückgabewerts. Der zweite Parameter entspricht dem Namen des Systemcall, der als solcher in der Datei »asm/unistd .h« festgelegt ist. Der dortige Namensvorsatz »__NR_« ist allerdings wegzulassen; aus »__NR_write« wird damit »write«. Danach folgt – immer paarweise – die Beschreibung der einzelnen Parameter: zuerst der Typ, dann der Name. Zum Einsatz der Makros ist die Datei »asm/unistd.h« einzubinden.

Funktion statt Makro

Linus Torvalds & Co. sind allerdings nicht sonderlich glücklich über die Verwendung der Makros. Tatsächlich denken die Kernelentwickler darüber nach, sie den Anwendungsprogrammierern in Zukunft vorzuenthalten. Stattdessen empfehlen sie, lieber die Funktion »syscall()« zu verwenden, die in »unistd.h« deklariert ist. Wie man sie benutzt, zeigt Listing 5.

Die »syscall«-Funktion und damit auch das Makro »_syscallX« verarbeiten den Rückgabewert des Systemcall weiter. Bewegt sich der Wert zwischen -1 und -125, belegt die Funktion damit die »errno«-Variable und gibt »-1« zurück. Im Kernel selbst gibt es für jeden Systemcall eine System Call Service Routine (Systemcall-Funktion), die den Auftrag weiterbearbeitet. Wie der Kernel die Systemcall-Parameter verarbeitet und die Serviceroutine aufruft, ist größtenteils in Assembler programmmiert und plattformabhängig. Die Adressen der Systemcall-Funktionen enthält die Funktionstabelle »syscall_table« gemäß den Systemcall-Nummern. Die entsprechende Tabelle ist für die x86-Architektur in der Datei »arch/i386/kernel/entry.S« im Kernel-Quellcode definiert.

Implementierung im Kernel

Die Bearbeitungsfunktion selbst ist plattformunabhängig und im Kernel als einfache Routine implementierbar. Vor der Funktionsdefinition muss nur das Schlüsselwort »asmlinkage« stehen. Es legt abhängig von der Plattform eine passende Aufrufkonvention fest.

Die Namen der Systemcall-Funktionen beginnen in der Regel mit dem Vorsatz »sys_«. Somit lautet die Funktion, die »write« implementiert, »sys_write()«. Systemcalls liefern grundsätzlich ein Ergebnis vom Typ »long«. Die Konvention dabei ist: 0 oder ein positiver Wert besagen, dass kein Fehler aufgetreten ist. Ein negativer Wert dagegen zeigt den Misserfolg der Operation an. Einige Systemcalls implementiert die Datei »kernel/sys.c«, »sys_write()« findet sich unter »fs/read_write.c«.

Im Kernel laufen die Systemcalls im Kontext des zugehörigen Prozesses ab. Das bedeutet, dass der Systemcall den Rechenprozess schlafen legen kann und dass Daten zwischen User- und Kernelspace über die Funktionen »copy_from

Intel kann es schneller

Einen Software-Interrupt über den Befehl »int« aufrufen ist auf einem x86-Prozessor vergleichsweise Ressourcen-intensiv. Bereits seit dem Pentium II hat Intel in den Befehlssatz seiner Prozessoren daher das neue Befehlspaar »sysenter« und »sysexit« aufgenommen. Mit ihm führt der Kernel Systemcalls erheblich schneller aus.

Allerdings ist ein Wechsel vom bisherigen »int«-Mechanismus zum »sysenter«-Mechanismus nicht ohne weiteres möglich. Da es sich um die Schnittstelle zwischen User- und Kernelspace handelt, müssen sowohl die Anwendungen als auch der Kernel angepasst werden. Die größte Schwierigkeit ist aber, dass es »sysenter« nicht bei allen x86-kompatiblen Prozessoren gibt. Deshalb führt Kernel 2.6 die so genannte »vsyscall«-Page ein. In den Adressraum einer Applikation wird eine Speicherseite eingeblendet, in der abhängig vom Prozessor entweder der Code für den Software-Interrupt oder für »sysenter« zu finden ist.

