H 6. Übung

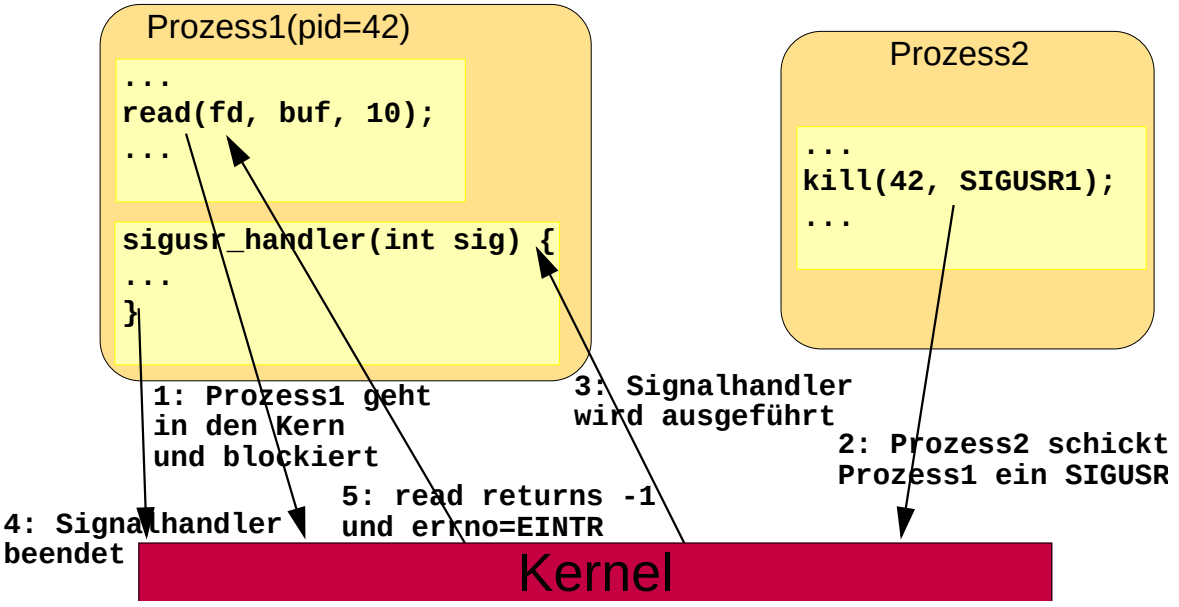
H.1 Überblick

- Besprechung 4. Aufgabe (mysh)
- Signale (Fortsetzung)
- Byteorder bei Netzwerkkommunikation
- Netzwerkprogrammierung - Sockets
- Duplizieren von Filedeskriptoren
- Netzwerkprogrammierung - Verschiedenes

H.2 Signale (Fortsetzung)

