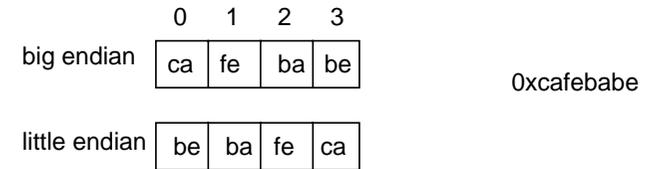


- Besprechung Aufgabe 7 (lash)
- Informationen zu Aufgabe 8 (simail)
 - Byteorder bei Netzwerkkommunikation
 - Netzwerkprogrammierung - Sockets
 - Netzwerkprogrammierung - Verschiedenes

- Wiederholung: Byteorder



- Kommunikation zwischen Rechnern verschiedener Architekturen z. B. Intel Pentium (litte endian) und Sun Sparc (big endian)
- **htons, htonl**: Wandle Host-spezifische Byteordnung in Netzwerk-Byteordnung (big endian) um (**htons** für **short int**, **htonl** für **long int**)
- **ntohs, ntohl**: Umgekehrt

U9-2 Sockets

- Endpunkte einer Kommunikationsverbindung
- Arbeitsweise: FIFO, bidirektional
- Attribute:
 - **Name** (Zuweisung eines Namens durch *Binding*)
 - **Communication Domain**
 - **Typ**
 - **Protokoll**

1 Communication Domain und Protokoll

- **Communication Domain** legt die **Protokoll-Familie**, in der die Kommunikation stattfindet, fest
- durch die Protokoll-Familie wird gleichzeitig auch die Adressierungsstruktur (**Adress-Familie**) festgelegt (war unabhängig geplant, wurde aber nie getrennt)
- das **Protokoll**-Attribut wählt das Protokoll innerhalb der Familie aus
- ursprünglich (bis BSD 4.3) existierten nur zwei Communication Domains
 - UNIX-Domain (PF_UNIX / AF_UNIX)
 - Internet-Domain (PF_INET / AF_INET)
- nur PF_INET ist generell vorhanden daneben derzeit ca. 25 Protokollfamilien definiert (ISO-Protokolle, DECnet, SNA, Appletalk, ...)

2 Internet Domain Protokoll-Familie

- ▶ Protokolle: **TCP/IP** oder **UDP/IP**
- Internet Protocol - IP
 - ▶ Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze
 - ▶ definiert Format der Dateneinheit - IP-Datagramm
 - ▶ unzuverlässige Datenübertragung
 - ▶ Routing-Konzepte (IP-Pakete über mehrere Zwischenstationen leiten)
 - ▶ IP-Adressen: 4 Byte bei IPv4 bzw. 16 Byte bei IPv6
- User Datagram Protocol - UDP
 - ▶ IP adressiert Rechner, UDP einen Dienst (siehe Port-Nummern)
 - ▶ Übertragung von Paketen (**sendto**, **recvfrom**), unzuverlässig (Fehler werden erkannt, nicht aber Datenverluste)
- Transmission Control Protocol - TCP
 - ▶ zuverlässige Verbindung (Datenstrom) zu einem Dienst (Port)

2 Internet Domain Protokoll-Familie (2)

- ▶ Namen: **IP-Adressen** und **Port-Nummern**
- Internet-Adressen (IPv4)
 - ▶ 4 Byte, Notation: **a.b.c.d** (z. B. **131.188.34.45**)
- Port-Nummern
 - ▶ bei IP definiert eine Adresse einen Rechner
 - ▶ keine Möglichkeit, einen bestimmten Benutzer oder Prozess (Dienst) anzusprechen) (vgl. Vorl. 7-95, Tor)
 - ▶ die intuitive Lösung, als Ziel einen Prozess zu nehmen, hat Nachteile
 - Prozesse werden dynamisch erzeugt und vernichtet
 - Prozesse können ersetzt werden - die *PID* ändert sich dadurch
 - Ziele sollten aufgrund ihrer Funktion (Dienst) ansprechbar sein
 - Prozesse könnten mehrere Dienste anbieten (vgl. *inetd*)
 - ▶ Lösung: **Port** als "abstrakte Adresse" für einen Dienst
 - Diensterbringer (Prozess) verbindet einen Socket mit dem Port