Den bisherigen Aufruf »int $0x80« ersetzt der Programmierer durch einen Unterprogrammaufruf an die Adresse »0xfffe400«. Der Beispielaufruf aus Listing 2 ist mit den entsprechenden Änderungen in Listing 3 dargestellt. Wird der Code mit diesem »make hello2« übersetzt und aufgerufen, muss ebenfalls „Hello World“ auf dem Bildschirm erscheinen.

Listing 1: Auszug aus »asm/unistd.h«

```
01 #ifndef _ASM_I386_UNISTD_H_
02 #define _ASM_I386_UNISTD_H_
03
04 /*
05  * This file contains the system call numbers.
06  */
07
08 #define __NR_restart_syscall      0
09 #define __NR_exit                 1
10 #define __NR_fork                 2
11 #define __NR_read                 3
12 #define __NR_write                4
13 #define __NR_open                 5
14 #define __NR_close                6
15 ...
```

Listing 2: »hello.S«

```
01 .text
02 .globl main
03 main:
04   movl $4,%eax        ; //Code fuer "write" systemcall
05   movl $1,%ebx        ; //File descriptor fd (1=stdout)
06   movl $message,%ecx  ; //Adresse des Textes (buffer)
07   movl $12,%edx       ; //Laenge des auszugebenden Textes
08   int $0x80           ; //SW-Interrupt, Auftrag an das BS
09   ret
10 .data
11 message:
12   .ascii "Hello World\n"
```

Listing 3: »hello2.S«

```
01 .text
02 .globl main
03 main:
04   movl $4,%eax        ; //Code fuer "write" systemcall
05   movl $1,%ebx        ; //File descriptor fd (1=stdout)
06   movl $message,%ecx  ; //Adresse des Textes (buffer)
07   movl $12,%edx       ; //Laenge des auszugebenden Textes
08   call 0xffffe400     ; //Auftrag an das BS
09   ret
10 .data
11 message:
12   .ascii "Hello World\n"
```

Listing 4: Syscall mit »_syscallX«-Makro

```
01 #include <asm/unistd.h>
02 #include <errno.h>
03
04 _syscall3( int, write, int, fd, char *, buffer, int,
   size );
05
06 int main( int argc, char **argv )
07 {
08         write( 1, "hello world\n", 13 );
09         return 0;
10 }
```

Listing 5: Systemcall über »syscall()«

```
01 #include <unistd.h>
02 #include <asm/unistd.h>
03
04 int main( int argc, char **argv )
05 {
06         syscall( __NR_write, 1, "Hello World\n", 13 );
07         return 0;
08 }
```

Listing 6: »sys_udelay()«

```
01 #include <linux/syscalls.h>
02 #include <linux/jiffies.h>
03 #include <asm/delay.h>
04
05 asmlinkage long sys_udelay( int museconds )
06 {
07         printk("sys_udelay( %d )\n", museconds );
08         if( (museconds < 0) || (museconds > 10000) )
09                 return -EINVAL;
10         udelay( museconds );
11         return 0;
12 }
```

Listing 7: Änderung in »kernel/Makefile«

```
01 #
02 # Makefile for the linux kernel.
03 #
04
05 obj-y     = sched.o fork.o exec_domain.o panic.o printk.o
   profile.o \
06             exit.o itimer.o time.o softirq.o resource.o \
07             sysctl.o capability.o ptrace.o timer.o user.o \
08             signal.o sys.o kmod.o workqueue.o pid.o \
09             rcupdate.o intermodule.o extable.o params.o
   posix-timers.o \
10             kthread.o sysudelay.o
11
12 obj-$(CONFIG_FUTEX) += futex.o
13 ...
```

