Vorlesung Client/Server Programmierung

Teil 1 – Einführung und Socket API

Inhalt

Inhaltsverzeichnis

Einführung

Was ist Client/Server

In der Informatik wird ein Programm, das Daten oder Ressourcen zur Verfügung stellt als “Server” bezeichnet. Ein Server weiß nicht, wann seine Dienste in Anspruch genommen werden, er muß also ständig verfügbar sein. Computer, auf denen Serverprogrammelaufen, werden als „Host“ bezeichnet [1].

Ein Programm, das die Dienste eines Servers verwendet, wird als „Client“ bezeichnet. Der Client initiiert die Kommunikation mit dem Server und bestimmt damit den Zeitpunkt der Kommunikation.

Das „Client- Server- Prinzip“ in der Informatik bedeutet, daß eine Anwendung in zwei Teile aufgespalten wird, den Client und den Server. Man unterscheidet zwischen „thin clients“ und „fat clients“.

„thin clients“ sind Clients, die nur sehr wenig selbst machen, sie fragen im Prinzip nur Eingaben vom Benutzer ab, schicken diese an den Host, der dann die verarbeiteten Daten wieder an den Client zur Anzeige schickt.

„fat clients“ übernehmen Teile der Verarbeitung selbst und holen sich nur bestimmte Daten vom Server.

Beispiele von Thin Clients sind Webbrowser, klassische X-Terminals und Datenbankanwendungen, bei denen der Client nur die Ein- und Ausgabe der Daten übernimmt (Bestellterminals bei großen Versandhäusern, Terminals bei Behörden).

Fat Clients wären z.B. Anwendungen wie Lotus Notes, da hier beim Client nicht nur Daten angezeigt werden, sondern auch Daten auf komplexe weise verarbeitet werden können.

1 Oft wird ein Computer, auf dem ein Serverprogramm läuft, als “Server” bezeichnet. Dies ist falsch, die korrekte Bezeichnung für einen Computer, auf dem ein Serverprogramm läuft ist “Host”.

„two-tier“[2] [3] Struktur

Die „two-tier“-Architektur ist eine Client-Server Architektur, die aus zwei Schichten besteht:

- einem Server
- einm Fat Client

In dieser Aufteilung stellt der Server typischerweise nur Daten zur Verfügung (Datenbankserver) und der Client implementiert die Verarbeitungslogik und die Darstellung.

„multi-tier“ Struktur

In einer „multi-tier“-Architektur wird die Client-Server Anwendung auf mehrere Schichten aufgeteilt. Typischerweise sind das drei Schichten:

- User Interface
- Verarbeitungssserver („Buisness Logic“ etc.)
- Datenbankserver

„Peer-to-Peer[4]“ Struktur

Die „Peer-to-Peer“[5] Struktur ist keine Client-Server-Struktur. Hier werden die Unterschiede zwischen Client unde Server aufgehoben. Ein Server kann hier auch Client sein (gleichzeitig), und umgekehrt.

Die EMAIL Server im Internet besitzen eine Peer-to-Peer Struktur, da die einzelnen Server gleichzeitig EMAIL verschicken (als Client von anderen EMAIL Servern arbeiten) und EMAIL empfangen (Server fuer andere Clients sein).

Die Webserver und Webbrowser dagegen sind klassische Client-Server Strukturen, da ein Webbrowser im Allgemeinen keine Serverfunktionen besitzt, und ein Webserver keine Browserfunktionalität.

Peer-To-Peer Netzwerke sind sehr dezentral, da es keine didizierten Server gibt. Solche Netzwerke sind sehr Ausfallsicher und schwer zu überwachen.

Moderne Beispiele für Peer-To-Peer Netzwerke sind Tauschbörsen. Es gibt neuerdings auch Groupware-Systeme, die eine Peer-To-Peer Struktur besitzen.

2 „tier“ = Schicht
3 Vgl. http://www.canberra.edu.au/~sam/whp/appl-arch.html
4 „peer“ = Ebenbürtiger, Gleichgestellter
5 Das ist die allgemeine Lesart von „P2P“. Es kann auch „Program-to-Program“ heißen.

Sockets[6]

Im Folgenden werden wir die Benutzung der „Socket API“, eine Programmierschnittstelle für die Netwerkprogrammierung. Diese API wurde 1983 in BSD Unix eingeführt, und die meisten komerziellen und freien UNIX derivate basieren auf diesem Code. LINUX bietet eine Implementation der Socket API, die **nicht** auf der BSD Codebase beruht, sondern neu entwickelt wurde.

Neben der Sockt- API gibt es noch die XTI API, die sich jedoch nicht durchsetzen konnte und die ich hier nicht behandeln möchte.

Liste der Systemcalls

Es folgt eine Liste der Systemcalls, die in diesem Kapitel erklärt werden:

Systemcall	Kurzbeschreibung
socket(2)	Erstellt einen Socket zur Kommunikation für eine bestimmte Adressfamilie. Socket ist noch nicht „benannt“, d.h. hat noch keine Protokolladresse. Verschiedene Sockettypen möglich. Liefert Filedeskriptor.
connect(2)	Startet den Verbindungsaufbau (3WHS bei TCP). Protokolladresse des Kommunikationspartners ist anzugeben, ebenso deren Länge. Liefert Fehlercode.
bind(2)	Bindet einen Socket an eine Protokolladresse, d.h. „benennt“ einen unbenannten Socket.
listen(2)	Teilt dem OS mit, daß der Prozess, d.h. Der Server nun bereit ist, auf eingehende Verbindungen zu hören.
accept(2)	Holt die nächste anstehende Verbindung aus der Queue des OS. Blockiert, solange keine Verbindung vorhanden. Liefert **neuen** Socket.
getsockname(2)	Liefert den Namen (Protokolladresse) eines Sockets.
getpeername(2)	Liefert den Namen (Protokolladresse) des Peers (Clients) eines Sockets. Socket muß verbunden sein.
getsockopt(2)	Lesen von Socketoptionen. Siehe **socket(7)** und **tcp (7)**.
setsockopt(2)	Setzen von Socketoptionen. Siehe **socket(7)** und **tcp (7)**.

