

U1 Interprozesskommunikation mit Sockets

- Organisatorisches
- IPC-Grundlagen
- Adressierung in IP-Netzwerken
- 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 und 2")
- Nächste SP2-Tafelübungen ab Donnerstag, 10.05.2012
 - ◆ Terminplan: http://www4.cs.fau.de/Lehre/SS12/V_SP2/Uebung/fohlen.shtml
- Projektverzeichnisse in diesem Semester unter `/proj/i4sp2`
- SP1-Abgaben weiterhin in eurem Repository aus dem letzten Semester verfügbar
 - ◆ <https://www4.informatik.uni-erlangen.de/i4sp/ws11/sp1/<login>>

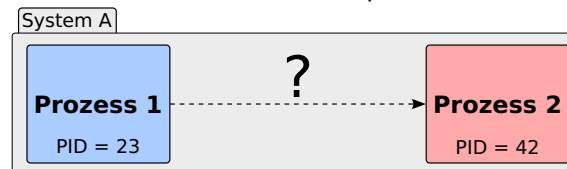
U1-2 IPC-Grundlagen

1 Client-Server-Modell

- ★ Ein **Server** ist ein Programm, das einen **Dienst** (*Service*) anbietet, der über einen Kommunikationsmechanismus erreichbar ist (vgl. Vorlesung B VI-2, Seite 30, ungleichberechtigte Kommunikation)
- Server
 - ◆ **akzeptieren Anforderungen**, die von außen 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

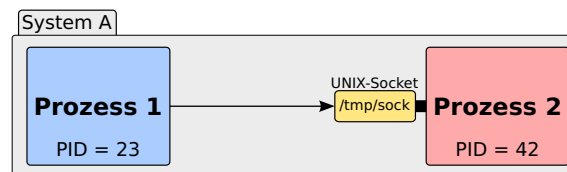
2 Kommunikation innerhalb eines Systems

- Intuitiv: Auffinden des Kommunikationspartners über dessen Prozess-ID



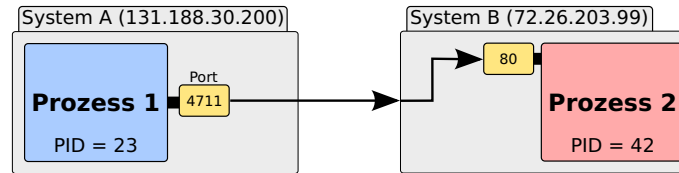
- ◆ Prozesse werden allerdings dynamisch erzeugt und vernichtet; PID ändert sich

- Besser: Verwendung eines abstrakten "Namens" (Beispiel: UNIX-Socket)



- ◆ Prozess 2 ist so über speziellen Eintrag im Dateisystem erreichbar

3 Kommunikation über Systemgrenzen hinweg



- Auffinden eines Kommunikationspartners über zweistufigen Socket-"Namen" (Beispiel: Internet Protocol - dazu gleich mehr):
 - ◆ Zunächst Auffinden des Systems (*hier: per IP-Adresse*)
 - ◆ Danach Auffinden des Prozesses im System (*hier: per Port-Nummer*)
 - ◆ Mittels IP-Adresse und Port-Nummer ist der Prozess eindeutig identifizierbar

4 Adressierung in IP-Netzwerken

- Adressierung des Systems mittels Internet Protocol (IP)
 - ◆ Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze (Routing)
 - ◆ unzuverlässige Datenübertragung
 - ◆ Adressierung bei IPv4: 4 Bytes
 - Notation: 4 mit '.' getrennte Byte-Werte in Dezimaldarstellung
 - z. B. **131.188.30.200**
 - ◆ Adressierung bei IPv6: 16 Bytes
 - 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**

4 Adressierung in IP-Netzwerken

■ 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)
- ◆ Bei Verwendung von IPv6 besteht automatisch die Möglichkeit, IPv4-Verbindungen aufzubauen/anzunehmen
 - 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 03. Februar 2011 wurden die letzten verfügbaren IPv4-Adressen durch die IANA (Internet Assigned Numbers Authority) vergeben

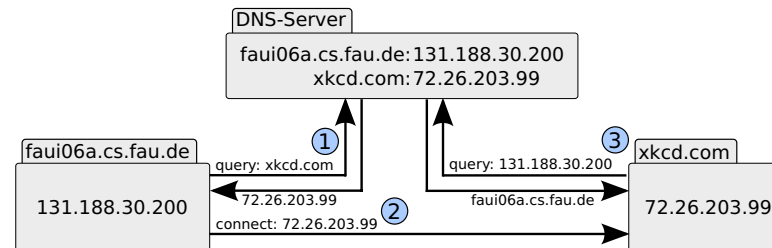
4 Adressierung in IP-Netzwerken

■ Adressierung des Prozesses mittels Portnummern:

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

5 Zusätzliche Abstraktion: Rechnername statt IP-Adresse

- ... mit Hilfe des DNS-Protokolls



- ◆ Nachschlagen der IP-Adresse für einen bestimmten Rechnernamen ("forward DNS lookup") und umgekehrt ("reverse DNS lookup")
- ◆ Schritt 3 ist optional - nur nötig, wenn der Server wissen will, wie sein Gesprächspartner heißt