3 Socket-Typen

- **Stream-Sockets**
 - ◆ unterstützen bidirektionalen, zuverlässigen Datenfluss
 - ◆ gesicherte Kommunikation (gegen Verlust und Duplizierung von Daten)
 - ◆ die Reihenfolge der gesendeten Daten bleibt erhalten
 - ◆ Vergleichbar mit einer *pipe* – allerdings bidirektional (UNIX-Domain- und Internet-Domain-Sockets mit TCP/IP)
- **Datagramm-Sockets**
 - ◆ unterstützen bidirektionalen Datentransfer
 - ◆ Datentransfer unsicher (Verlust und Duplizierung möglich)
 - ◆ die Reihenfolge der ankommenden Datenpakete stimmt nicht sicher mit der der abgehenden Datenpakete überein
 - ◆ Grenzen von Datenpaketen bleiben im Gegensatz zu **Stream-Socket**-Verbindungen erhalten (Internet-Domain-Sockets mit UDP/IP)

4 Client-Server-Modell

- ★ Ein **Server** ist ein Programm, das einen **Dienst (Service)** anbietet, der über einen Kommunikationsmechanismus erreichbar ist (vgl. Vorl. 7-93, ungleichberechtigte Kommunikation)
- **Server**
 - ◆ **akzeptieren Anforderungen**, die von der Kommunikationsschnittstelle kommen
 - ◆ **führen** ihren angebotenen **Dienst aus**
 - ◆ **schicken** das **Ergebnis zurück** zum Sender der Anforderung
 - ◆ **Server** sind normalerweise als normale Benutzerprozesse realisiert
- **Client**
 - ◆ ein Programm wird ein **Client**, sobald es
 - ▶ eine **Anforderung an einen Server** schickt und
 - ▶ auf eine Antwort wartet

5 Generieren eines Sockets

- Sockets werden mit dem Systemaufruf `socket(2)` angelegt

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

- **domain**, z. B.

- ◆ **PF_INET**: Internet (PF_ = Protocol Family)
- ◆ **PF_UNIX**: Unix Filesystem

- **type** in PF_INET und PF_UNIX Domain:

- ◆ **SOCK_STREAM**: Stream-Socket (bei PF_INET = TCP-Protokoll)
- ◆ **SOCK_DGRAM**: Datagramm-Socket (bei PF_INET = UDP-Protokoll)

- **protocol**

- ◆ Default-Protokoll für Domain/Type Kombination: 0 (z.B. INET/STREAM -> TCP) (siehe `getprotobyname(3)`)

7 Namensgebung für TCP-Sockets

- Name eines TCP-Sockets durch IP-Adresse und Port-Nummer definiert

```
struct sockaddr_in {
    sa_family_t    sin_family; /* = AF_INET */
    in_port_t      sin_port;   /* Port */
    struct in_addr sin_addr;   /* Internet-Adresse */
    char           sin_zero[8]; /* Füllbytes */
};
```

- ◆ **sin_port**: Port-Nummer
 - Port-Nummern sind eindeutig für einen Rechner und ein Protokoll
 - Port-Nummern < 1024: privilegierte Ports für root (in UNIX) (z.B. www=80, Mail=25, finger=79)
 - Portnummer = 0: die Portnummer soll vom System gewählt werden
 - Portnummern sind 16 Bit, d.h. kleiner als 65536
- ◆ **sin_addr**: IP-Adresse, mit `gethostbyname(3)` zu finden
 - **INADDR_ANY**: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkkarten) Verbindungen akzeptieren soll

