Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 15

Kern-Technik

Dass es TCP/IP-Netzwerkfunktionen im Kernel gibt, überrascht kaum, denn schließlich implementiert der Betriebssystemkern den zugehörigen Stack. Das Nutzen der Funktionen in eigenen Kernelmodulen funktioniert aber etwas anders als bei den Userspace-Verwandten. Eva-Katharina Kunst, Jürgen Quade

Wer im Kernel eigene Netzwerkdienste programmiert, muss einige Eigenheiten des dort implentierten TCP/IP-Codes beachten, ähnlich wie beim Zugriff auf Dateien [1]. Glücklicherweise erinnert das Kernel-Interface an das der bekannten Userspace-Funktionen, siehe Kasten „TCP/IP im Userspace“. Die meisten Netzwerkzugriffe sind allerdings nur in einem Prozesskontext möglich. Deshalb muss sich eine TCP/IP-Funktionen im Kernel entweder den Prozesskontext einer Userspace-Anwendung borgen oder braucht einen eigenen Kernel-Thread, wie [2] beschreibt.

Listing 1 zeigt Ausschnitte eines Kernel-internen Echoservers, der die vorgestellten Funktionen benutzt – der komplette Code und das Makefile liegen auf [3]. Alle Daten, die der Server empfängt, schickt er an den Absender zurück. Das lässt sich leicht prüfen, siehe Kasten „Test mit Telnet“.

Im Kernel wird ein Socket durch die Funktion »sock_create()« erzeugt, siehe Kasten „TCP/IP-Schnittstellenfunktionen im Kernel“ und Abbildung 2. Das auf diese Weise erzeugte Socket-Objekt stellt in der Struktur »ops« weitere Methoden über Funktionspointer zur Verfügung, siehe den Typ »proto_ops« im Header »linux/net.h«.

Die Methode »bind()« weist dem Socket eine Adresse zu, die sie als erstes Argument erwartet. Die beiden übrigen Parameter entsprechen denen ihres Gegenstücks im Userspace: Das Element vom Typ »struct sockaddr_in« spezifiziert unter anderem den Port, auf den der Server hört, »addrlen« gibt die Länge dieser Datenstruktur an.

Der Port muss, wie bei Netzanwendungen üblich, mit Hilfe von »htons()« in Network Byte Order konvertiert werden. Die Funktion »listen()« lässt den vorbereiteten Socket schließlich auf Verbinungen warten. Neben der Adresse des Sockets steht hier als zweites Argument die Anzahl der Clients, die gleichzeitig zugreifen dürfen.

Sockets klonen

Nimmt ein Client Kontakt zum Server auf, brauchen er zur Kommunikation den existierenden Socket. Also muss der Server für jede neue Verbindung einen neuen Socket anlegen, was er im Kernel mit »socket_alloc()« erledigt, Listing 1 Zeile 47. Der neue Socket und der auf Verbindungen wartende sind Argumente der Funktion »ops->accept()«. Zusätzlich erwartet sie den Zugriffsmodus, wie er von der »open()« bekannt ist. Ohne besonderen Zugriffsmodus (»flags = 0«), blockiert der Kernel-Thread so lange, bis ein Client eine Verbindung aufbaut. Wird »flags« mit »O_NONBLOCK« initialisiert, kehrt »ops->accept()« sofort zurück. Die Funktion »socket_accept()« (Zeilen 40 bis 69) implementiert ungefähr dieselbe Funktionalität wie das im Userspace bestehende »accept()«.

Ein positiver Return-Wert von »ops->accept()« zeigt an, dass der Verbindungsaufbau erfolgreich war. Wer mehr über den Client erfahren möchte, ruft die Methode »getname()« auf, die dessen IP-Adresse und Port zurückgibt, siehe Listing 1, Zeilen 57 bis 67.