1 Unterbrechen von Systemcalls

- Signale können die Ausführung von Systemaufrufen unterbrechen

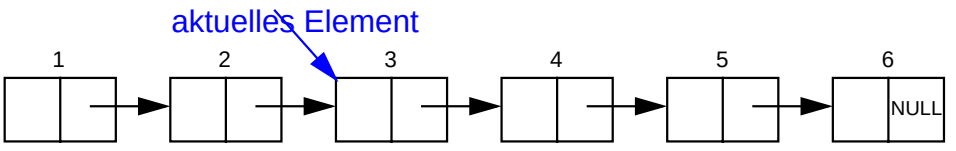


1 Unterbrechen von Systemcalls (2)

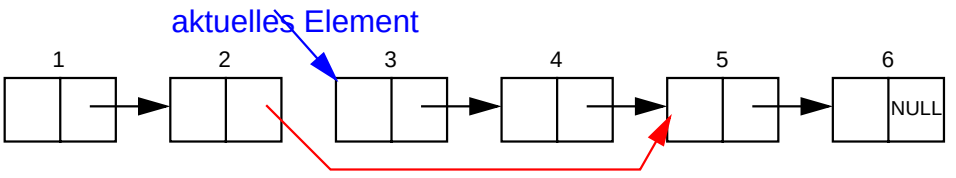
- dies betrifft nur "langsame Systemcalls" (welche sich über einen längeren Zeitraum blockieren können, z.B. **`wait()`**, **`waitpid()`** oder **`read()`** von einem Socket oder einer Pipe)
- der Systemcall setzt dann **`errno`** auf **`EINTR`**
- in einigen UNIXen (z.B. 4.2BSD) werden unterbrochene Systemcalls automatisch neu aufgesetzt
- bei einigen UNIXen (SVR4, 4.3BSD), kann man für jedes Signal einstellen (**SA_RESTART**), ob ein Systemcall automatisch neu aufgesetzt werden soll
- POSIX.1 läßt dies unspezifiziert

2 Signale und Race Conditions

- Signale erzeugen Nebenläufigkeit innerhalb des Prozesses
- diese Nebenläufigkeit kann zu Race-Conditions führen
- Beispiel:
  - ◆ main-Funktion läuft durch eine verkettete Liste



  - ◆ Prozess erhält Signal; Signalhandler entfernt Elemente 3 und 4 aus der Liste und gibt den Speicher dieser Elemente frei

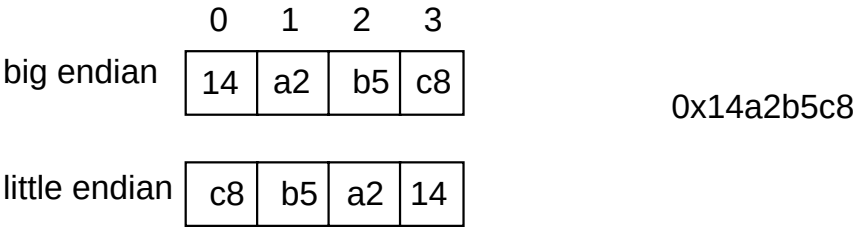


2 Signale und Race Conditions

- Lösung: Signal während Ausführung des kritischen Abschnitts blockieren!
- weiteres Problem:
  - Aufruf von Bibliotheksfunktionen, z.B. **`getpwuid()`**, wird durch Signal unterbrochen und nach Ausführung des Signalhandlers fortgesetzt
  - Signalhandler ruft auch **`getpwuid()`** auf-> Race Condition!
- Lösung:
  - in Signalhandlern nur Funktionen aufrufen, die in POSIX.1 als reentrant gekennzeichnet sind (**`getpwuid`** und **`malloc`**/**`free`** sind z.B. nicht reentrant, **`wait`** und **`waitpid`** sind reentrant)
    - Achtung: wenn in einem Signalhandler Funktionen verwendet werden, die **`errno`** verändern, muß der Wert von **`errno`** vorher gesichert und vor Beendigung des Signalhandlers wieder zurückgesetzt werden
  - oder Signal während Ausführung der Funktion blockieren

H.3 Netzwerkkommunikation und Byteorder

- 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

H.4 Sockets

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

- Domain, z. B. (PF_ = Protocol Family)
  - PF_INET: Internet
  - PF_UNIX: Unix Filesystem
  - PF_APPLETALK: Appletalk Netzwerk
- Type in PF_INET und PF_UNIX Domain:
  - SOCK_STREAM: Stream-Socket (bei PF_INET TCP)
  - SOCK_DGRAM: Datagramm-Socket (bei PF_INET UDP)
  - SOCK_RAW
- Protokoll
  - Default-Protokoll für Domain/Type Kombination: 0 (z.B. INET/STREAM -> TCP) (siehe **`getprotobyname(3)`**)