Listing 8: »include/linux/syscalls.h«

```
01 ...
02 asmlinkage long sys_uselib(const char __user
   *library);
03 asmlinkage long sys_ni_syscall(void);
04 asmlinkage long sys_ni_syscall(void);
05
06 asmlinkage long sys_udelay(int musec);
07
08 #endif
```

Listing 9: »arch/i386/kernel/entry.S«

```
01          ...
02          .long sys_mq_timedsend
03          .long sys_mq_timedreceive       /* 280 */
04          .long sys_mq_notify
05          .long sys_mq_getsetattr
06          .long sys_udelay
07
08 syscall_table_size=(.-sys_call_table)
```

_user()« und »copy_to_user()« (siehe [2]) ausgetauscht werden.

Wer Erfahrung in der Programmierung mit C, mit dem Kompilieren und Installieren eines Linux-Kernels hat (siehe [3]), kann einen Systemcall in nur wenigen Schritten selbst programmieren. Als Kernelmodul ist das allerdings nicht möglich – zumindest nicht ohne besondere Hacks. Ein Systemcall wird statisch im Kernel implementiert.

Systemcall selbst gestrickt

Das hier aufgeführte Beispiel realisiert einen Systemcall »udelay«, der in einem Busy Loop zwischen 0 und 10 Millisekunden Rechenzeit beansprucht, die der Benutzer in Mikrosekunden angibt. Als Basis dient ein Kernel 2.6.6, der in dem Verzeichnis »/usr/src/linux-2.6.6« installiert ist. Die im Folgenden aufgeführten Pfad- und Dateinamen beginnen – falls nicht anders angegeben – in diesem Verzeichnis. Der komplette Dateiname von »include/linux/unistd.h« lautet damit »/usr/src/linux-2.6.6/include/linux/uni-std.h«.

Abbildung 2 zeigt die neun Schritte zum eigenen Systemcall: Zuerst legt der Programmierer in den Kernelquellen eine neue Datei an (Schritt 1), zum Beispiel im Verzeichnis »kernel«. Hier heißt sie »sysudelay.c« und besteht im Wesentlichen aus der Systemcall-Serviceroutine »sys_udelay()« (Schritt 2). Der zugehörige Code ist in Listing 6 zu sehen. Abgesehen vom Schlüsselwort »asmlinkage«, dem Rückgabewert vom Typ »long« und der maximalen Parameteranzahl von fünf gibt es hier nichts Besonderes zu beachten. Der Rückgabewert wird später im Userspace (im Makro »_syscall*X*()« respektive in der Funktion »syscall()«) in die globale Fehlervariable »errno« umgesetzt.

Als Headerdatei ist normalerweise nur »linux/syscalls.h« zu inkludieren. In diesem Fall muss zusätzlich »asm/delay.h« eingebunden werden, die das Makro »udelay()« definiert. Im dritten Schritt wird das Makefile angepasst: Die Variable »obj_y« erhält zusätzlich den Namen des neuen Objektfile. Listing 7 zeigt die Änderungen rot markiert. Schritt 4 erweitert dann die Headerdatei »include/linux/syscalls.h« um den Prototyp der neuen Systemcall-Servicefunktion, siehe Listing 8.

Schritt 1:	Im Linux-Quellcode eine neue Datei anlegen
Schritt 2:	In der neuen Datei den Systemcall implementieren
Schritt 3:	Makefile anpassen
Schritt 4:	Funktionsprototyp in <include/linux/syscalls.h> eintragen
Schritt 5:	Neuen Systemcall in die »sys_call_table« eintragen
Schritt 6:	Einführung der neuen Systemcall-Nummer in <include/asm/unistd.h>
Schritt 7:	Kernel erzeugen
Schritt 8:	Kernel installieren
Schritt 9:	Test

Abbildung 2: In neun einfachen Schritten lassen sich eigene Systemcalls im Kernel implementieren und benutzen.

