U1 Interprozesskommunikation mit Sockets

- Organisatorisches
- Client-Server-Modell
- Kommunikation innerhalb eines Systems
- Kommunikation über Systemgrenzen hinweg
- Betriebssystemschnittstelle zur IPC
- POSIX-I/O vs. Standard-C-I/O

U1-1 Organisatorisches

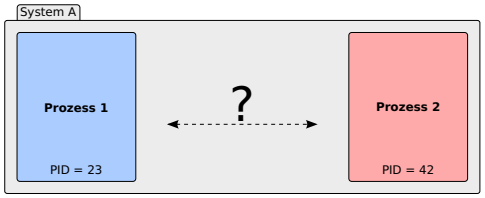
- Rechnerübungen von Systemprogrammierung 1 und 2 finden gleichzeitig statt (Vgl. Univis "Rechnerübungen zu Systemprogrammierung 1&2")
- In der Woche vom 16. - 20. Mai finden keine SP2-Tafelübung statt
- Projektverzeichnisse in diesem Semester unter `/proj/i4sp2`

U1-2 Client-Server-Modell

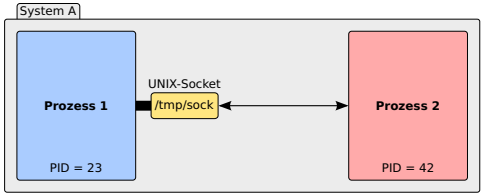
★ Ein ***Server*** ist ein Programm, das einen **Dienst** *(Service)* anbietet, der über einen Kommunikationsmechanismus erreichbar ist (vergleiche Vorlesung *B VI Prozesse*, Seite 30, ungleichberechtigte Kommunikation)

- 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

U1-3 Kommunikation innerhalb eines Systems

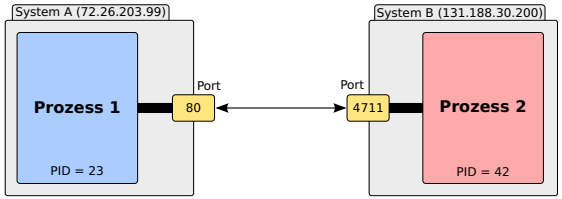


- Intuitiv: Auffinden des Kommunikationspartners über dessen Prozess-ID
  - Prozesse werden allerdings dynamisch erzeugt und vernichtet; PID ändert sich
- Besser: Verwendung einer abstrakten "Adresse" (hier: UNIX-Socket)



  - Prozess 1 ist so über speziellen Eintrag im Dateisystem ansprechbar

U1-4 Kommunikation über Systemgrenzen hinweg



- Mittels IP-Adresse und Port-Nummer ist Prozess (eindeutig) identifizierbar
- Auffinden eines Kommunikationspartners hier zweistufig
  - zunächst Auffinden des Systems über Adresse (hier: IP-Adresse)
  - danach Auffinden des Prozesses über abstrakte "Adresse" (hier: Port)
- Zusätzliche Abstraktion der IP-Adresse mit Hilfe des DNS-Protokolls
  - Zuordnung von IP-Adressen zu Rechnernamen
  - Beispielsweise: faui06.informatik.uni-erlangen.de -> 131.188.30.200

1 Adressierung des Systems

- Verwendetes Protokoll: Internet Protocol (IP)
  - Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze
  - definiert Format der Datengrundeinheit - IP-Datagramm
  - unzuverlässige Datenübertragung
  - Routing-Konzepte (IP-Pakete über mehrere Zwischenstationen leiten)
  - Adressen bei IPv4: 4 Byte
    - Notation: 4 mit '.' getrennte Byte-Werte in Dezimaldarstellung
    - z. B. **`131.188.30.200`**
  - Adressen bei IPv6: 16 Byte
    - Notation: acht mit ':' getrennte 2-Byte-Werte in Hexadezimaldarstellung
    - z.B.: **`2001:638:a00:1e:219:99ff:fe33:8e75`**
    - in der Adresse kann einmalig '::' als Kurzschreibweise einer Nullfolge verwendet werden
    - Beispiel: IPv6 localhost-Adresse: **`0:0:0:0:0:0:0:1 = ::1`**