1 Binden von Sockets

- Was wird gebunden?
  - lokale und remote IP-Adressen, lokale und remote Ports
  - Portnummern sind eindeutig für einen Rechner und ein Protokoll
  - Portnummern < 1024: privilegierte Ports für root (z.B. www=80, Mail=25, finger=79)
  - Portnummern sind 16 Bit, d.h. kleiner als 65535
- Eine Verbindung ist eindeutig gekennzeichnet durch
  - <lokale Adresse, Port> und <remote Adresse, Port>
- **`bind`** bindet an lokale IP-Adresse + Port
  - bind(s, name, namelen)
  - name: Protokollspezifische Adresse (Address Family **`AF_INET`**, IP-Addresse, Port)
  - namelen: Größe der Adresse in Byte

2 Lokales Binden eines TCP Socket

- **INADDR_ANY**: wenn Socket auf allen lokalen Adressen (z.B. allen Netzwerkinterfaces) Verbindungen akzeptieren soll
- **sin_port = 0**: wenn die Portnummer vom System ausgewählt werden soll (Portnummer könnte dann z.B. über Portmapper abfragbar sein)
- 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);
```


3 Socket Adressen

- Socket-Interface (**<sys/socket.h>**) ist protokoll-unabhängig

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

- Internet-Protokoll-Familie (**<netinet/in.h>**) verwendet

```
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 */
}
```

4 Verbindungsannahme durch Server

- Server:
  - **`listen`** stellt ein, wieviele ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein **`accept`** wartend)
  - **`accept`** nimmt Verbindung an:
    - ➤ accept blockiert solange, bis ein Verbindungswunsch ankommt
    - ➤ es wird ein neuer Socket erzeugt und an remote Adresse + Port 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;
  ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```


5 Verbindungsaufbau durch Client

- Client:
  - **`connect`** meldet Verbidungswunsch 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 gleichzeit wird eine lokale Bindung hergestellt (Portnummer wird vom System gewählt)

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

6 Lesen und Schreiben auf Sockets

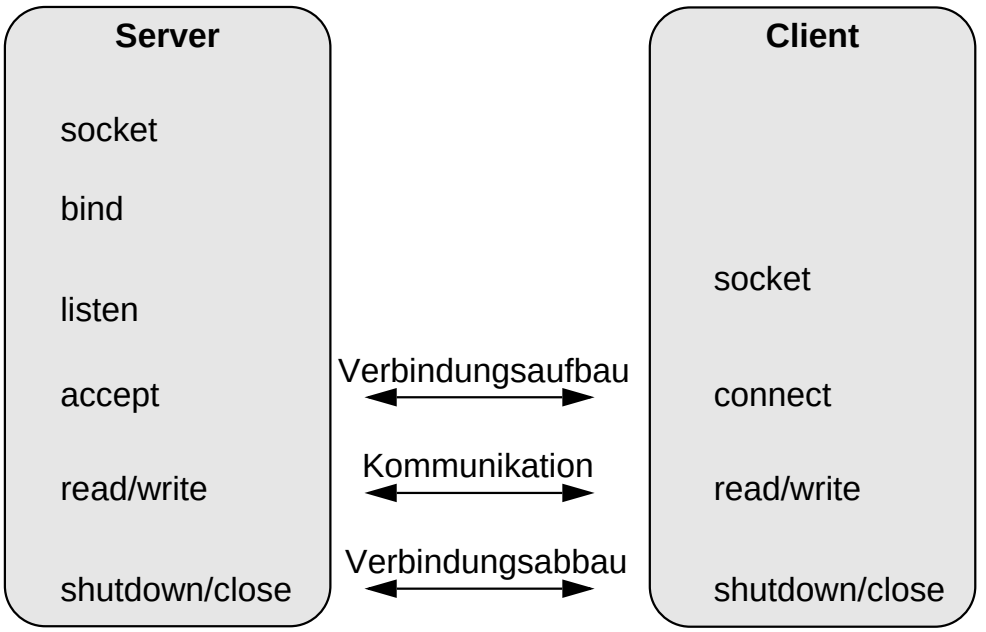
- mit **`read`**, **`write`**
- Beispiel: Server, der alle Eingaben wieder zurückschickt