6 Vgl. Auch Stevens: “UNIX Network Programming, Volume 1”

Sockets

Ein „Socket“ ist ein bidirektionaler Kommunikationsendpunkt. Sockets sind Objekte, die vom Betriebssystem verwaltet werden. Ein Prozess kann über Systemcalls beim Betriebssystem einen neuen Socket anfordern. Das Betriebssystem vergibt eine Nummer, über die dann der neue Socket referenziert werden kann.

Sockets sind unabhängig von der Art der Kommunikation (Protokoll, Vervindunslos oder Verbindungsorientiert). Beim Erzeugen des Sockets muss daher angegeben werden, für welche „Adressfamilie“[7] der Socket erzeugt werden soll (Ipv4, Ipv6, UNIX-Socket, ...). Außerdem muss der Typ der Verbindung fesgelegt werden (Verbindungsorientier, Verbindungslos). Zusätzlich kann noch eine Protokollnummer angegeben werden, da es theoretisch möglich ist, daß für einen Sockettyp in einer bestimmten Adressfamiliemehrere Protokolle implementiert sind[8].

Adressfamilien

Adressfamilie	*Bedeutung*	*Manpage*
PF_UNIX, PF_LOCAL	„Unix-to-unix“ Socket	unix (7)
PF_INET	IPv4	ip (7)
PF_INET6	IPv6	ip (7)

Sockettypen

Typ	*Bedeutung*	*Bei IP*
SOCK_STREAM	Geordnete, Zuverlässig, Bidirektional, Verbindungsorientiert	TCP
SOCK_DGRAM	Paketbasiert, Unzuverlässig, Verbindungslos	UDP
SOCK_RAW	Low-Level zugriff auf den Networklayer	-

7 „domain“ in der Manpage

8 Bei uns ist hier immer „0“ anzugeben, da wir nur IPv4 verwenden und hier keine zusätzlichen Protokolle implementiert sind.

Typ	*Bedeutung*	*Bei IP*
SOCK_SEQPACKET	Geordnet, Zuverlässig, Verbindungslos	nicht implementiert

Erzeugen eines Sockets

Sockets werden mit dem Systemcall **socket (2)**[9] erzeugt:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Dieser Systemcall liefert deinen **Filedescriptor** zurück. Dieser Filedescriptor ist ein postitive, kleine Integer. Er dient dem Programm als Referenznummer für das im Betriebssystem angelegte Socket-Objekt. Diese Referenznummer muss immer angegeben werden, wenn mit dem Socket gearbeitet wird.

Schließen eines Sockets

Ein Socket wird, wie eine normale Datei, mit **close (2)** geschlossen. Nach dem Schließen des Sockets ist der dazugehörige Filedescriptor für diesen Prozess ungültig.

Beispiel: Erzeugen und Schließen eines TCP-Sockets

```
#include <sys/socket.h>
#include <sys/types.h>

void main( void )
{
       int sock;      /* filedescriptor des sockets */

       /* Socket erzeugen. IPv4, STREAM (TCP) */
       sock = socket( PF_INET, SOCK_STREAM, 0);
       if ( sock <= 0 ) { ....fehlerbehandlung... }

       /* Schliessen des Sockets */
       close(fd);
}
```

Protokolladressen

Für die meisten Systemcalls, die Sockets bearbeiten, wird eine **Protokolladresse** benötigt. Diese werden in Strukturen abgelegt. Jede Adressfamilie besitzt u.U. andere Protokolladressen (IPv4, IPv6, UNIX_DOMAIN, etc). Um dennoch nicht für jede Adressfamilie eigene Systemcalls bereitzustellen, wird eine **generische Struktur** definiert:

```
struct sockaddr {
       uint8_t              sa_len; /* Länge der Struktur */
       sa_family_t    sa_family;     /* Adressfamilie */
       char           data[14];      /* generische Daten */
}
```

9 Diese Angabe bedeutet, daß dei Manpage zu diesem Systemcall in der Sektion „2“ zu finden ist (d.h. man 2 socket)

In dieser generischen Struktur ist die Adressfamilie codiert, alle Systemcalls erwarten Zeiger auf diese Struktur. Anhand der codierten Adressfamilie kann dann innerhalb des Systemcalls entschieden werden, wie die Protokolladresse wirklich aussieht. Für IPv4 ist die **struct sockaddr_in** in **<netinet/in.h>** definiert:

```
struct in_addr {
       in_addr_t      s_addr;        /* IPv4 Adresse (32 bit) */
}

struct sockaddr_in {
       uint8_t               sin_len;       /* Länge der Stuktur */
       sa_family_t           sin_family;    /* AF_INET */
       in_port_t             sin_port;      /* Port */
       struct in_addr sin_addr;      /* IP Adresse */
       char                  sin_zero[];    /* padding */
}
```

Der Eintrag **sin_len** ist Optional, d.h. nicht muß nicht auf jeder Platform vorhanden sein. Unter Linux ist **sin_len** nicht vorhanden [10]

Byteorder

Unterschiedliche Computersysteme können Daten auf unterschiedliche weise speichern. Der Unterschied besteht in der Reichenfolge, wie die einzelnen Bytes im Speicher abgelegt werden:

little endian: Hier werden die Datenbytes in aufsteigender Reihenfolge im Speicher abgelegt, d.h. das LSB eines 16-bit Wertes befindet sich auf der niedrigsten Adresse.

