Übungen zu Systemprogrammierung 2 (SP2)

Ü1 – Interprozesskommunikation mit Sockets

Christoph Erhardt, Jens Schedel, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 22. bis 26. April 2013

http://www4.cs.fau.de/Lehre/SS13/V_SP2

Agenda

Übungswoche

- Üblicherweise abwechselnd Tafelübung und Besprechung
- Nächste Übungswoche beginnt am Donnerstag

	Mon, 15.04	Tue, 16.04	Wed, 17.04	Thu, 18.04	Fri, 19.04	Sat, 20.04	Sun, 21.04	Mon, 22.04	Tue, 23.04	Wed, 24.04	Thu, 25.04	Fri, 26.04	Sat, 27.04
Vorlesung	V1							V2		M			
Übung								U1	U1	U1	U1	U1	
Mo								A1	A1	A1	A1	A1	
Di									A1	A1	A1	A1	
Mi										A1	A1	A1	
Do											A1	A1	
Fr												A1	
Sun, 28.04	**Mon, 29.04**	**Tue, 30.04**	**Wed, 01.05**	**Thu, 02.05**	**Fri, 03.05**	**Sat, 04.05**	**Sun, 05.05**	**Mon, 06.05**	**Tue, 07.05**	**Wed, 08.05**	**Thu, 09.05**	**Fri, 10.05**	**Sat, 11.05**
Vorlesung	V3							V4					
Übung				U2	U2			U2	U2	U2			
Mo	A1	A1		A1	A1			A1	A2	A2		A2	
								A2					
Di	A1	A1		A1	A1			A1	A1	A2		A2	
									A2				
Mi	A1	A1		A1	A1			A1	A1	A1		A2	
										A2			
Do	A1	A1		A1	A1			A1	A1	A1		A1	
												A2	
Fr	A1	A1		A1	A1			A1	A1	A1		A1	

Quelle: `http://www4.cs.fau.de/Lehre/SS13/V_SP2/semesterplan.shtml`

Organisatorisches

Übungsbetrieb

- Rechnerübungen von SP1 und SP2 finden gemeinsam statt
  - Termine siehe `http://www4.cs.fau.de/Lehre/SS13/V_SP2/Uebung/univis.ushtml`

SVN-Repository

- URL: `https://www4.cs.fau.de/i4sp/ss13/sp2/<login>`
  - Passwort setzen: `/proj/i4sp2/bin/change-password`
- Projektverzeichnisse in diesem Semester unter `/proj/i4sp2`
- SP1-Repository aus dem Vorsemester weiterhin zugreifbar

Änderungen zu SP1

- Durchgängige Verwendung von 64-Bit
- Betriebssystemstandard SUSv4

Kompilieren & Linken

- Standard-Flags in SP: `-std=c99 -pedantic -Wall -Werror \ -D_XOPEN_SOURCE=700`
  - Zum Debuggen zusätzlich `-g`
- In SP2 konsequente Benutzung von Makefiles

Agenda

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

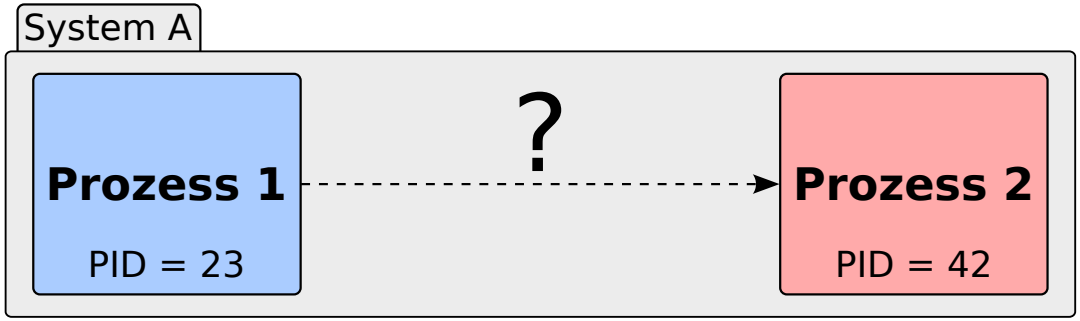
- **Akzeptiert Anforderungen**, die von außen kommen
- **Führt** einen angebotenen **Dienst aus**
- **Schickt** das **Ergebnis zurück** zum Sender der Anforderung
- Ist in der Regel als normaler Benutzerprozess realisiert

