Netzwerkprogrammierung mit C++

Bernhard Trummer
Linux User Group Graz
für die Linuxtage03
bernhard.trummer@gmx.at

24. April 2003

Übersicht

System Calls: Wie erzeugt man Sockets? Wie arbeitet man mit Sockets?

Socket Multiplexing: Wie arbeitet man mit mehreren Sockets gleichzeitig?

Blocking vs. Non-blocking I/O: Unterschiede, Vorteile, Nachteile.

Datenübertragung: Wie kann man Daten über Sockets übertragen?

Design von Servern: Was ist alles zu berücksichtigen, wenn man einen Server schreiben will?

System Calls

- Jeder Socket ist in Wirklichkeit ein Filedeskriptor[1]. Dieser muß bei den System Calls als Parameter übergeben werden.
- Jeder System Call liefert im Fehlerfall den Returnwert -1 zurück. Über die Variable `errno` kann die genaue Fehlerursache abgefragt werden.
- Jeder System Call sollte auf Fehler überprüft werden!

[1] Dies entspricht der Unix-Philosophie: „Alles ist ein File“.

socket()

```
int socket(int domain, int type, int protocol);
```

domain: Die Protokollfamilie des Sockets (z.B. PF_INET oder PF_UNIX).

type: Der Socket-Typ (z.B. SOCK_DGRAM, SOCK_STREAM oder SOCK_RAW).

protocol: Zu verwendende Protokoll (meistens 0, da dieser Parameter nur selten gebraucht wird).

Returnwert: Der erzeugte Filedescriptor.

close()

```
int close(int fd);
```

fd: Der zu schließende Filedescriptor (bzw. Socket).

bind()

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

sockfd: Der zu bindende Socket.

my_addr: Pointer zu einer lokalen Adresse, die der Socket bekommen soll.

addrlen: Länge der angegebenen Adresse.

Ein IP-Socket kann an eine IP-Adresse, an einen Port oder an beides gebunden werden.

connect()

```
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
```

sockfd: Socket, der zu einem anderen Socket verbunden werden soll.

serv_addr: Pointer zur Adresse des Peers.

addrlen: Länge der angegebenen Adresse.

Im Falle SOCK_STREAM blockiert dieser System Call, bis der TCP-Handshake (SYN, SYN/ACK, ACK) abgeschlossen ist.

listen()

```
int listen(int s, int backlog);
```

s: Socket, der in den Listening-State versetzt werden soll.

backlog: Angabe, wie viele Verbindungen das Betriebssystem puffern soll.

Erst durch `connect()` oder `listen()` wird festgelegt, ob ein Socket einer Connection oder einem Listener entspricht.

accept()

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

s: Listener-Socket, von dem eine Verbindung akzeptiert werden soll.

addr: Optionaler Pointer zu einer Adress-Struktur, in der die Adresse vom Client gespeichert werden soll.

addrlen: Optionale Länge der gegebenen Client-Adresse. Diese Länge muß mit der Länge der Client Adress-Struktur initialisiert werden!

Returnwert: Der Filedeskriptor des aktzeptierten Sockets.

send(), sendto() und sendmsg()

```
int send(int s, const void *msg, size_t len, int flags);

int sendto(int s, const void *msg, size_t len, int flags, const struct
sockaddr *to, socklen_t tolen);

int sendmsg(int s, const struct msghdr *msg, int flags);
```

recv(), recvfrom() und recvmsg()

```
int recv(int s, void *buf, size_t len, int flags);

int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *fromlen);

int recvmsg(int s, struct msghdr *msg, int flags);
```

Erzeugen eines TCP Server Sockets

socket(): Erzeuge den Socket.

bind(): Binde den Socket an eine Adresse bzw. Port.

listen(): Setze den Socket in den LISTEN State.

accept(): Warte auf eingehende Verbindungen.

Erzeugen eines TCP-Sockets

socket(): Erzeuge den Socket.

bind(): Optional kann der Socket an eine Adresse gebunden werden, um so z.B. das Netzwerkinterface festzulegen.

connect(): Verbinde den Socket zu einem Server.

Erzeugen eines UDP-Sockets

socket(): Erzeuge den Socket.

bind(): Falls ein Listener erzeugt werden soll, muß er an eine Adresse gebunden werden.

connect(): „Verbinde“ den Socket zu einem Server.