Adresse A + 1	***Adresse A***
MSB	LSB

big endian: Hier werden die Datenbytes in absteigender Reihenfolge im Speicher abgelegt, d.h. das LSB eines 16-bit Wertes befindet sich auf der höheren Adresse.

Adresse A + 1	***Adresse A***
LSB	MSB

Für Netzwerke ist eine einheitliche Byteorder definiert, die **Network Byte Order**. Das **IP** verwendet **big-endian**. Zur Umrechnung zwischen **Host Byte Order** und **Network Byte Order** sind Funktionen definiert. Diese sind **unbedingt** zu verwenden, um z.B. Die Protokolladressen

10 Vgl. auch [UNPv1] 58ff

auszufüllen.

Um Werte von der **Host Byte Order** in die **Network Byte Order** zu wandeln, werden die Funktionen **htons()** und **htonl()** verwendet. „Host to Net short“ und „Host to Net long“.

Umgekehrt werden **ntohs()** und **ntohl()** verwendet.

Aufbau einer Verbindung

Um von einem Client aus eine Verbindung zum Server aufzubauen, verwendet man **connect()**.

Im Falle von **TCP** wird duch **connect()** ein Verbindungsversuch gestartet (TCP Three- Way- Handshake, **3WHS**). Die Funktion blockiert solange, bis eine Verbindung zustande kommt, oder ein Fehler auftritt.

Beu **UDP** wird mit connect() keine Verbindung aufgebaut (Verbundungslos!), aber es wird eine default- Adresse eingestellt, an die Pakete gesendet werden, und von der Pakete empfangen werden.

Beispiel: Einfacher TCP Client

Prinzipiell werden folgende Schritte durchgeführt:

- Socket Anlegen
- Socket an Protokolladresse binden
- Verbindung zum Server herstellen
- Mit Server Kommunizieren
- Socket schließen

Funktion

Das Programm verbindet sich mit einem Host, dessen IP als Parameter übergeben werden muss. Wird kein Port angegeben (2. Parameter), wird Port 80 verwendet.

Kommt eine Verbindung zustande, wird eine einfacher HTTP Request an den Host geschickt, und die Antwort gelesen und ausgegeben. Dann beendet sich das Programm.

Beschreibung Beispielquelltext

Zeilen 1- 7:

Hier werden einige wichtige Headerdateien includiert, die u.A. die Systemcalls zur Socketprogrammierung und diverse Konstanten enthalten. Die benötigten Headerfiles sind zu jedem Systemcall in den

Manpages aufgelistet.

Außerdem wird noch „debug.h“ eingebunden, das ein einfaches Log-Makro enthält.

Zeilen 9, 10:

Defaultport und Buffergröße werden definiert.

Zeilen 12 – 16:

Hier wird eine Funktion definiert, die eine Kurzhilfe(„usage“) ausgibt und dann das Programm beendet.

Zeile 18:

Die C Einsprungfunktion **main()** wird definiert.

Zeilen 20 – 27: Variablendeklarationen

Ich benutze immer eine Variable **ret** um den Rückgabewert von Funktionsaufrufen abzulegen.

- **sock:** Filedescriptor des Sockets, über den wir kommunizieren
- **port:** Port, an den wir uns verbinden.
- **buf**: Buffer für die Daten vom Host.
- **req:** Request- string den wir an den Host schicken (HTTP).
- **serv_addr:** Struktur für die Protokolladresse der Serverprogramms auf dem Host.

Zeilen 31 – 35: Auswerten der Kommandozeilenparameter

Wenn kein Parameter angegeben wurde, dann geben wir die Kurzhilfe aus und beenden das Programm. Ggf. wird der Port gesetzt.

Zeilen 37 – 42: Socket erstellen

Hier wird mit **socket(2)** ein TCP Socket im IPv4 Adressraum erzeugt. Sehen Sie sich hierzu die Man Page zu **socket()** an. Die Konstanten **PF_INET** und **SOCK_STREAM** wählen IPv4 und TCP aus. Als Protokoll geben wir **0** an, da keine weiteren Protokolle definiert sind. Der Systemcall liefert einen Filedescriptor oder einen Fehlercode. Filedescriptoren sind immer positiv, wir prüfen deshalb ob **sock** negativ ist. Wenn ja, geben wir mit **perror(3)** den Fehlergrund aus und beenden das Programm.

Zeile 44: Löschen der Protokolladresse

Hier löschen wir mit **bzero(3)** die Struktur der Protokolladresse. Das wird gemacht, da wir nicht alle Felder setzen und sicherstellen wollen, das der Rest mit **0** belegt ist.

Zeile 45, 46: Adressfamilie und Port

Hier wird in der Protokolladresse die Adressfamilie auf IPv4 gesetzt. Die Konstante **PF_INET** ist in den Systemheadern definiert.