6 Kommunikation: Arten des Datenaustausches

- Datenstromorientiert:
 - ◆ Gesicherte Kommunikation (gegen Verlust und Duplizierung von Daten)
 - ◆ Die Reihenfolge der gesendeten Daten bleibt erhalten
 - ◆ Vergleichbar mit einer Pipe – allerdings bidirektional
 - ◆ Implementierung: Transmission Control Protocol (TCP)
- Paketorientiert:
 - ◆ Datentransfer unsicher (Verlust und Duplizierung möglich)
 - ◆ Empfänger kann Datenpakete möglicherweise in falscher Reihenfolge erhalten
 - ◆ Grenzen von Datenpaketen bleiben im Gegensatz zu datenstromorientierten Verbindungen erhalten
 - ◆ Implementierung: User Datagram Protocol (UDP)
 - Übertragung von Paketen (`sendto`, `recvfrom`), unzuverlässig (Fehler werden erkannt, nicht aber Datenverluste)

6 Vorsicht Fallstrick!

- Beim Austausch von binären Datenwörtern ist die Reihenfolge der einzelnen Bytes zur richtigen Interpretation wichtig
- Kommunikation zwischen Rechnern verschiedener Architekturen
- z. B. Intel x86 (*little endian*) und Sun SPARC (*big endian*) - setzt einen Konsens über die verwendete Byteorder voraus
- Definierter Standard: Netzwerk-Byteorder ist auf *big endian* festgelegt
- Beispiel:

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

U1-3 Betriebssystemschnittstelle zur IPC

1 Sockets

- Generische Abstraktion zur Interprozesskommunikation:
 - ◆ Verwendung im Programm ist unabhängig von der Kommunikations-Domäne
 - ... egal, ob der Kommunikationspartner ein Prozess auf dem selben Rechner ist oder ob er tausende von Kilometern entfernt ist
 - ◆ Betriebssystemseitige Implementierung ist abhängig von der jeweiligen Kommunikations-Domäne:
 - Innerhalb des selben Systems: z. B. UNIX-Socket - Adressierung über Dateinamen, Kommunikation über gemeinsamen Speicher, keine Sicherungsmechanismen notwendig
 - Über Rechnergrenzen hinweg: z. B. TCP/UDP-Socket - Adressierung über IP-Adresse + Port, nachrichtenbasierte Kommunikation, Sicherungsmechanismen bei TCP

2 Erzeugen 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**

- Ergebnis ist ein numerischer Socket-Deskriptor
 - ◆ Entspricht einem Datei-Deskriptor und unterstützt (bei Stream-Sockets) die selben Operationen

2 Exkurs: Datei-Deskriptoren

- Philosophie von Unix: "Alles, was irgendwie Ein-/Ausgabe betreibt, ist eine Datei"
- Einen Datei-Deskriptor erhält man je nach Gerät auf anderem Weg:
 - ◆ Datei: **open(2)**
 - ◆ Serielle Schnittstelle: **open(2)** auf Pseudo-Datei **/dev/ttyS{0,1,...}**
 - ◆ Netzwerksocket: **socket(2)**
 - ◆ ...
- Danach Benutzung über eine einheitliche Systemaufruf-Schnittstelle:
 - ◆ Lesen: **read(2)**
 - ◆ Schreiben: **write(2)**
 - ◆ Schließen (**wichtig!**): **close(2)**
 - ◆ ...

3 Binden eines Sockets an einen Namen - allgemein

- Ein neu erzeugter Socket ist zunächst namenlos und somit nutzlos
- Erst nach dem "Binden" an einen Namen kann er verwendet werden
- Der Systemaufruf **bind(2)** stellt eine generische Schnittstelle zum Binden von Sockets in unterschiedlichen Domänen bereit

```
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

- **sin_port** und **sin_addr** müssen in Netzwerk-Byteorder vorliegen!

- ◆ 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 Verbindungsaufbau durch Client

- **connect(2)** meldet Verbindungswunsch an Server

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

- ◆ **sockfd**: Socket über den die Kommunikation erfolgen soll
- ◆ **addr**: Beeinhaltet abstrakte "Adresse" (bei uns: IP-Adresse und Port) des Servers
- ◆ **addrlen**: Länge der **addr**-Struktur

- **connect** blockiert solange, bis Server Verbindung annimmt
- Falls Socket noch nicht lokal gebunden ist, wird gleichzeitig eine lokale Bindung hergestellt (Port-Nummer wird vom System gewählt)
- Socket wird an die remote Adresse gebunden

7 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)`

7 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 Adresse 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

7 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. Adresstypen

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);
```

U1-4 POSIX-I/O vs. Standard-C-I/O

- Für Ein- und Ausgabe stehen verschiedene Funktionen zur Verfügung

Ebene	Variante	Ein-/Ausgabedaten	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(), printf()

- ◆ 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.

U1-4 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