Wird ein UDP-Socket zu einem Server „verbunden“, können statt den System Calls `recvfrom()` und `sendto()` die System Calls `recv()` und `send()` verwendet werden.

Socket Optionen

Jeder Socket besitzt verschiednene Parameter, die zur Laufzeit abgefragt und verändet werden können.

Siehe man 7 socket, man 7 ip und man 7 tcp für eine Liste aller Socket Optionen inkl. Dokumentation. Zu beachten ist, daß manche Socketoptionen betriebssystemspezifisch sind und daher nicht auf allen Plattformen verfügbar sind.

getsockopt() / setsockopt()

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t
*optlen);

int setsockopt(int s, int level, int optname, const void *optval, socklen_t
optlen);
```

s: Socket bei dem eine Socketoption ausgelesen bzw. gesetzt werden soll.

level: SOL_IP (Option auf IP-Ebene), SOL_SOCKET (Option auf TCP-Ebene)

optname: Name der Option (z.B. SO_REUSEADDR oder SO_RCVBUF)

optval: Pointer auf den Wert für die Socket Option.

optlen: Die Länge des optval Puffers.

Adress-Strukturen

```
1 struct sockaddr
2 {
3     sa_family_t sa_family;
4     char        sa_data[14];
5 };
```

```
1 struct sockaddr_in
2 {
3     sa_family_t  sin_family;
4     in_port_t sin_port;
5     struct in_addr sin_addr;
6     unsigned char sin_zero[8];
7 };
```

Socket Multiplexing

Ohne weitere Hilfsmittel ist es nicht möglich, mehrere Sockets „gleichzeitig“ zu behandeln. Mit Hilfe der System Calls `select()` bzw. `poll()` können (theoretisch) beliebig viele Sockets überwacht werden.

select()

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

n: Der größte zu überwachende Filedeskriptor plus 1.

readfds, writefds, exceptfds: Sets von Sockets (Filedeskriptoren) die auf Lese-, Schreib- und Exception-Events überwacht werden sollen.

timeout: Ein optionales Timeout für den `select()` call.

Die drei Filedeskriptor-Sets sind üblicherweise als Bitfelder implementiert, bei denen ein gesetztes Bit an der i-ten Stelle dem Filedeskriptor i entspricht.

Beispiel für select()

```
1   fd_set readSet;
2   struct timeval tv;
3   int rc;
4
5   FD_ZERO(&readSet);
6   FD_SET(s, &readSet);
7
8   tv.tv_sec = 5;
9   tv.tv_usec = 0;
10
11  rc = select(s+1, &readSet, NULL, NULL, &tv);
```

Was ist bei select() zu beachten?

- Vor jedem Aufruf von `select()` müssen mit `FD_SET()` alle zu überwachenden Sockets (neu) gesetzt werden. `select()` überschreibt nämlich den Inhalt dieser Bitfelder.

- Auch das Timeout sollte immer neu initialisiert werden, da es je nach Plattform von `select()` überschrieben wird.

- Der erste Parameter von `select()` gibt an, wie viele Filedeskriptoren aus den drei Sets in den Kernel kopiert werden. Er muß daher mindestens auf den Wert des größten Filedeskriptors plus 1 gesetzt werden.

poll()

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

ufds: Pointer auf ein Array von `pollfd` Strukturen.

nfds: Die Anzahl der `pollfd` Strukturen im Array.

timeout: Ein Timeout in Millisekunden.

```
1 struct pollfd {
2         int fd;           /* file descriptor */
3         short events;     /* requested events */
4         short revents;    /* returned events */
5 };
```

Beispiel für poll()