Außerdem wird hier der Port gesetzt, an den wir uns am Host verbinden wollen. Da der Port in **network byte order** angegeben werden muß, verwenden wir **htons(3)** um von Host- nach Netzwerkbyteorder zu konvertieren.

Zeile 48-53: IP-Adresse

Hier wird die IP-Adresse der Protokolladresse gesetzt. Dabei wird der String von der Kommandozeile mit **inet_pton(3)** in eine IP-Adresse umgewandelt und im Member **sin_addr** der Protokolladresse abgelegt. **inet_pton()** kann für IPv4 und IPv6 verwendet werden, deshalb muß auch die Adressfamilie mit angegeben werden (hier **PF_INET** für IPv4). Liefert **inet_pton()** einen Fehler, so ist die übergebene IP-Adresse nicht korrekt bzw. im falschen Format.

```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/socket.h>
4. #include <netinet/in.h>
5. #include <arpa/inet.h>
6.
7. #include "debug.h"
8.
9. #define BUF_SIZ            80
10.#define DEFAULT_PORT       80
11.
12.int usage( char *pname )
13.{
14.     printf( "%s: ip port\n", pname );
15.     exit( 10 );
16.}
17.
18.int main( int argc, char *argv[] )
19.{
20.     int ret = 0;
21.     int sock;
22.     int port = DEFAULT_PORT;
23.     char buf[ BUF_SIZ + 1];
24.     char req[]=
25.             "GET / HTTP/1.1\n"
26.             "host: gazong\n"
27.             "\n";
28.
29.     struct sockaddr_in     serv_addr;
30.
31.     if ( argc <= 1 )
32.             usage( argv[0] );
33.
34.     if ( argc > 2 )
35.             port = atoi( argv[2] );
36.
37.     sock = socket( AF_INET, SOCK_STREAM, 0 );
38.     if ( sock < 0 ) {
39.             perror( "socket" );
40.             ret = 11;
41.             goto DONE;
42.     }
43.
44.     bzero( &serv_addr, sizeof serv_addr );
45.     serv_addr.sin_family = AF_INET;
46.     serv_addr.sin_port = htons( port );
47.
48.     ret = inet_pton( AF_INET, argv[1], &serv_addr.sin_addr );
49.     if ( ret <= 0 ) {
50.             fprintf( stderr, "invalid IP: %s\n", argv[1] );
51.             ret = 12;
52.             goto DONE;
53.     }
54.
55.     ret  =  connect(  sock,  (struct  sockaddr  *)&serv_addr,  sizeof
   serv_addr );
56.     if ( ret ) {
57.             perror( "connect" );
58.             ret = 13;
59.             goto DONE;
60.     }
61.
62.     write( sock, req, sizeof req );
63.
64.     while ( 1 ) {
65.             int n;
```

```
66.
67.             n = read( sock, buf, BUF_SIZ );
68.             if ( n<= 0 )
69.                     break;
70.
71.             buf[n] = 0;
72.             fputs( buf, stdout );
73.     }
74.
75.     close( sock);
76.
77.     ret = 0;
78.DONE:
79.     return ret;
80.}
```

Binden eines Sockets an eine Protokolladresse

Auf der **Serverseite** muß ein TCP Socket, bevor er benutzt werden kann, **benannt** werden. Sockets, die von **socket()** erzeugt wurden, existieren zwar bereits in einem speziellen Namensraum (d.h. „Ipv4 TCP Stream Socket“, „UNIX Domain Socket“, etc), haben aber noch keine eindeutige Protokolladresse. Dies wird bei Verbindungorientireten Protokollen aber benötigt, damit das Betriebssystem die eingehenden Verbindungen zuordnen und annehmen kann.

Benannt wird ein Socket mit dem Systemcall **bind(2)**.

```
#include <sys/types.h>
#include <sys/socket.h>

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

Die Parameter sind analog zu **connect(2)** zu versehen und auszufüllen, bedeuten hier aber die Protokolladresse, auf die sieser Serverprozess „hört“.

Für IP kann auch ein spezieller Wert angegeben werden, der auf alle IP adressen passt, d.h. Der Serverprozess hört auf allen IP Adressen, die auf dem System vorhanden sind.

Der Name eines Sockets kann mit **getsockname(2)** geholt werden:

```
#include <sys/socket.h>

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

Hören auf Verbindungen

Der Systemcall **socket(2)** generiert normalerweise einen „aktiven“ Socket, d.h. ein Socket, über den man eine Verbindung aufbauen kann.

Der Systemcall **listen(2)** teilt dem OS mit, daß der Severprozess nun bereit ist, Verbindungen zu akzeptieren. Das OS erzeugt eine Queue für diesen Socket, in der das OS vollständige **und** unvollständige Verbindungen einreiht.

listen(2) benötigt einen Parameter, der die Queuegröße angibt. Dieser Parameter, **backlog** genannt, ist aber je nach OS unterschiedlich definiert. **backlog** gibt ist jedoch immer eine Maßzahl für die Anzahl der Verbindungen, die in der Queue für einen bestimmten Socket gehalten werden. Ist die Queue voll, so lehnt das OS weitere Verbindungsversuche ab (vgl. auch **SYN Flooding**).

```
#include <sys/socket.h>

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

Akzeptieren von Verbindungen

Mit **accept(2)** wird versucht, die nächste **vollständige** Verbindung aus der Queue des Sockets vom OS abzuholen. Ist keine Verbindung verfügbar, so blockiert **accept()** normalerweise (bei Sockets, die nicht als **non- blocking** gekennzeichnet sind).

accept(2) wird nur für Verbindungsorientierte Protokolle verwendet (**AF_INET SOCK_STREAM**).

accept(2) erzeugt einen **neuen** Socket, über den mit dem Client kommuniziert werden kann. Der originale (listen-) Socket wird nicht verändert.

accept(2) muß ein Socket übergeben werden, der mit **socket(2)** erzeugt, mit **bind(2)** an eine Protokolladresse gebunden und mit **listen (2)** in den „listen“- Status gebracht worden ist (d.h. das OS akzeptiert Verbindungen und reiht sie in die Queue ein).

Accept liefert die Protokolladresse des Verbindungspartners und deren Länge, wenn die Zeiger nicht **NULL** sind.