Client

- Schickt eine **Anforderung an einen Server**
- Wartet auf eine Antwort

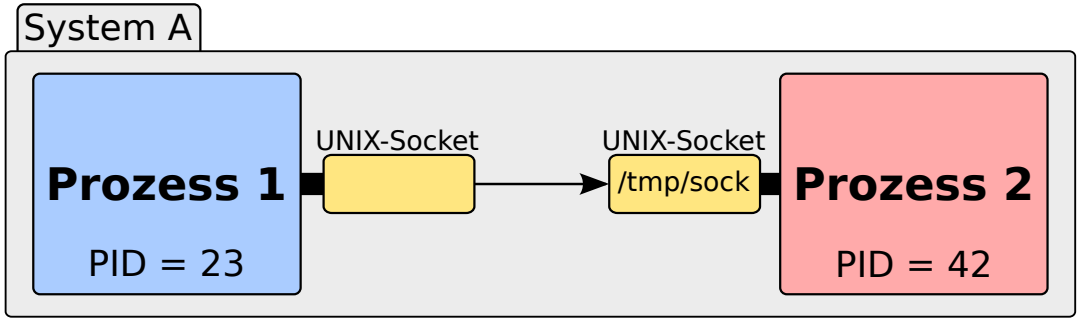
? **Wie findet man seinen gewünschten Kommunikationspartner?**

- Intuitiv: über dessen Prozess-ID



  - **Problem:** Prozesse werden dynamisch erzeugt/beendet; PID ändert sich

- **Lösung:** Verwendung eines abstrakten „Namens“

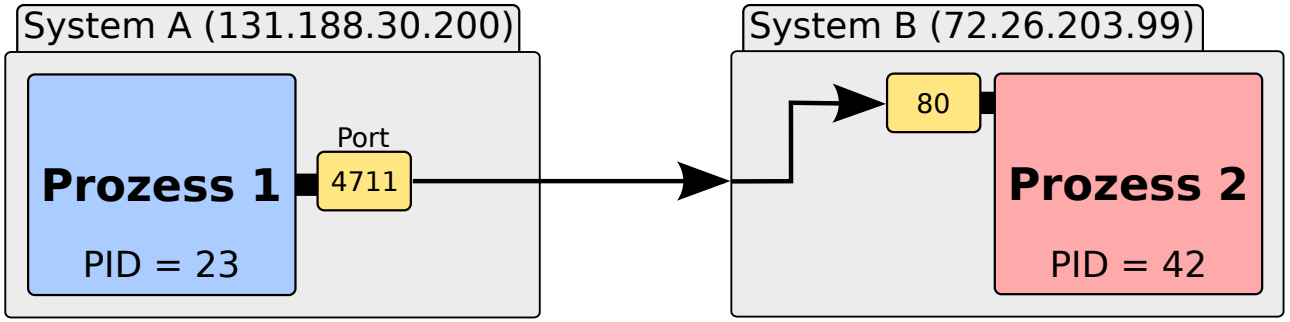


  - Prozess 2 ist so über einen speziellen Eintrag im Dateisystem erreichbar

Kommunikation über Systemgrenzen hinweg

? Wie findet man nun seinen gewünschten Kommunikationspartner?

- Wieder über einen Socket...
- ... diesmal aber mit zweistufig aufgebautem „Namen“:
  1. Identifikation des Systems innerhalb des Netzwerks
  2. Identifikation des Prozesses innerhalb des Systems
- Beispiel TCP/IP: eindeutige Kombination aus
  1. IP-Adresse
  2. Port-Nummer



Adressierung in IP-Netzwerken

Internet Protocol

- Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze (Routing)
- Unzuverlässige Datenübertragung
  - Für Zuverlässigkeit ist darüberliegende Transportschicht zuständig

IPv4

- 32-Bit-Adressraum ($\approx$ 4 Milliarden Adressen)
- Notation: 4 mit `.` getrennte Byte-Werte in Dezimaldarstellung
  - z. B. `131.188.30.200`
- Nicht zukunftsfähig wegen des zu kleinen Adressraums
  - Alle Adressen in Asien/Pazifikraum (seit April 2011) und Europa/Nahost (seit September 2012) bereits vergeben!