1 Adressierung des Systems

- Transparente IPv4-in-IPv6-Unterstützung
  - Spezieller Adressbereich ::ffff:0:0/96 zur Abbildung von IPv4 auf IPv6
  - z.B. 131.188.30.200 auf ::ffff:83bc:1ec8 (auch ::ffff:131.188.30.200)
  - Binden eines PF_INET6-Sockets bindet standardmäßig auch den entsprechenden IPv4-Port
    - dem Prozess erscheinen eingehende IPv4-Verbindungen als IPv6-Verbindungen aus diesem Adressbereich
  - ausgehende IPv6-Verbindungen an diesen Adressbereich werden auf entsprechende IPv4-Verbindungen abgebildet

- Anmerkung zu IPv6:
  - Einführung von IPv6 schleppend
  - 1998 verabschiedet, Verbreitung immer noch sehr gering
  - Am 3. Februar 2011 wurden die letzten verfügbaren IPv4-Adressen durch die IANA (Internet Assigned Numbers Authority) vergeben

2 Adressierung des Prozesses

- über Portnummern
  - Portnummern sind 16 Bit, d.h. kleiner als 65536
  - Port-Nummern < 1024: privilegierte Ports für root (in UNIX)
    (z.B. www=80, Mail=25, finger=79)

- Eingesetzte Protokolle
  - User Datagram Protocol - UDP (Transportschicht)
    - 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 (Transportschicht)
    - zuverlässige Verbindung zu einem Dienst (Port)

3 Arten des Datenaustausches

- Datenstromorientiert (Stream-Sockets)
  - ◆ unterstützen bidirektionalen, zuverlässigen Datenaustausch
  - ◆ gesicherte Kommunikation (gegen Verlust und Duplizierung von Daten)
  - ◆ die Reihenfolge der gesendeten Daten bleibt erhalten
  - ◆ Vergleichbar mit einer *pipe* – allerdings bidirektional
- Paketorientiert (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

4 Format des Datenaustausches

- Beim Austausch von Binärdaten ist die Reihenfolge der einzelnen Bytes zur richtigen Interpretation wichtig
- Kommunikation zwischen Rechnern verschiedener Architekturen z. B. Intel Pentium (litte endian) und Sun Sparc (big endian) setzt Vereinheitlichung der Netzwerkbyteorder voraus
- Netzwerk-Byteorder ist auf big endian festgelegt
- Exkurs: Byteorder

Wert	Repräsentation	0	1	2	3
0xcafebabe	big endian	ca	fe	ba	be
	little endian	be	ba	fe	ca

U1-5 Betriebssystemschnittstelle zur IPC

1 Sockets

- Generische Abstraktion zur Interprozesskommunikation
  - Verwendung im Quellcode unabhängig von Kommunikations-Domäne
  - Betriebssystemseite Implementierung abhängig von jeweiliger Kommunikations-Domäne

2 Erzeugen und Schließen 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_UNIX`** (UNIX-Domäne), **`PF_INET`** (IPv4-Domäne), **`PF_INET6`** (IPv6)
  - **`type`** innerhalb der Domain:
    - **`SOCK_STREAM`**: Stream-Socket (bei PF_INET(6) = TCP-Protokoll)
    - **`SOCK_DGRAM`**: Datagramm-Socket (bei PF_INET(6) = UDP-Protokoll)
  - **`protocol`**
    - Standard-Protokoll für Domain/Type Kombination: **`0`**

- Schließen des Sockets und Freigeben der "enthaltenen" Ressourcen

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

3 Binden an einen Namen - Allgemein

- Sockets werden ohne Namen generiert
- Systemaufruf ***bind(2)*** stellt eine generische Schnittstelle zum Binden von Sockets in unterschiedlichen Domänen bereit
- der Systemaufruf ***bind(2)*** bindet einen Namen an einen Socket

```
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 */
};
```

  - **namelen**: Länge der konkret übergebenen Struktur in Bytes

4 Namensgebung für IPv4-Sockets

- Name 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
  - **sin_addr:** IP-Adresse
    - **INADDR_ANY**: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkinterfaces) Verbindungen akzeptieren soll