Zugriffsmodus festlegen

Der Datenaustausch läuft über die Funktionen »sock_recvmsg()« und »sock_sendmsg()«, Zeilen 89 und 112. Sie besitzen drei gleiche Parameter: den Socket, ein Objekt vom Typ »msghdr« und die Länge des Speicherbereichs. Bei »sock_recvmsg()« kommen noch die Flags für die Zugriffsart dazu, die »linux/socket.h« festlegt. Für einen nicht blockierenden Zugriff steht beispielsweise »MSG_DONTWAIT«.

Der wichtigste Parameter dieser Funktionen ist die Datenstruktur vom Typ »msghdr«. Sie übernimmt nämlich in »msg_iov« die Adresse des Speicherbereichs »iov«, in dem die Daten bei Lese- und Schreibzugriffen abgelegt werden, siehe Listing 1, Zeilen 79 bis 84. In »msg_iovlen« steht die Anzahl der verwendeten Speicherblocks, in diesem Fall nur einer. Die Strukturelemente »msg_

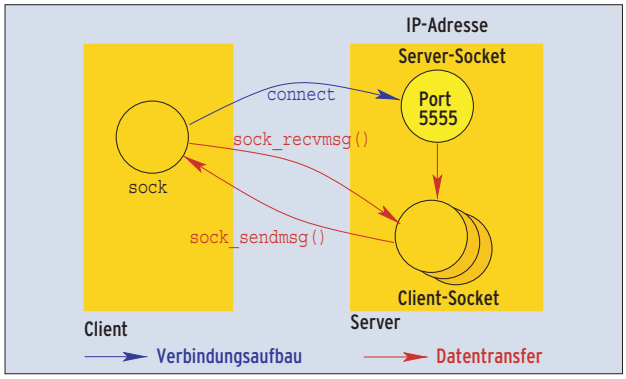

Abbildung 1: Beim Verbindungsaufbau erzeugt der Server einen neuen Port, der für den eigentlichen Datenaustausch verwendet wird.

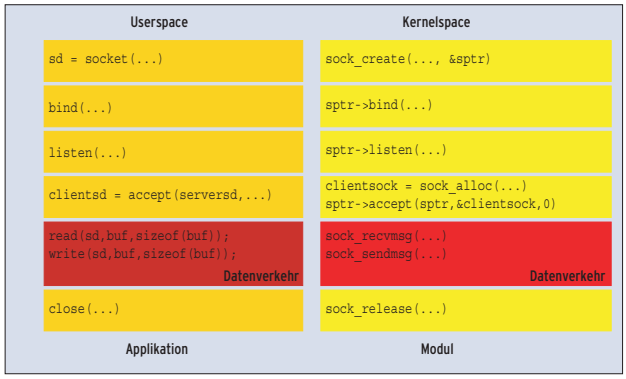

Abbildung 2: Die meisten Interfaces zu TCP/IP auf der Applikationsseite haben direkte Entsprechungen im Kernel.

control« und »msg_controllen« spielen im Beispiel keine Rolle und erhalten dementsprechend nur Standardwerte. Sie dienen sonst unter anderem der Übermittlung von IP-Optionen über Unix-Sockets. Näher sind die einzelnen Elemente von »msghdr« in der Manualseite zu der Funktion »recvmsg()« beschrieben, die so funktioniert wie die hier verwendete Kernelfunktion »sock_recvmsg()«.

Tricks zum Speicherzugriff

Die Makros »get_fs()« und »set_fs()« ermöglichen, wie in [1] beschrieben, den Datentransfer zwischen User- und Kernelspace, siehe Listing 1, Zeilen 87 bis 90. Die Funktion »sock_recvmsg()« gibt die Anzahl der empfangenen, »sock_sendmsg()« die der verschickten Bytes zurück. Ist zwischenzeitlich die Verbindung abgebrochen, liefern sie einen Wert kleiner null. Der Server ruft in diesem Fall – oder wenn er seinerseits die Verbindung beenden will – die Funktion »sock_release()« auf. Damit gibt der Programmierer auch später den verwendeten Socket wieder frei.

Nächster Kunde, bitte!