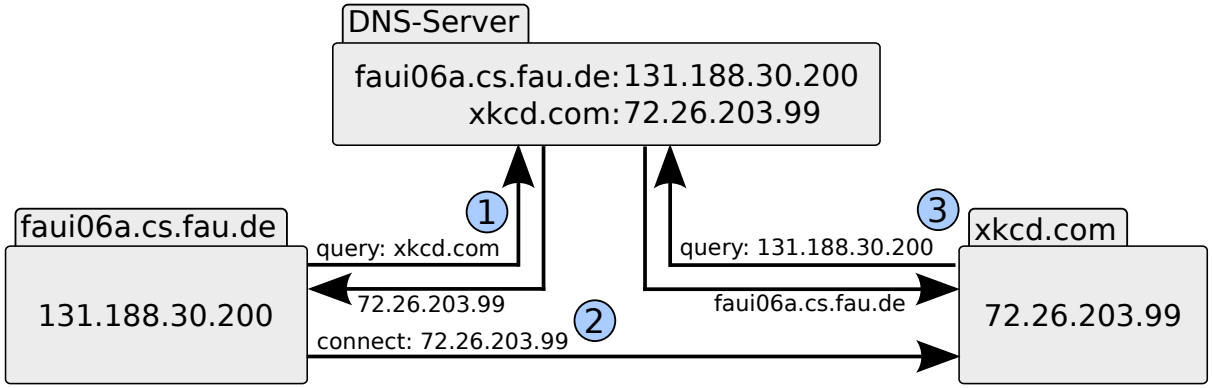
Adressierung in IP-Netzwerken

IPv6

- 128-Bit-Adressraum ($\approx 3{,}4 \cdot 10^{38}$ Adressen)
- Notation: 8 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
  - z. B. *localhost*-Adresse: `0:0:0:0:0:0:0:1 = ::1`
- Abwärtskompatibilität durch transparente IPv4-in-IPv6-Unterstützung
- Bereits 1998 als Standard verabschiedet, aber seither nur schleppende Einführung

Domain Name System (DNS)

- IP-Adressen sind nicht leicht zu merken
- ... und ändern sich, wenn man einen Rechner in ein anderes Rechenzentrum umzieht
- **Lösung:** zusätzliche Abstraktion durchs DNS-Protokoll



1. *Forward lookup*: Rechnername → IP-Adresse
2. Kommunikationsaufbau
3. *Reverse lookup* (im Beispiel optional): IP-Adresse → Rechnername

Portnummer

- Zur Identifikation eines Prozesses innerhalb eines Systems
- 16-Bit-Zahl, d. h. kleiner als 65536
- Portnummern < 1024: *well-known ports*
  - Können nur von Prozessen gebunden werden, die mit speziellen Privilegien gestartet wurden (Ausführung als *root*)
  - z. B. ssh = 22, smtp = 25, http = 80

Kommunikationsarten

Verbindungsorientiert (Datenstrom)

- Gesichert gegen Verlust und Duplizierung von Daten
- Reihenfolge der gesendeten Daten bleibt erhalten
- Vergleichbar mit einer Pipe – allerdings bidirektional
- Implementierung: Transmission Control Protocol (TCP)

Paketorientiert

- Schutz vor Bitfehlern – nicht vor Paketverlust oder -duplizierung
- Datenpakete können eventuell in falscher Reihenfolge ankommen
- Grenzen von Datenpaketen bleiben erhalten
- Implementierung: User Datagram Protocol (UDP)

Vorsicht Fallstrick!

- Beim Austausch von binären Datenwörtern ist die Reihenfolge der Einzelbytes zur richtigen Interpretation relevant
- Kommunikation zwischen Rechnern verschiedener Architekturen – z. B. x86 (*little endian*) und SPARC (*big endian*) – setzt Konsens über die verwendete Byteorder voraus
- Beispiel:

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

- **Definierter Standard:** Netzwerk-Byteorder = *big endian*

Agenda

Sockets

- Generischer Mechanismus 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
    $\rightarrow$ Adressierung über Dateinamen, Kommunikation über gemeinsamen Speicher, keine Sicherungsmechanismen notwendig
  - Über Rechnergrenzen hinweg: z. B. TCP/UDP-Socket
    $\rightarrow$ Adressierung über IP-Adresse + Port, nachrichtenbasierte Kommunikation, Sicherungsmechanismen bei TCP

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-Domäne
  - `type` innerhalb der gewählten Domäne:
    - `SOCK_STREAM`: Stream-Socket
    - `SOCK_DGRAM`: Datagramm-Socket
  - `protocol`:
    - `0`: Standard-Protokoll für gewählte Kombination (z. B. TCP/IP bei `PF_INET(6)` + `SOCK_STREAM`)

- Ergebnis ist ein numerischer Socket-Deskriptor
  - Entspricht einem Datei-Deskriptor und unterstützt (bei Stream-Sockets) die selben Operationen: `read(2)`, `write(2)`, `close(2)`, ...