```
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);
```


7 Schließen einer Socketverbindung

- close(s)
- shutdown(s, how)
- how:
  - SHUT_RD: verbiete Empfang
  - SHUT_WR: verbiete Senden
  - SHUT_RDWR: verbiete Senden und Empfangen

8 TCP-Sockets: Zusammenfassung



9 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)

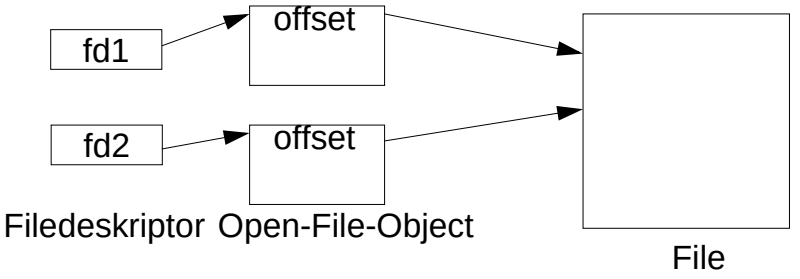
H.5 Duplizieren von Filedeskriptoren

- Ziel: Socket-Verbindung soll als stdout/stdin verwendet werden
- **`newfd = dup(fd)`**: Dupliziert Filedeskriptor fd, d.h. Lesen/Schreiben auf newfd ist wie Lesen/Schreiben auf fd
- **`dup2(fd, newfd)`**: Dupliziert FD in anderen FD (newfd), falls newfd schon geöffnet ist, wird newfd erst geschlossen
- Verwenden von dup2, um stdout umzuleiten:

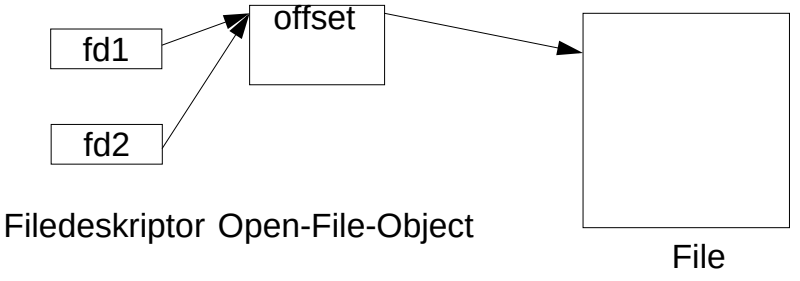
```
fd = open("/tmp/myoutput", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
dup2(fd,fileno(stdout));
printf("Hallo\n"); /* wird in /tmp/myoutput geschrieben */
```


H.5 Duplizieren von Filedeskriptoren (2)

- erneutes Öffnen eines Files



- bei dup werden FD dupliziert, aber Files werden nicht neu geöffnet!



H.6 Netzwerk-Programmierung - Verschiedenes

- Informationen über Socket-Bindung
- Hostnamen und -adressen ermitteln

1 getsockname, getpeername

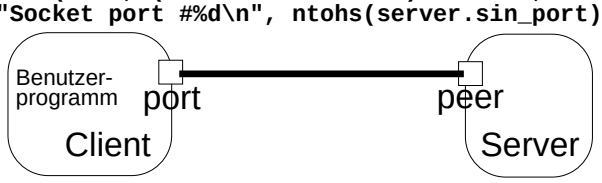
```
#include <sys/socket.h>
int getsockname(int s, void *addr, int *addrlen);
int getpeername(int s, void *addr, int *addrlen);
```

- Informationen über die lokale Adresse des Socket

```
struct sockaddr_in server;
size_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;
size_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 */
};

#define h_addr h_addr_list[0]
```

- **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 = "faui07a";
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((char *) &saddr.sin_addr, (char *) host->h_addr, host->h_length);
saddr.sin_family = AF_INET;
saddr.sin_port   = htons(port);

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