Im Gegensatz zum Server muss der Client nicht auf Verbindungen warten. Deshalb beschränkt sich die Implementation auf den Verbindungsaufbau. Zuerst erzeugt »socket_create()« einen Socket. Die Struktur »sockaddr_in« spezifiziert dann den gewünschten Server, siehe Zeilen 25 bis 27 in Listing 2. Dessen IP-Adresse darf übrigens auch in Ascii vorliegen, der so genannten Dotted Decimal Notation. Dafür bietet der Kernel die Funktion »in_aton()«, die die IP-Adresse in Binärform übersetzt. Die Struktur »sockaddr_in« ist einer der vier Parameter der Methode »connect()«, die den Server kontaktiert (Zeile 28).

Für den Datenaustausch benutzen Client wie Server die beiden Methoden »sock_recvmsg()« und »sock_sendmsg()«. Die Verbindung schließt der Client ebenfalls mit »sock_release()«. Der Client schickt ein »hallo« an den Echoserver, der auf demselben Rechner auf Port 5555 wartet (siehe Kasten „Test mit Telnet“). Er

TCP/IP im Userspace

Die Kommunikation mit der Internet-Standardprotokollfamilie TCP/IP funktioniert nach dem Client-Server-Prinzip. Der Client addressiert einen Server über die Kombination von IP-Adresse und Port, der Server wartet auf die Anfrage eines Clients. Ist es so weit, erzeugt er einen neuen Socket, über den der Datentransfer abläuft. Parallel dazu nimmt der Server bereits die nächste Anfrage entgegen (siehe Abbildung 1).

Um einen Userspace-Server zu programmieren, kommen überlicherweise die Funktionen »socket()«, »bind()«, »listen()« und »accept()« zum Einsatz. Der Client baut eine Verbindung über »socket()« und »connect()« auf (Abbildung 2).

Der Aufruf von »socket()« richtet einen Kommunikationsendpunkt ein, der als Socket bezeichnet wird. Die Funktion »bind()« weist dem Socket einen Port zu, der dem Client bekannt sein muss. Für einige Applikationen sind die Ports festgelegt (siehe die Datei »/etc/services«). So steht die Portnummer 80 für den HTTP-Service, also das übliche WWW-Protokoll. Der Aufruf von »listen()« legt die maximale Anzahl gleichzeitiger Verbindungen fest. Damit lässt sich die Gefahr einer Überlastung des Servers von vornherein minimieren.

»accept()« ist die Funktion, mit der der Server auf eine neue Verbindung wartet. Sobald ein Client per »connect()« eine Verbindung aufbaut, erzeugt diese Funktion einen neuen Socket, über den Client und Server Daten austauschen, meist über »read()« und »write()«. Mit dem ursprünglichen Socket wartet der Server auf die nächste Verbindung. Wird ein Socket nicht mehr benötigt, gibt die Funktion »close()« ihn wieder frei.

Test mit Telnet

Haben Sie das Server-Modul »echo.c« übersetzt und mit »insmod echo.ko« geladen, sollten Sie sicherstellen, dass Sie den hier verwendeten Port 5555 nicht anderweitig verwenden oder durch eine Firewall geblockt haben. Dann geben Sie »telnet localhost 5555« ein. Der Aufruf kann natürlich auch von einem anderen Rechner aus erfolgen. In diesem Fall ist »localhost« durch den Rechnernamen des neuen Echoservers zu ersetzen.

Sie können jetzt Nachrichten eintippen, die nach jedem Return zum Server gesendet und von diesem zurückgeschickt werden. Telnet gibt die Antworten des Echoservers dann aus. Um die Verbindung zu beenden, verlassen Sie einfach das Telnet-Programm. Dazu geben Sie [Strg]+[ ]] gefolgt von »quit« ein. Die Verbindung wird ebenfalls abgebrochen, wenn Sie den Echoserver per »rmmod echo« aus dem Kernel entladen.