```
1   struct pollfd fds[2];
2   int rc;
3   int timeout;
4
5   fds[0].fd = s1;
6   fds[0].events = POLLIN
7   fds[1].fd = s2;
8   fds[1].events = POLLOUT;
9
10  int timeout = 5000;
11
12  rc = poll(fds, 2, timeout);
```

Was ist bei poll() zu beachten?

- Der Parameter nfds entspricht der Anzahl der `pollfd` Strukturen, die übergeben werden sollen, nicht der `sizeof()`!
- Jeder Aufruf von `poll()` verändert nur die `revents` der `pollfd` Strukturen. Die `events` müssen daher nicht immer neu initialisiert werden.
- Soll ein Socket vorrübergehend nicht überwacht werden, müssen die `events` auf 0 gesetzt werden.
- Wird ein Socket geschlossen, so muß der entsprechende `pollfd` Eintrag entfernt werden. Wird darauf „vergessen“, selektiert `poll()` das POLLNVAL event und kehrt immer sofort zurück.

Non-blocking I/O

Die System Calls `recv()` und `recvfrom()` können blockieren, wenn keine Daten im Empfangspuffer vorhanden sind. Der System Call `send()` blockiert so lange, bis der gesamte angegebene Puffer in den Sendepuffer übernommen worden ist.

Weiters blockiert der System Call `connect()`, bis feststeht, ob der Verbindungsaufbau erfolgreich war oder fehlgeschlagen ist. Und `accept()` blockiert, bis ein Client eine Verbindung zum Listener aufgebaut hat.

Manchmal ist es wünschenswert, daß keiner dieser System Calls blockiert. Dadurch ist es im Programmablauf möglich, anstatt zu blockieren etwas „sinnvolles“ zu tun.

Kombination mit poll() bzw. select()

Wenn man mit `poll()` (bzw. `select()`) arbeitet, sollte zusätzlich immer non-blocking I/O verwendet werden. Über `poll()` kann nämlich nur festgestellt werden, ob z.B. Daten empfangen werden können, jedoch nicht, wie viel empfangen werden kann.

Permanentes Umstellen

```
1 fcntl(F_GETFL)
2 flags |= O_NONBLOCK
3 fcntl(F_SETFL)
4
5 fcntl(F_GETFL)
6 flags &= ~O_NONBLOCK
7 fcntl(F_SETFL)
```

recv() und send()

Die Calls recv() und recvfrom() geben den Returnwert -1 zurück und setzen errno auf EAGAIN. Bei send() unterscheidet man zwei Fälle. Wenn ein Teil des Puffers übernommen werden kann, entspricht der Returnwert der Anzahl an übernommenen Bytes. Ist der Sendepuffer voll, gibt send() ebenfalls -1 zurück und setzt errno auf EAGAIN.

Non-blocking Connect 1/2

Der System Call connect() blockiert so lange, bis der Verbindungsaufbau geklappt hat oder gescheitert ist. Ein non-blocking connect erfolgt nach dem Setzen des O_NONBLOCK Flags in folgenden Schritten[2]:

- Aufruf von connect()
- Ein Returnwert von 0 entspricht einem erfolgreichen Aufbau.
- Ein Returnwert von -1 und errno == EINPROGRESS

[2] Siehe man 2 connect

Non-blocking Connect 2/2

- Mit select() bzw. poll() kann gewartet werden, bis die Verbindung aufgebaut ist. Dabei muß der Socket im Write-Set bzw. POLLOUT selektiert werden.

- Auslesen der Socketoption SOL_SOCKET/SO_ERROR, um festzustellen, ob der Verbindungsaufbau erfolgreich war.

Datenübertragung

Direkte Übertragung