6 Namensgebung

- Sockets werden ohne Namen generiert

- durch den Systemaufruf `bind(2)` wird einem Socket ein Name zugeordnet

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

- ◆ **s**: socket
- ◆ **name**: Protokollspezifische Adresse
Socket-Interface (`<sys/socket.h>`) ist zunächst protokoll-unabhängig

```
struct sockaddr {
    sa_family_t    sa_family; /* Adressfamilie */
    char           sa_data[14]; /* Adresse */
};
```

im Fall von **AF_INET**: IP-Adresse / Port

- es wird konkret eine `struct sockaddr_in` übergeben

- ◆ **namelen**: Länge der konkret übergebenen Adresse in Bytes

8 Binden eines TCP-Socket — Beispiel

- Adresse und Port müssen in Netzwerk-Byteorder vorliegen!

- Beispiel

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
s = socket(PF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof(sin));
```

9 Verbindungsannahme durch Server

■ Server:

- ◆ **listen(2)** stellt ein, wie viele ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein *accept* wartend)
- ◆ **accept(2)** nimmt Verbindung an:
 - *accept* blockiert solange, bis ein Verbindungswunsch ankommt
 - es wird ein neuer Socket erzeugt und an remote Adresse + Port (Parameter *from*) gebunden
lokale Adresse + Port bleiben unverändert
 - dieser Socket wird für die Kommunikation benutzt
 - der ursprüngliche Socket kann für die Annahme weiterer Verbindungen genutzt werden

```
struct sockaddr_in from;
socklen_t fromlen;
...
listen(s, 5);          /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

11 Verbindungsaufbau und Kommunikation

■ Beispiel: Server, der alle Eingaben wieder zurückschickt

```
fd = socket(PF_INET, SOCK_STREAM, 0); /* Fehlerabfrage */
name.sin_family = AF_INET;
name.sin_port = htons(port);
name.sin_addr.s_addr = htonl(INADDR_ANY);

bind(fd, (const struct sockaddr *)&name, sizeof(name)); /* Fehlerabfrage */
listen(fd, 5); /* Fehlerabfrage */
in_fd = accept(fd, NULL, 0); /* Fehlerabfrage */

/* hier evtl. besser Kindprozess erzeugen und eigentliche
Kommunikation dort abwickeln */
for(;;) {
    n = read(in_fd, buf, sizeof(buf)); /* Fehlerabfrage */
    write(in_fd, buf, n); /* Fehlerabfrage */
}
close(in_fd);
```

10 Verbindungsaufbau durch Client

■ Client:

- ◆ **connect(2)** meldet Verbindungswunsch an Server
 - *connect* blockiert solange, bis Server Verbindung mit *accept* annimmt
 - Socket wird an die remote Adresse gebunden
 - Kommunikation erfolgt über den Socket
 - falls Socket noch nicht lokal gebunden ist, wird gleichzeitig eine lokale Bindung hergestellt (Port-Nummer wird vom System gewählt)

```
struct sockaddr_in server;
...
connect(s, (struct sockaddr *)&server, sizeof server);
```

- Eine Verbindung ist eindeutig gekennzeichnet durch
 - ◆ <lokale Adresse, Port> und <remote Adresse, Port>

12 Schließen einer Socketverbindung

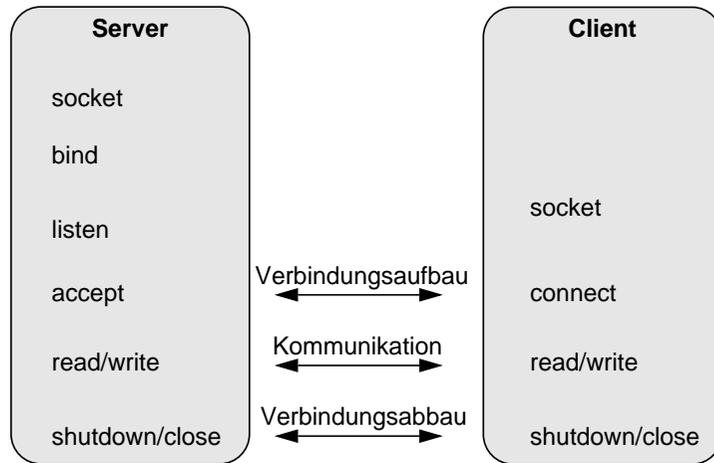
- *close(s)*
- *shutdown(s, how)*
 - ◆ how:
 - SHUT_RD: verbiete Empfang (nächstes *read* liefert EOF)
 - SHUT_WR: verbiete Senden (nächstes *write* führt zu Signal SIGPIPE)
 - SHUT_RDWR: verbiete Senden und Empfangen