```
#include <sys/types.h>
#include <sys/socket.h>

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

Blockieren **mehrere** Prozesse **gleichzeitig** in **accept(2)**, so wählt das Betriebsystem einen zufälligen Prozess aus, der die nächste Verbindung bekommt. Dies wird bei sog. „pre- forked“ Servern ausgenutzt. Hier wird beim Start des Servers eine Anzahl von Prozessen erzeugt, die alle im **accept(2)** blockieren. Man verspricht sich hier eine verbesserte Antwortzeit auf Clientanfragen, da das Erstellen eines neuen Prozesses wegfällt (bzw. beim Start des Servers schon gemacht wurde).

Socket Optionen

Mit den Systemcalls **getsockopt(2)** und **setsockopt(2)** können verschiedenen Einstellungen für Sockets verändert werden:

```
#include <sys/types.h>
#include <sys/socket.h>

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);
```

Die Optionen sind in verschiedene Klassen (hier **level**) eingeteilt.Optionen für **TCP** sind in **tcp(7)** beschrieben, die generischen

Optionen für den Socket- Level unter **socket(7)**. Wichtige Optionen des Socket- Layers sind z.B.

- **SO_RCVTIMEO:** Receive timeout
- **SO_SNDTIMEO:** Send timeout
- **SO_KEEPALIVE:** Keepalive- flag

Die Optionen können verschiedenen C- Typen für die Einstellung verwenden, deshalb ist **optval** ein **void** Zeiger, und es muß die Länge des verwendeten Typs angegeben werden bzw. sie wird zurückgegeben (bei **getsockopt(2)**).

Beispiel: Socket timeouts

Um z.B. ein Timeout von 5 Sekunden beim Lesen von einem Socket **fd** einzustellen, wird **setsockopt(2)** verwendet. Die **optval** ist hier vom Typ **struct timeval**. Diese Struktur besitzt zwei Member **tv_sec** und **tv_usec** für Sekunden und Mikrosekunden. Um also den Timeout beim lesen zu setzen, verwenden wir die Option **SO_RCVTIMEO** des Levels **SOL_SOCKET**[11]. Wir legen eine **struct timeval** an, hier **tv**, und befüllen sie. Beim Aufruf von **setsockopt(2)** geben wir die Adresse und Größe von **tv** an.

```
struct timeval tv;

tv.tv_sec = 5;
tv.tv_usec = 0;

ret = setsockopt( fd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof tv );
if ( ret != 0 )                      _CSERR( dbg, ret );

ret = read(fd, ... );
if ( ret < 0 ) {
       switch ( errno ) {
       case EAGAIN:
       case EWOULDBLOCK:
              /* timeout */
              break;
       default:
              /* sonstiger fehler */
       }
}
```

Ab nun wird ein **read(2)** von **fd** den Wert **-1** zurückliefern, wenn 5 Sekunden lang nichts gelesen werden konnte. Die **errno** wir dann auf **EWOULDBLOCK** oder **EAGAIN** stehen.

Beispiel: TCP Server

Ein TCP Server führt im Allgemeinen folgende Schritte durch:

1. Socket erzeugen („listen Socket“)
2. Socket an eine Protokolladresse binden (Socket benennen)
3. Dem Betriebssystem mitteilen, daß es auf eingehende

11 siehe **socket(7)**

Verbindungen hören soll
4. Eine Verbindung aus der Queue holen (neuer Socket, „connection Socket“)
5. Mit dem Client Kommunizieren („connection Socket“)
6. „connection Socket“ schließen
7. ggf. zu Schritt (4) gehen
8. „listen Socket“ schließen

Um mehrere Clients **parallel** abarbeiten zu können, wird üblicherweise [12] die Kommunikation mit dem Client in einen eigenen Prozess oder Thread ausgelagert. In der einfachsten Form wird unter UNIX **fork(2)** verwendet, um einen neuen Prozeß für die Kommunikation mit dem Client zu erzeugen.

Der weiter untenstehende Programmcode implementiert einen soclchen einfachen TCP Server, der **fork(2)** verwendet. Dieser Servertyp lagert Schritt **5** und **6** in einen neu erzeugten Prozess aus. Prozesse werden dabei nach bedarf erzeugt.

Bei einem **preforked** Server werden die Prozesse für die Kommunikation beim Start des Servers erzeugt. Dabei wird eine bestimmte, vorgegebene Anzahl von Prozessen erzeugt, die alle dann einen **accept(2)** ausführen. Alle Prozesse blockieren, bis eine Verbindung eingeht, dann wählt das Betriebssystem zufällig einen Prozess aus, der die Verbingund „bekommt“, die andern blockieren weiter.