```
1  struct Message
2  {
3    int i1;
4    char c;
5    /* etc. */
6  };
7
8  ...
9
10 Message m;
11 /* Initialize m */
12 socket.send(&m, sizeof(m));
```

Nachteile einer direkten Übertragung

Bei dieser direkten Übertragung ergeben sich jedoch folgende Probleme, sobald zwischen zwei verschiedenen Plattformen Daten ausgetauscht werden sollen:

- Ein int muß nicht überall gleich groß sein (32bit vs. 64bit).
- Die Byte-Order kann unterschiedlich sein (Little bzw. Big Endian).
- Das Byte Alignment kann je nach Plattform bzw. Compiler unterschiedlich sein, wodurch die Structs verschieden groß sind.
- Wird über die Struct ein Pointer auf eine andere Struct übertragen, kann der Peer mit diesem Pointer nichts anfangen.

Einsatz einer direkten Übertragung

Die einzig sinnvolle Einsatzmöglichkeit der direkten Übertragung besteht in der lokalen Kommunikation zwischen zwei Threads bzw. Prozessen einer Applikation. Hier kann auf eine Serialisierung der zu übertragenden Daten verzichtet werden, da der Sender und Empfänger auf der gleichen Maschine laufen und mit dem gleichen Compiler übersetzt worden sind.

Serialisierung von Daten

Bei einer Datenübertragung ist eine Serialisierung der lokalen Daten in eine plattformunabhängige Repräsentation (Data Description Language) und umgekehrt notwendig. Dies kann sowohl ASCII-basierend (z.B. XML) als auch in binärer Form (z.B. XDR[3] oder ASN.1/BER) erfolgen.

Was für eine DDL verwendet wird, hängt letztendlich vom Einsatzzweck des Servers ab (z.B. ob ein Protokoll implementiert werden soll, daß per Definition eine bestimmte DDL verwendet).

Siehe `06-codec.cpp` für ein Binär- und ASCII-Beispiel einer DDL.

[3] Wird u.a. für RPC verwendet. Siehe RFC1014

Designs für Server

Allgemeine Richtlinien

Bullet-Proofness: Egal was passiert, der Server muß immer einen definierten Zustand haben.

Blockieren: Egal was passiert, der Server darf nie in einem System Call blockieren. Ausnahmen sind natürlich `poll()` und `select()`. :-)

Prüfung empfangener Daten: Empfangenen Daten darf nie vertraut werden. Es ist immer eine Plausibilitätsprüfung notwendig.

Bearbeiten von Requests

- Bearbeiten im Hauptprozeß
- Erzeugen eines Child-Prozesses pro Request.
- Erzeugen eines Threads pro Request.

Direkte Bearbeitung im Hauptprozeß

- Leicht zu debuggen.
- Für kleinere Server durchaus ausreichend.
- Skaliert u.U. schlecht.
- Wenn der Prozeß abstürzt, ist der ganze Server weg.

Child-Prozesse

- Günstig, wenn alle Requests voneinander unabhängig behandelt werden können (d.h. wenn kein Zugriff auf gemeinsame Datenstrukturen notwendig ist).
- Wenn ein Child abstürzt, laufen alle anderen Childs und der Parent (der eigentliche Server) ganz normal weiter.
- Das Erzeugen der Child-Prozesse kostet etwas Zeit, was sich auf die Latenz des Servers auswirkt.
- Auf Multiprozessormaschinen können die Child-Prozesse parallel ausgeführt werden.

Threads 1/2

- Günstig, wenn mehrere Requests voneinander abhängig sind, da alle Threads z.B. Zugriff auf globale Datenstrukturen im Heap haben.
- Threads können i.A. schneller erzeugt werden, als Child Prozesse.
- Auch Threadwechsel sind i.A. schneller, als Prozeßwechsel.
- Wenn ein Thread abstürzt, kann der ganze Server als abgestürzt betrachtet werden.
- Fehler im Design (z.B. Synchronisation) sind schwer zu finden und machen sich laut Murphy erst im Live-System bemerkbar.

Threads 2/2

- Threadwechsel können jederzeit (und unreproduzierbar) stattfinden, was die Fehlersuche und das Debugging sehr schwierig gestaltet.

- Ruft ein Thread `close()` auf einen Socket auf, der in einem anderen Thread per `select()` überwacht wird, blockiert `close()`, bis der andere Thread von `select()` zurückkehrt.

- Auf Multiprozessormaschinen können Threads zwar auch aufgeteilt werden. Allerdings kann durch die notwendige Synchronisation der Threads die Performance schlechter sein, als bei einer Einprozessormaschine.

Design-Variationen

Pool-Modelle: Beim Hochfahren des Servers kann ein Pool von Childs/Threads im Voraus angelegt werden. Dadurch „entfällt“ beim Bearbeiten der Requests die Zeit für das Anlegen.

Mehrfachbenutzung: Jeder Child/Thread kann auch mehrere Requests gleichzeitig behandeln. Somit sind weniger Childs/Threads notwendig.

Aufgabentrennung: Meistens kann die Bearbeitung von Requests in mehrere Schritte aufgetrennt werden. Werden diese Schritte von separaten Childs/Threads behandelt, kann sich dadurch die Synchronisation und das Testen vereinfachen.

Dokumentation

- Die man-Pages der jeweiligen System Calls. :-)
- W. Richard Stevens – Unix Network Programming
- Unix Socket FAQ: http://www.developerweb.net/sock-faq/
- Usenet: de.comp.os.unix.programming
- Fefe – Skalierbare Netzwerkprogrammierung (Folien zum 19C3 Vortrag) http://www.fefe.de/scalable-networking.pdf