Binden eines Sockets an eine Adresse

- Nach seiner Erzeugung muss ein Socket zunächst an eine Adresse *gebunden* werden, bevor er verwendet werden kann
- Der Systemaufruf `bind(2)` stellt eine generische Schnittstelle zum Binden von Sockets in unterschiedlichen Domänen bereit:

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

  - `sockfd`: Socket-Deskriptor
  - `addr`: protokollspezifische Adresse
    - Socket-Interface (`<sys/socket.h>`) ist zunächst protokollunabhängig:

```
struct sockaddr {
    sa_family_t sa_family; // Adressfamilie
    char sa_data[14];      // Platzhalter-Bytes für die Adresse
};
```

  - `addrlen`: Länge der konkret übergebenen Struktur in Bytes

Binden eines 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
};
```

  - `sin_port`: Port-Nummer
  - `sin_addr`: IPv4-Adresse
    - `INADDR_ANY`: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkschnittstellen) Verbindungen akzeptieren soll

- `sin_port` und `sin_addr` müssen in Netzwerk-Byteorder vorliegen!
  - Umwandlung mittels `htons(3)`, `htonl(3)`: konvertiert Datenwort von Host-spezifischer Byteordnung in Netzwerk-Byteordnung (*big endian*) – bzw. zurück:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Binden eines IPv6-Sockets

- Name durch IP-Adresse und Port-Nummer definiert:

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   // = AF_INET6
    in_port_t       sin6_port;     // Port-Nummer
    uint32_t        sin6_flowinfo; // = 0
    struct in6_addr sin6_addr;     // IPv6-Adresse
    uint32_t        sin6_scope_id; // = 0
};

struct in6_addr {
    unsigned char   s6_addr[16];
};
```

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

- Die Werte für `in6addr_any` bzw. `IN6ADDR_ANY_INIT` liegen bereits in Netzwerk-Byteorder vor

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`: Beinhaltet abstrakten „Namen“ (bei uns: IP-Adresse und Port) des Servers
  - `addrlen`: Länge der `addr`-Struktur

- `connect()` blockiert solange, bis der Server die Verbindung annimmt oder zurückweist
- Falls der Socket noch nicht lokal gebunden ist, wird automatisch eine lokale Bindung hergestellt (Port-Nummer wird vom System gewählt)
- Socket ist anschließend bereit zur Kommunikation mit dem Server

DNS-Anfragen

- Zum Ermitteln der Werte für die `sockaddr`-Struktur kann das DNS-Protokoll verwendet werden
- `getaddrinfo(3)` liefert die nötigen 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 die IP-Adresse als String)
  - `service` gibt entweder den 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
- Freigabe der Ergebnisliste nach Verwendung mit `freeaddrinfo(3)`

DNS-Anfragen

```
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
  socklen_t        ai_addrlen;   // Größe von ai_addr
  struct sockaddr *ai_addr;      // Adresse fuer bind()/connect()
  char            *ai_canonname; // Offizieller Hostname (FQDN)
  struct addrinfo *ai_next;      // Nächstes Listenelement 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

DNS-Anfragen: Beispiel

```c
struct addrinfo hints = {
    .ai_socktype = SOCK_STREAM,  // Nur TCP-Sockets
    .ai_family   = PF_UNSPEC,    // Beliebige Protokollfamilie
    .ai_flags    = AI_ADDRCONFIG // Nur lokal verfuegbare Adresstypen
}; // C99: alle anderen Elemente der Struktur werden implizit genullt

struct addrinfo *head;
int error = getaddrinfo("xkcd.com", "80", &hints, &head);
if (error != 0) {
    // Fehler! Behandlung siehe Man-Page
}

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

freeaddrinfo(head);
```


Ein-/Ausgabemechanismen

- Nach dem Verbindungsaufbau lässt sich ein Stream-Socket nach dem selben Schema benutzen wie eine geöffnete Datei
- Für Ein- und Ausgabe stehen verschiedene Funktionen zur Verfügung:
  - 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 *`)

Ebene	Variante	Ein-/Ausgabedaten	Funktionen
2	blockorientiert	Puffer, Länge	`read()`, `write()`
3	blockorientiert	Array, Elementgröße, -anzahl	`fread()`, `fwrite()`
	zeichenorientiert	Einzelbyte	`getc()`, `putc()`
	zeilenorientiert	`'\0'`-terminierter String	`fgets()`, `fputs()`
	formatiert	Formatstring, beliebige Variablen	`fscanf()`, `fprintf()`

Ein-/Ausgabemechanismen

- Auf Grund ihrer Flexibilität eignen sich `FILE *` für String-basierte Ein- und Ausgabe wesentlich besser
- Konvertierung von Dateideskriptor nach `FILE *`:

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

  - `mode` kann sein: `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"` (`fd` muss entsprechend geöffnet sein)
  - Sockets sollten mit `"a+"` geöffnet werden
- Schließen des erzeugten `FILE *`:

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

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

Agenda

Aktive Mitarbeit erforderlich!

„Aufgabenstellung“

- Programm schreiben, welches den Flächeninhalt verschiedener geometrischer Formen berechnen kann
  - Eingaben werden bis `EOF` von `STDIN` eingelesen
  - Eingabeformat: `<zahl> <zahl> <form>`
  - Unterstützte Formen: `dreieck`, `rechteck`
  - Ausgabe soll die eingegebene Werte und den Flächeninhalt beeinhalten