Funktion

Der Server erzeugt einen TCP Stream Socket und bindet ihn an eine Protokolladresse. Dabei wird die Konstante **INADDR_ANY** verwendet, um anzuzeigen, daß der Server auf allen auf dem Host definierten IP Adressen hören will. Der Port wird über die Kommandozeile gelesen. Nachdem der Server den Socket in den „listen“ Modus geschaltet hat, wartet er auf eingehende Verbindungn. Pro eingehende Verbindung wird ein neuer Prozess gestartet, der die Kommunikation mit dem Client behandelt.

Das implementierte **Protokoll** [13] dieses Servers ist einfach, er sendet alle Zeilen, die er vom Client empfängt, wieder an diesen zurück. Dies macht er so lange, bis der Client die Verbindung schließt oder ein

12 Es sind noch implementationen mit **select(2)** oder **poll(2)** möglich, die gänzlich ohne Threads oder Kindprozesse auskommen, aber trotzdem mehrere Verbindungen gleichzeitig bedienen. Dies wird „IO multiplexing“ genannt und ist nicht trivial zu implementieren, ist aber i.d.R. die Implementation, die den größten IO Durchsatz bietet.

13 Dies ein Protokoll zu nennen ist etwas fragwürdig. Immerhin arbeitet der Server **zeilenweise** , d..h. er liest immer erst eine Zeile vom Client, bevor er sie verarbeitet. Sehr viele Protokolle funktionieren so.

Fehler auftritt.

Zeilen 79- 81: Port

Hier wird, wenn vorhanden, das erste Argument des Programms in ein Integer gewandelt und als Port verwendet. Ansonsten wird der Prot 10000 verwendet.

Zeilen 83- 85: TCP Stream Socket erzeugen

Hier wird der TCP Stream Socket mit **socket()** erzeugt. Er wird „listenfd“ genannt.

Zeilen 92- 95: Socket an Protokolladresse binden

Wie bei einem TCP Client wird eine „struct sockaddr_in“ befüllt, die unsere Protokolladresse enthält. Wieder wird die Struktur zuerst mit Nullbytes gefüllt und dann erst mit unseren Werten befüllt. Auch hier wird beachtet, daß die Werte in der Struktur in **Network Byte Order** anzugeben sind, deshalb wird **htoxxxx()** verwendet.

Anstatt einer IP Adresse wird hier **INADDR_ANY** angegeben, um anzuzeigen, daß der Server an allen auf dem Host definierten IP Adressen hören will. Mit Adressfamilie, IP Adresse und Port ist die Protokolladresse komplett, der Socket kann mit **bind()** benannt werden.

Zeilen 96- 97: Zeilen Auf Verbindungen hören

Mit **listen()** wird dem Betriebssystem mitgeteilt, daß der Server nun bereit ist, Verbindungen von Clients anzunehmen. Das Betriebssystem richtet Warteschlangen für Verbindungen ein und nimmt diese entgegen.

Zeile 99- 119: Serverschleife

In dieser Endlosschleife werden Verbindungen mit **accept()** angenommen und neue Prozesse für die Verarbeitung der Verbindungen mit **fork()** erzeugt.

Zeile 101: Verbindung annehmen

Hier wird versucht, eine fertig aufgebaute Verbindung aus der Warteschlange des Betriebssystems zu holen, ggf. blockiert der Server hier, bis eine Verbindung aufgebaut wurde und zur Verfügung steht. **accept()** liefert dabei einen neuen Socket, der nun verwendet werden kann, um mit dem Client zu kommunizieren. Über die Argumente **cliaddr** und **clilen** wird die Protokolladresse des Verbindungspartners (des Clients) und deren Länge zurückgegeben. Diese Zeiger können **NULL** sein, wenn diese Information nicht verwendet wird.

Zeile 107, 108: Prozess erzeugen

Nun wird mit **fork()** ein neuer Prozess erzeugt. Der komplette Programmcode samt Daten wird kopiert[14]. Wenn **fork()** zurückkehrt, existieren zwei Prozesse, die den selben Code und die selben Daten enthalten. Über den Rückgabewert von **fork()** kann entschieden werden, ob man sich im Vater- oder im Kindprozess befindet.
Der Vaterprozess bekommt die PID ses Kindes zurück, der Kindprozess bekommt 0 (Null) zurück.
Der Vater schließt den Socket, der vom **accept()** zurückgeliefert wird. Dies beteutet nicht, daß die Verbindung geschlossen wird, denn auch der Kindprozess hat diesen Socket noch „offen“. Erst wenn der letzte Benutzer[15] des Sockets diesen schließt, wird die Verbindung wirklich geschlossen. Sofort darauf wird wieder versucht, eine Verbindung anzunehmen (siehe oben, Schleife!).

Zeilen 32- 66, 110- 118: Programmcode des Kindes

Dies ist der Teil des Servers, der nur in Kindprozessen ausgeführt wird.

Zeile 111: Listen Socket schließen

Da der Kindprozess den „listen socket“ nicht benötigt, wird er hier geschlossen.

Zeile 113: Mit dem Client Kommunizieren

Hier wird mit dem Client kommuniziert. Der Inhalt dieser Funktion implementiert das Protokoll, das der Server mit dem Client spricht.

Zeile 118: Kindprozess beenden

Mit **exit(2)** wird der Kindprozess beendet. Dies schließt implizit alle offenen Filedeskriptoren, also auch unseren Socket zum Client.