- **INADDR_**-Werte liegen in Hostbyteorder vor
  - Umwandlung mittels **htons**, **htonl**: Wandle Host-spezifische Byteordnung in Netzwerk-Byteordnung (big endian) um
    (**htons** für **short int**, **htonl** für **long int**)

5 Namensgebung für IPv6-Sockets

- Name durch IP-Adresse und Port-Nummer definiert

```
struct sockaddr_in6 {
   uint16_t           sin6_family;   /* = AF_INET6 */
   uint16_t           sin6_port;     /* Port */
   uint32_t           sin6_flowinfo;
   struct in6_addr    sin6_addr;     /* IPv6-Adresse */
   uint32_t           sin6_scope_id;
};
struct in6_addr {
   unsigned char      s6_addr[16];
};
```

  - **sin6_addr:** IPv6-Adresse
    - ➤ **`in6addr_any / IN6ADDR_ANY_INIT`**:
      auf allen lokalen Adressen Verbindungen akzeptieren

- Die **`IN6ADDR_`**-Werte liegen bereits in Netzwerk-Byteorder vor

6 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

6 Verbindungsannahme durch Server - Beispiel

- Beispiel: Server, der alle Eingaben wieder zurückschickt (ohne Fehlerbehandlungen)

```
fd = socket(PF_INET6, SOCK_STREAM, 0);

memset(&name, 0, sizeof(name));
name.sin6_family = AF_INET6;
name.sin6_port = htons(port);
name.sin6_addr = in6addr_any;

bind(fd, (const struct sockaddr *)&name, sizeof(name));

listen(fd, 5);

in_fd = accept(fd, NULL, NULL);

/* hier evtl. besser Kindprozess erzeugen und
   eigentliche Kommunikation dort abwickeln */
for(;;) {

   n = read(in_fd, buf, sizeof(buf));

   write(in_fd, buf, n);

}

close(in_fd);
close(fd);
```


7 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)

8 Das Domain-Name-System (DNS)

- Zum Ermitteln der Werte für die **sockaddr**-Struktur kann das DNS-Protokoll verwendet werden
- **getaddrinfo** liefert nötige Werte

```
int getaddrinfo(const char *node, const char *service,
   const struct addrinfo *hints, struct addrinfo **res);
```

  - **node** gibt den DNS-Namen des Hosts an (oder IP-Adresse als String)
  - **service** gibt entweder numerischen Port als String (z.B. "25") oder den Dienstnamen (z.B. "smtp", **getservbyname(3)**) an
  - Mit **hints** kann die Adressauswahl eingeschränkt werden (z.B. auf IPv4-Sockets). Nicht verwendete Felder auf **0** bzw. **NULL** setzen.
  - Ergebnis ist eine verkettete Liste von Socket-Namen; ein Zeiger auf das Kopfelement wird in ***res** gespeichert
  - Fehlerbehandlung siehe **getaddrinfo(3)**
- Freigabe der Ergebnisliste nach Verwendung mit **freeaddrinfo(3)**

8 Das Domain-Name-System (DNS)

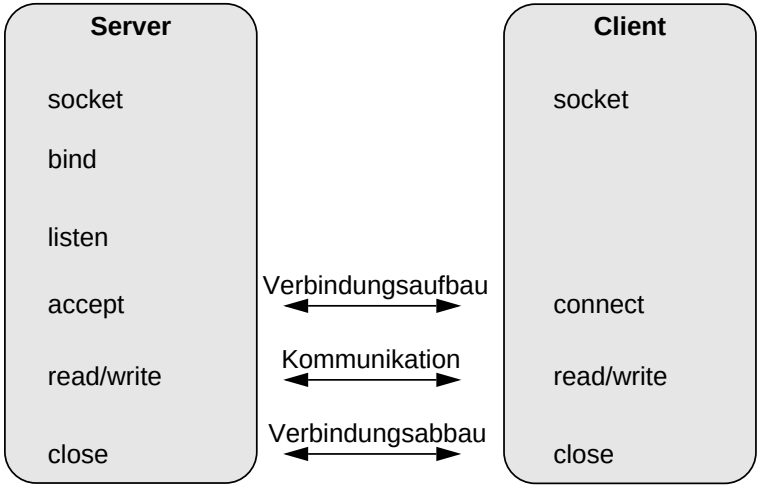
```
struct addrinfo {
   int              ai_flags;    // flags zur Auswahl (hints)
   int              ai_family;   // z.B. PF_INET6
   int              ai_socktype; // z.B. SOCK_STREAM
   int              ai_protocol; // Protokollnummer
   size_t           ai_addrlen;  // Größe von ai_addr
   struct sockaddr *ai_addr;     // Adresse f. bind/connect
   char            *ai_canonname;// offizieller Hostname
   struct addrinfo *ai_next;     // nächste Addresse oder NULL
};
```