Header ändern

Die Adresse des neuen Systemaufrufs »sys_udelay« muss jetzt noch ans Ende der Syscall-Tabelle »sys_call_table« angehängt werden (Schritt 5). Dieser Schritt ist plattformabhängig, wie auch der folgende. Auf der »i386« Architektur befindet sich die »sys_call_table« beispielsweise in der Assembler-Datei »arch/i386/kernel/entry.S« (siehe Listing 9). Für die PowerPC-Plattform heißt die entsprechende Datei »arch/ppc/kernel/misc.S«. Wer einen AMD 64 unterstützen möchte, trägt die Zeile in die Datei »include/asm-x86_64/unistd.h« ein.

Der neue Syscall bekommt in der Datei »unistd.h« eine Nummer zugeordnet – das gilt für alle Plattformen. Der Beispiel-Systemcall erhält gemäß Konvention die Bezeichnung »__NR_udelay«. Ein solcher Eintrag muss für jede unterstützte Plattform bestehen, auf dem PC in der Datei »include/asm-i386/unistd.h«. Der Systemcall bekommt auf dieser Plattform und dem Kernel 2.6.6 die Nummer 283. Das Define »__NR_syscalls«, das die Anzahl der Systemcalls angibt, wird um eins erhöht und erhält somit den Wert 284 (siehe Listing 10).

Jetzt kann der Kernel neu kompiliert (Schritt 7) und installiert (Schritt 8) werden.

Nach einem Reboot steht der neue Systemcall zum Test bereit. Dazu dient das in Listing 11 vorgestellte Programm. Es ruft »sys_udelay()« zweimal auf, davon einmal mit ungültigem Parameter.
Nach dem Übersetzen und Starten des Programms, gibt die Syscall-Routine beim ersten Aufruf von »syscall()« den Wert »0« zurück, beim zweiten Mal »-1« (vergleiche Listing 6). In diesem Fall setzt sie die globale Variable »errno« zusätzlich auf »-EINVAL«. Die Ausgabe des Testprogramms lautet damit:

```
# ./testsys
udelay(9000)=0
udelay: Invalid argument
udelay(10001)=-1
```

Wie geht's weiter?

Nur selten ist es erforderlich, einen eigenen Systemcall zu implementieren. Für die meisten Anwendungen genügen jene Systemaufrufe voll und ganz, die der Kernel von Haus aus mitbringt. Normalerweise benutzen nur Anwendungsprogramme oder Libraries Systemaufrufe. Manchmal möchte allerdings auch der Kernelhacker gerne einen Dienst des Betriebssystem-Kerns verwenden. Wie man das macht, zeigt die nächste Folge der Kern-Technik. *(ofr)*

Infos

[1] Informationen zu den x86-Prozessoren: [http://developer.intel.com/design/Pentium4/documentation.htm]
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 2: Linux-Magazin 9/03, S. 86
[3] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur": Linux-Magazin 2/04, S. 28

Die Autoren

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

Listing 10: »include/asm-i386/unistd.h« neu

```
01 ...
02 #define __NR_mq_timedsend   (__NR_mq_open+2)
03 #define __NR_mq_timedreceive    (__NR_mq_open+3)
04 #define __NR_mq_notify      (__NR_mq_open+4)
05 #define __NR_mq_getsetattr  (__NR_mq_open+5)
06 #define __NR_udelay            283
07
08 #define NR_syscalls 284
09 ...
```

Listing 11: Testprogramm »testsys.c«

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <asm/unistd.h>
04
05 int main( int argc, char **argv )
06 {
07         int ret;
08
09         ret = syscall( __NR_udelay, 9000 );
10         printf("udelay(9000)=%d\n", ret);
11         ret = syscall( __NR_udelay, 10001 );
12         perror( "udelay" );
13         printf("udelay(10001)=%d\n", ret);
14         return 0;
15 }
```

User Friendly – der monatliche Comic im Linux-Magazin