Zeilen 32- 66: Clientkommunikation, implementiertes Protokoll

Hier wird in einer Schleife vom Client zeilenweise gelesen und das Gelesene wieder an den Client geschickt. Die Schleife wird verlassen, sobald entweder nichts mehr gelesen oder nichts mehr geschrieben werden kann. Hier werden Funktionen zum Lesen und Schreiben verwendet, die **EINTR** bei **read(2)** und **write(2)** richtig behandeln (**csp_writen()** und **csp_readn()**), und die genau eine Zeile lesen (**csp_readline()**).

14 In realen Systemen wird hier ein **copy- on- write** verwendet, d.h. nur Speicherseiten, auf die geschrieben wird, werden „on- the- fly“ kopiert. Die Seiten, von denen nur gelesen wird, bleiben gleich, sie werden nur mit der MMU in den Adressraum des neuen Prozesses abgebildet („gemapt“).

15 Die Struktur im Betriebssystem besitzt Referenz Counter, die durch **close()** erniedrigt werden. Erreicht dieser Zähler 0, kann die Struktur freigegeben werden.

Übungen

1. Was passiert bei einem Server, wie er unten dargestellt ist, wenn ein Client *sehr* viele Verbindungen aufbaut? Ändern Sie den Server so, daß er nur ein vorgegebenes Maximum an Kindprozessen erzeugt!
2. Wandeln Sie den Server in einen **preforked** Server um!
3. Messen Sie die mittlere Zeit ab dem erfolgreichen Verbindungsaufbau und dem Empfang der ersten Antwort im **Client**, verwenden Sie dabei das Beispielprogramm „tcp-client“ und eine fixe Zeichenkette (z.B. 25 Bytes). Verwenden Sie zur Zeitmessung **getitimer(2)**!
4. Mischen Sie beide implementationen, d.h. schreiben Sie einen Server, der ein bestimmtes Minimum an Kindprozessen vorhält (preforked), aber **bei Bedarf** noch bis zu einem Maximum (z.B. doppelte Anzahl preforked) erzeugt!
5. Führen Sie die Messung (3) erneut mit der Implementation (4) durch. Erstellen Sie ein Histogramm über die Verteilung der Reaktionszeiten der einzelnen Implementationen. Variiren Sie die Anzahl der Clients und die Parameter (Min/Max Prozesse) der Server, bis Sie eine Änderung sehen.

```
1. /*************************************************************************
2.  * echo-server.c
3.  *
4.  * Rahmenprogramm Uebung 2 CS prog 2004
5.  * (c) Stefan Eletzhofer <stefan.eletzhofer@inquant.de>
6.  *
7.  * This code is released under the terms of the GPLv2.
8.  *
9.  * $Id: echo-server.c,v 1.8 2004/10/19 10:41:24 seletz Exp $
10. */
11.#include <stdio.h>
12.#include <string.h>
13.#include <errno.h>
14.#include <signal.h>
15.#include <sys/types.h>
16.#include <sys/wait.h>
17.#include <sys/socket.h>
18.#include <netinet/in.h>
19.#include <arpa/inet.h>
20.
21.#include <csdebug.h>
22.#include <csprog.h>
23.
24.#ifdef DEBUG
25.static int dbg = 1;
26.#else
27.static int dbg = 0;
28.#endif
29.
30.#define MAXLINE    128
31.
32.int do_echoserver( int fd )
33.{
34.    int ret = 0;
```

```
35.     char buf[MAXLINE];
36.
37.     _CSDBG( dbg, "do_echoserver( fd=%d ) {", fd );
38.
39.     while ( 1 ) {
40.             int nread;
41.             int nwritten;
42.
43.             /* read a line from client */
44.             nread = csp_readline( fd, buf, sizeof buf );
45.             if ( nread <= 0 ) {
46.                     /* hmm, cannot read from client. He
47.                      * probably closed connection.
48.                      */
49.                     break;
50.             }
51.
52.
53.             /* echo back what client said */
54.             nwritten = csp_writen( fd, buf, nread );
55.             if ( nwritten != nread ) {
56.                     /* hmm. Did not write whole data or
57.                      * write error.
58.                      */
59.                     break;
60.             }
61.     }
62.
63.DONE:
64.     _CSDBG( dbg, "%d }", ret );
65.     return ret;
66.}
67.
68.int main( int argc, char *argv[] )
69.{
70.     int ret;
71.     int listenfd, connfd;
72.     pid_t pid;
73.     struct sockaddr_in cliaddr, servaddr;
74.     short port = 10000;
75.
76.     //_CSDBGSET( "echo-server.dbg" );
77.     _CSDBG( dbg, "main() {" );
78.
79.     if ( argc>1 )
80.             port = atoi( argv[1] );
81.     _CSDBG( dbg, "port=%d", port );
82.
83.     listenfd = socket( AF_INET, SOCK_STREAM, 0 );
84.     if ( listenfd <= 0 )          _CSERR( dbg, 10 );
85.     _CSDBG( dbg, "listenfd=%d", listenfd );
86.
87.     bzero( &servaddr, sizeof servaddr );
88.     servaddr.sin_family = AF_INET;
89.     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
90.     servaddr.sin_port = htons( port );
91.
92.     ret = bind( listenfd, (struct sockaddr *)&servaddr,
93.                     sizeof servaddr );
94.     if ( ret )                    _CSERR( dbg, 11 );
95.
96.     ret = listen( listenfd, 5 );
97.     if ( ret )                    _CSERR( dbg, 12 );
98.
99.     while ( 1 ) {
100.            int clilen = sizeof cliaddr;
101.            connfd = accept( listenfd,
102.                    (struct sockaddr *)&cliaddr, &clilen );
```

```
103.            if ( connfd <= 0 )    _CSERR( dbg, 13 );
104.
105.            _CSDBG( dbg, "connection accepted, fd=%d", connfd );
106.
107.            pid = fork();
108.            if ( pid < 0 )        _CSERR( dbg, 14 );
109.            if ( pid == 0 ) {
110.                    _CSDBG( dbg, "CHILD( pid=%d) {", getpid() );
111.                    close( listenfd );
112.
113.                    do_echoserver( connfd );
114.
115.                    _CSDBG( dbg, "CHILD %d exit }", getpid() );
116.                    exit( 0 );
117.            }
118.            close( connfd );
119.    }
120.
121.    ret = 0;
122.DONE:
123.    return ret;
124.}
```