Um den Client zu testen, laden sie entweder das Servermodul oder verwenden den Standard-Echoserver im Userspace. Läuft der Internet-Superdaemon »xinetd«, setzen Sie in »/etc/xinet.d/echo« die Variable »disabled« auf »No« und schicken dem Server ein HUP-Signal. Dann müssen Sie allerdings auch im Sourcecode des Clients den Port auf »7« ändern.

wartet auf die Antwort und baut die Verbindung anschließend wieder ab. Trägt man im Code von Listing 2 anstelle von »127.0.0.1« die IP-Adresse des Rechners im lokalen Netz ein, können Client und Server auch auf unterschiedlichen Rechnern laufen.

Um das Beispiel kurz zu halten, leiht sich das Modul den Prozesskontext von Insmod. Das Modul zeigt nach erfolgter Kommunikation dem Kernel einen Initialisierungsfehler an. Das erspart manuelles Entladen. Entfernt man den Kommentar von »//return 0;« in Zeile 55, verschwindet dieser Fehler, das Modul muss dann allerdings mit »rmmod« von Hand entladen werden. Den Testablauf dokumentiert der Client über »printk()« im Syslog.

Laufen Netzwerk-Services im Betriebssystemkern, sind sie besonders perfor-

Listing 1: »echo.c«

```
39 ...
40 static struct socket *socket_accept( struct
   socket *server )
41 {
42   struct socket *clientsocket=NULL;
43   struct sockaddr address;
44   int error, len;
45
46   if( server==NULL ) return NULL;
47   clientsocket = sock_alloc();
48   if( clientsocket==NULL ) return NULL;
49
50   clientsocket->type = server->type;
51   clientsocket->ops = server->ops;
52   error=server->ops->accept(server,
       clientsocket,0);
53   if( error<0 ) {
54     sock_release(clientsocket);
55     return NULL;
56   }
57   error=server->ops->getname(clientsocket,
58                         (struct sockaddr *)
       &address, &len,2);
59   if( error<0 ) {
60     sock_release(clientsocket);
61     return NULL;
62   }
63   printk(KERN_INFO "new connection (%d) from
       %u.%u.%u.%u\n", error,
64      (unsigned char)address.sa_data[2],
65      (unsigned char)address.sa_data[3],
66      (unsigned char)address.sa_data[4],
67      (unsigned char)address.sa_data[5] );
68   return clientsocket;
69 }
70
71 static int server_send( struct socket *sock,
   unsigned char *buf, int len )
72 {
73   struct msghdr msg;
74   struct iovec iov;
75   mm_segment_t oldfs;
76
77   if( sock->sk==NULL )
78     return 0;
79   iov.iov_base = buf;
80   iov.iov_len  = len;
81   msg.msg_control = NULL;
82   msg.msg_controllen = 0;
83   msg.msg_iov        = &iov;
84   msg.msg_iovlen = 1;
85   msg.msg_flags = 0;
86
87   oldfs = get_fs();
88   set_fs( KERNEL_DS );
89   len = sock_sendmsg( sock, &msg, len );
90   set_fs( oldfs );
91
92   return len;
93 }
94
95 static int server_receive( struct socket
   *sptr, unsigned char *buf, int len )
96 {
97   struct msghdr msg;
98   struct iovec iov;
99   mm_segment_t oldfs;
100
101   if( sptr->sk==NULL )
102     return 0;
103   iov.iov_base = buf;
104   iov.iov_len  = len;
105   msg.msg_control = NULL;
106   msg.msg_controllen = 0;
107   msg.msg_iov       = &iov;
108   msg.msg_iovlen = 1;
109
110   oldfs = get_fs();
111   set_fs( KERNEL_DS );
112   len = sock_recvmsg( sptr, &msg, len, 0 );
113   set_fs( oldfs );
114
115   return len;
116 }
117 ...
```