13 Verbindungslose Sockets

- Für Kommunikation über Datagramm-Sockets kein Verbindungsaufbau notwendig
- Systemaufrufe

sendto(2)	Datagramm senden
recvfrom(2)	Datagramm empfangen
- **Besonderheit: *Broadcasts*** über Datagramm-Sockets (Internet Domain)

14 TCP-Sockets: Zusammenfassung



15 Sockets und UNIX-Standards

- Sockets sind nicht Bestandteil des POSIX.1-Standards
- Sockets stammen aus dem BSD-UNIX-System, sind inzwischen Bestandteil von
 - ◆ BSD (-D_BSD_SOURCE)
 - ◆ SystemV R4 (-DSVID_SOURCE)
 - ◆ UNIX 95 (-D_XOPEN_SOURCE -D_XOPEN_SOURCE_EXTENDED=1)
 - ◆ UNIX 98 (-D_XOPEN_SOURCE=500)

U9-3 Netzwerk-Programmierung - Verschiedenes

- Parametrierung eines Sockets abfragen / setzen
 - ◆ `getsockopt(2)`, `setsockopt(2)`
- Informationen über Socket-Bindung
 - ◆ `getpeername(2)`
Namen der mit dem Socket verbundenen Gegenstelle abfragen
 - ◆ `getsockname(2)`
Namen eines Sockets abfragen
- Hostnamen und -adressen ermitteln
 - ◆ `gethostbyname(3)`

1 getsockname, getpeername

```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int s, struct sockaddr *addr, socklen_t *addrlen);
```

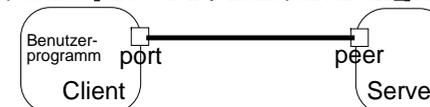
- Informationen über die lokale Adresse des Socket

```
struct sockaddr_in server;
socklen_t len;

len = sizeof(server);
getsockname(sock, (struct sockaddr *) &server, &len);
printf("Socket port %#d\n", ntohs(server.sin_port));
```
- Informationen über die remote Adresse des Socket

```
struct sockaddr_in server;
socklen_t len;

len = sizeof(server);
getpeername(sock, (struct sockaddr *) &server, &len);
printf("Socket port %#d\n", ntohs(server.sin_port));
```



2 Hostnamen und Adressen

- `gethostbyname` liefert Informationen über einen Host

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
    char    *h_name;        /* offizieller Rechnername */
    char    **h_aliases;   /* alternative Namen */
    int     h_addrtype;    /* = AF_INET */
    int     h_length;     /* Länge einer Adresse */
    char    **h_addr_list; /* Liste von Netzwerk-Adressen,
                           abgeschlossen durch NULL */
};
```

- `gethostbyaddr` sucht Host-Informationen für bestimmte Adresse

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
```

3 Socket-Adresse aus Hostnamen erzeugen

```
char *hostname = "fau107a";
struct hostent *host;
struct sockaddr_in saddr;

host = gethostbyname(hostname);
if(!host) {
    perror("gethostbyname()");
    exit(EXIT_FAILURE);
}
memset(&saddr, 0, sizeof(saddr)); /* Struktur initialisieren */
memcpy(&saddr.sin_addr, host->h_addr, host->h_length);
saddr.sin_family = AF_INET;
saddr.sin_port = htons(port);

/* saddr verwenden ... z.B. bind oder connect */
```