- **ai_flags** relevant zur Anfrage von Auswahlkriterien (**hints**)
  - **AI_ADDRCONFIG:** Auswahl von Adresstypen, für die auch ein lokales Interface existiert (z.B. werden keine IPv6-Adressen geliefert, wenn der aktuelle Rechner gar keine IPv6-Adresse hat)
- **ai_family, ai_socktype, ai_protocol** für **socket(2)** verwendbar
- **ai_addr, ai_addrlen** für **bind(2)** und **connect(2)** verwendbar

8 Das Domain-Name-System (DNS) - Beispiel

```
char *hostname = "lists.informatik.uni-erlangen.de";
int gai_ret, sock;
struct addrinfo *sa_head, *sa, hints;

memset(&hints,0,sizeof(hints));
hints.ai_socktype = SOCK_STREAM; /* nur TCP-Sockets */
hints.ai_family = PF_UNSPEC;     /* beliebige Protokollfamilie */
hints.ai_flags = AI_ADDRCONFIG;  /* nur lokal verf. Addresstypen */

gai_ret = getaddrinfo(hostname, "25", &hints, &sa_head);
if(gai_ret != 0 ) { /* Fehlerbehandlung s. Manpage */ }

/* Liste der Adressen durchtesten */
for(sa = sa_head; sa!=NULL; sa=sa->ai_next) {
    sock= socket(sa->ai_family,sa->ai_socktype,sa->ai_protocol);
    if(0 == connect(sock, sa->ai_addr, sa->ai_addrlen)) {
        break;
    }
    close(sock);
}
if(sa == NULL) { /* Fehler */ }

freeaddrinfo(sa_head);
```


9 TCP-Sockets: Zusammenfassung



U1-6 POSIX-I/O vs. Standard-C-I/O

- Für Ein- und Ausgabe stehen verschiedene Funktionen zur Verfügung

Ebene	Variante	Ein-/Augabedaten	Funktionen
2	blockorientiert	Puffer + Länge	read(), write()
3	blockorientiert	Array, Elementgröße, Anzahl	fread(), fwrite()
3	zeichenorientiert	Einzelbyte	getc(), putc()
3	zeilenorientiert	null-terminierter String	fgets(), fputs()
3	formatiert	Formatstring + beliebige Variablen	fscanf(), fprintf()

  - Ebene 2: POSIX-Systemaufrufe
    - Arbeiten mit Filedeskriptoren (`int`)
  - Ebene 3: Bibliotheksfunktionen
    - Greifen intern auf die Systemaufrufe zurück
    - Wesentlich flexibler einsetzbar
    - Arbeiten mit File-Pointern (`FILE *`)

- Auf Grund ihrer Flexibilität eignen sich `FILE*` für String-basierte Ein- und Ausgabe wesentlich besser.

U1-6 POSIX-I/O vs. Standard-C-I/O

- Konvertierung von Filedeskriptor nach Filepointer

```
FILE *fdopen(int fd, const char *type);
```

  - type kann sein "r", "w", "a", "r+", "w+", "a+"
    (**`fd`** muss entsprechend geöffnet sein!)
    - Sockets sollten mit "a+" geöffnet werden

- Schließen des erzeugten Filepointers mittels `fclose(3)`

```
int fclose(FILE *stream);
```

  - Darunterliegender Filedeskriptor wird dabei geschlossen
  - Erneutes `close(2)` nicht notwendig