Listing 2: »client.c«

```
11 ...
12 static int __init client_init( void )
13 {
14   int len, clienterror;
15   char buf[64];
16   struct msghdr msg;
17   struct iovec iov;
18   mm_segment_t oldfs;
19   struct sockaddr_in client;
20
21   if( sock_create( PF_INET,SOCK_STREAM,
       IPPROTO_TCP,&clientsocket)<0 ) {
22     printk( KERN_ERR "server: Error
         creating clientsocket.\n" );
23     return -EIO;
24   }
25   client.sin_family = AF_INET;
26   client.sin_addr.s_addr = in_aton
       ("127.0.0.1"); /* destination addr */
27   client.sin_port = htons( (unsigned
       short)serverport );
28   clienterror = clientsocket->ops->connect
       ( clientsocket,
29   (struct sockaddr *) &client, sizeof
       ( client ), 0 );
30   if( clienterror < 0 ) {
31     printk( KERN_ERR "Carrera: Connect
         error = %d on clientsocket",
32         clienterror );
33     return -EIO;
34   }
35   memcpy( buf, "hallo", 6 );
36   iov.iov_base = buf;
37   iov.iov_len  = 6;
38   msg.msg_control = NULL;
39   msg.msg_controllen = 0;
40   msg.msg_iov        = &iov;
41   msg.msg_iovlen = 1;
42   msg.msg_flags = 0;
43
44   oldfs = get_fs();
45   set_fs( KERNEL_DS );
46   len = sock_sendmsg( clientsocket, &msg,
       6 );
47   if( len > 0 ) {
48     iov.iov_len  = sizeof(buf);
49     len = sock_recvmsg( clientsocket,
         &msg, sizeof(buf), 0 );
50     if( len > 0 )
51       printk( KERN_INFO "returned:
           \"%s\"\n", buf);
52   }
53   set_fs( oldfs );
54
55   //return 0;
56   if( clientsocket ) {
57     sock_release( clientsocket );
58     clientsocket = NULL;
59   }
60   return -EIO;
61 }
62
63 static void __exit client_exit( void )
64 {
65   if( clientsocket )
66     sock_release( clientsocket );
67 }
68
69 module_init( client_init );
70 module_exit( client_exit );
```

mant. So ergab eine von IBM durchgeführte Untersuchung, dass ein im Kernel implementierter Webserver eine dreifach bessere Performance aufwies als ein Webserver im Userspace [5].

Eingebauter Server in 2.6

Im Kernel 2.4 überzeugte der eingebaute Webserver Khttpd durch seine Geschwindigkeit, doch er ist mittlerweile nicht mehr Teil des Standardkernels. Der In-Kernel-Server Tux [6] von Ingo Molnar funktioniert mit den aktuellen Kernen, zum Redaktionsschluss zumindest mit Linux 2.6.5 – also eine Gelegenheit für eigene Experimente ohne großen Programmieraufwand. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 14: Linux-Magazin 9/04, S. 92
[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 4: Linux-Magazin 11/03, S. 96
[3] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/10/Kern-Technik/]
[4] E.-K. Kunst und J. Quade, „Kern-Technik", Folge 13: Linux-Magazin 8/04, S. 92
[5] Joubert, King et al.: „High-Performance Memory-Based Web Servers: Kernel and User-Space Performance": [http://www.usenix.org/events/usenix01/full_papers/joubert/joubert_html/index.html]
[6] In-Kernel-Webserver Tux: [http://people.redhat.com/mingo/TUX-patches/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