POSIX Signale

Signale unter UNIX werden verwendet, um **Ereignisse** an Prozesse zu melden. Solche Signale können vom Betriebssystem oder von anderen Prozessen gesendet werden. Diese Signale treffen **asynchron** ein, d.h. man weiß nicht genau, ob und wann ein Signal eintrifft. Signale werden auch als **Software Interrupts** bezeichnet.

Ein Signal wird durch eine **Signalnummer** spezifiziert. Diese Zahlen sind positiv und haben jeweils einen Namen. Die Signalnummern unter Linux lassen sich einfach mit dem Kommando [16] „trap -l“ auflisten:

```
seletz@irgendwo > trap -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT     7) SIGBUS      8) SIGFPE
 9) SIGKILL    10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM    15) SIGTERM    17) SIGCHLD
18) SIGCONT    19) SIGSTOP    20) SIGTSTP    21) SIGTTIN
22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO
30)SIGPWR      31) SIGSYS     35) SIGRTMIN ........
```

Unter LINUX kann auch mit „man 7 signal“ eine Liste samt Beschreibung der Signale augerufen werden.

Signale informieren uns also über ein eingetretenes Ereignis. Welches Ereignis eingetreten ist, wird durch die Signalnummer spezifiziert. So wird z.B. ein **SIGQUIT** an ein Programm geschickt, wenn es über die Konsole ein CTRL-C empfängt. Oder es wird ein **SIGSEGV** vom Betriebssystem an einen Prozess gesendet, wenn dieser einen ungültigen Speicherzugriff ausführt.

Prozesse, die an Signalen interessiert sind, müssen einen **Signalhandler** installiert haben. Ein solcher Handler ist eine Funktion, die beim Eintreffen eines Signals aufgerufen wird. Ein Handler kann auch für mehrere Signale zuständig sein. Der Handler bekommt die Signalnummer des eintreffenden Signals als Argument. Ein solcher Signalhandler wird auch als **action** bezeichnet.

Für Signale ist beim Start eines Prozesses ein **Default Handler** installiert. Dieser Default Handler definiert für die Signale unterschiedliche Verhalten, so wird z.B. beim Empfang eines **SIGQUIT** oder **SIGHUP** der Prozess beendet. Genauer ist dies unter „man 7 signal“ beschrieben.

Signale können auch **ignoriert** werden in dem man einen speziellen Default Handler installiert der leer ist (keinen Programmcode enthält).

16 Dies ist kein Systemkommando, sondern ein eingebautes Kommandoi des BASH Kommandointerpreters. Dieser wird normalerweise unter Linux verwendet, ist aber für andere Systeme verfügbar.

Die Signale **SIGKILL** und **SIGSTOP** können nicht abgefangen, blockiert oder ignoriert werden.

VORSICHT

> Signale vertragen sich i.A. **nicht** mit Threads! Bei multithreaded Prozessen ist die Signalbehandlung **nicht** definiert. Hier müssen die entsprechenden Funktionen der Posix Threads Bilbliothek verwendet werden.

POSIX Signale

POSIX definiert einige Eigenschaften im Zusammenhang mit Signalen

- Ist ein Signalhandler einmal installiert, so bleibt er installiert, auch wenn der Signalhandler schon einmal aufgerufen wurde [17]
- Während ein Signalhandler läuft, ist das Signal, für das der Handler aufgerufen wurde, blockiert.
- Signale werden nicht in Warteschlangen eingereiht oder sonstwie gespeichert. Ist ein Signal blockiert, dann gehen diese eintreffenden Signale verloren.
- Signale können jederzeit blockiert und freigegeben werden [18].

System Calls

Zur Programmierung von Signalen stehen u.A. folgende Systemcalls zur Verfügung:

Systemcall	*Beschreibung*
signal(2)	Systemcall (alt) für Signalbehandlung unter **ANSI C**. Nicht POSIX konform.
sigaction(2)	Wird verwendet, um die **action** (=Signalhandler) eines bestimmten Signals zu ändern.
sigprocmask (2)	Ändert die Maske der aktuell geblockten Signale.

Installieren eines Signalhandlers

Blockieren und freigeben von Signalen

17 Verschiedene UNIX derivate hatten hier unterschiedliches Verhalten. Signalhandler, die einmal gelaufen waren, wurden vom System automatisch deinstalliert. Hier musste im Signalhandler der Signalhandler wieder installiert werden.

18 Ausnahme: Es gibt Signale, die grundsätzlich nicht blockiert und abgefangen werden können **SIGKILL** und **SIGSTOP**