TCP/IP-Schnittstellenfunktionen im Kernel

Sockets erzeugen und freigeben	
`int sock_create(int family, int type, int protocol, struct socket **res);`	Die Funktion erzeugt einen Socket vom Typ »type« aus der Protokollfamilie »family« für das Protokoll »protocol«. Die Adresse des erzeugten Sockets steht in »res«. Die »family« ist normalerweise »PF_INET«, der Typ »SOCK_DGRAM« für einen UDP-Socket und »SOCK_STREAM« für einen TCP-Socket. Bei Erfolg gibt die Funktion »0« zurück, sonst einen negativen Fehlercode. Unter anderem kommen die folgenden Fehlercodes vor: »EAFNOSUPPORT« (ausgewählte »family« wird nicht unterstützt), »EINVAL« (ausgewählter »type« wird nicht unterstützt) und »ENFILE« (keine Socket-Ressource mehr verfügbar).
`void sock_release(struct socket *sock)`	Die Funktion gibt den Socket »sock« wieder frei.
`struct socket *sock_alloc(void);`	Die Funktion allokiert und initialisiert ein Socket-Objekt. Bei Erfolg gibt die Funktion die Adresse des neu erzeugten Socket-Objekts zurück, ansonsten »0«.
Methoden des Socket-Objekts	
`int (*bind)(struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);`	Die Funktion weist dem Socket »sock« den in »myaddr« angegebenen Namen (Portnummer) zu. »myaddr« hat die Länge »sockaddr_len«. Läuft alles glatt, gibt die Funktion »0« zurück, sonst einen negativen Fehlercode.
`int (*listen)(struct socket *sock, int len);`	Setzt die Anzahl der gleichzeitig offenen Verbindungen des Sockets »sock« auf »len«. Die Funktion gibt nach erfolgreichem Ablauf »0« zurück, sonst einen negativen Fehlercode.
`int (*accept)(struct socket *sock, struct socket *newsock, int flags);`	Die Methode nimmt vom verbindungsorientierten Socket »sock« den nächsten Verbindungswunsch entgegen und initialisiert »newsock« entsprechend. Mit »flags« wird der Zugriffsmodus (siehe Headerdatei »linux/socket.h«) eingestellt. Liegt ein Verbindungswunsch vor, gibt die Funktion »0« zurück, andernfalls einen negativen Fehlercode.
`int (*getname)(struct socket *sock, struct sockaddr *addr, int *sockaddr_len, int peer);`	Diese Funktion schreibt in »addr« Informationen über Portnummer und IP-Adresse des Sockets »sock«. Ist »peer« mit »0« belegt, gibt die Methode die Adressangaben des Sockets selbst zurück. Bei jedem anderen Wert für »peer« schreibt sie die Adressangaben des Remote-Sockets in »addr«. In »sockaddr_len« wird die Größe der abgelegten Adressinformation zurückgegeben. Der Rückgabewert ist »-ENOTCONN«, falls der Socket nicht verbunden ist, sonst »0«.
Funktionen zum Datenaustausch	
`int sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t size, int flags);`	Diese Funktion liest Daten vom Socket »sock« und legt sie im Speicher ab, der neben anderen Informationen in »msg« definiert ist. »size« gibt die Größe des Speicherbereichs in Bytes an. »flags« spezifiziert die Zugriffsart (beispielsweise »MSG_DONTWAIT« für nicht blockierend). Die Funktion gibt die Anzahl der gelesenen Bytes zurück. Sollte die Verbindung zwischenzeitlich abbrechen, gibt die Funktion einen negativen Fehlerwert zurück.
`int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size);`	Verschickt die im Speicherbereich abgelegten Daten an den über »sock« verbundenen Kommunikationspartner. Der Speicherbereich selbst ist in einem Element der Struktur »msg« definiert. »size« gibt die Größe dieses Speicherbereichs an. Die Funktion gibt die Anzahl der gesendeten Bytes zurück, im Fehlerfall oder beim Verbindungsabbruch stattdessen einen negativen Fehlercode. Weitere Informationen über die Fehlerursache lassen sich der Struktur »msg« entnehmen.
Hilfsfunktionen	
`__u32 in_aton(const char *str)`	Die Funktion konvertiert einen Ascii-String (Dotted Decimal) in eine binäre Repräsentation der IP-Adresse. Rückgabewert ist die konvertierte Adresse.

`unsigned short int htons( unsigned short int );`
`__u32 htonl(__u32 );`
`unsigned short int ntohs( unsigned short int );`
`__u32 ntohl(__u32 );`
	Diese vier Funktionen übersetzen einen Long- oder einen Integerwert vom der Host Byte Order in Network Byte Order und umgekehrt.