D Überblick über die 4. Übung

- Infos zur Aufgabe2: Kommunikationssystem
- Wiederholung Sockets
  - TCP
  - UDP
- (Wiederholung) Threads
  - Pthreads

D.1 Aufgabe2

1 Basissystem

- eine einfache Kommunikationsschnittstelle

```
typedef void(*send_signal_t)(Buffer *buf);
typedef void(*recv_signal_t)(Address *sender, Buffer *buf);

class Buffer {
    public:
        char *buffer;
        int len;
}

class CommunicationSystem {
    public:
        CommunicationSystem(Address *addr);
        Address *getMyAddress();

        void send(Address *dest, Buffer *message);
        void set_send_signaler(send_signal_t fn);

        void set_receive_signaler(Buffer *buffer,
                                  recv_signal_t fm);
}
```


1 Basissystem

- eine einfache Semaphore

```
class Semaphor{
    public:
        Semaphor();
        void P(void);
        void V(void);
};
```

- eine einfache Thread-Verwaltung

```
class Runnable {
    virtual void run() = 0;
}

class Thread {
    public:
        Thread(Runnable run);
        void join();
}
```


2 Kommunikationsvarianten

- synchrones send
- asynchrones, pufferndes send
- asynchrones, nicht-blockierendes send

D.2 Sockets

- Erzeugung
- Binden
  - Socket Adressen
- TCP-Sockets
  - Server
  - Client
- UDP-sockets

1 Erzeugen eines neuen Sockets:

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

- **`domain`** gibt die Kommunikations Domäne an, z.B.
  - **`PF_INET`**: Internet, IPv4
  - **`PF_UNIX`**: Unix Filesystem, lokale Kommunikation
  - **`PF_APPLETALK`**: Appletalk Netzwerk
- Durch **`type`** wird die Kommunikations Semantik festgelegt z.B.: in **`PF_INET`** Domain:
  - **`SOCK_STREAM`**: Stream-Socket
  - **`SOCK_DGRAM`**: Datagramm-Socket
  - **`SOCK_RAW`**

1 Erzeugen eines neuen Sockets (2)

- **`protocol`** legt das Protokoll fest.
  - 0 bedeuted hierbei: Standardprotokoll für Domain/Type Kombination
  - Normalerweise gibt es zu jeder Kombination aus Sockettyp/-familie nur ein Protokoll:
    - **`PF_INET`**, **`SOCK_STREAM`**: → TCP
    - **`PF_INET`**, **`SOCK_DGRAM`**: → UDP

2 Binden von Sockets

- Binden eines Sockets an eine lokale Adresse
- **`bind`** bindet an lokale IP-Adresse + Port

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

  - **`sockfd`**: Socketdeskriptor
  - **`my_addr`**: Protokollspezifische Adresse
  - **`addrlen`**: Größe der Adresse in Byte

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 Lokales Binden eines Sockets

- `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)
- Portnummern < 1024: privilegierte Ports für root
- 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);
```


5 Socket-Adresse aus Hostnamen erzeugen

- `gethostbyname` liefert Informationen über einen Host

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

- Strktur `hostent`

```
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]
```


5 Socket-Adresse aus Hostnamen erzeugen (2)

- Beispiel

```
char *hostname = "faui40";
int port = 4711;
struct hostent *host;
struct sockaddr_in saddr;

host = gethostbyname(hostname);
if(!host) {
    perror("gethostbyname()");
        exit(EXIT_FAILURE);
}
memset(&saddr, 0, sizeof(saddr)); /* 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, connect bzw. sendto*/
```

6 TCP Verbindung: Verbindungsannahme durch Server

- **`listen`** stellt ein, wieviele (**`backlog`**) ankommende Verbindungswünsche gepuffert werden können (d.h. auf ein **`accept`** wartend)

```
int listen(int s, int backlog);
```

- **`accept`** nimmt Verbindung an

```
int accept(int s,
           struct sockaddr *addr, socklen_t *addrlen);
```

  - **`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
  - **`addr`**: enthält die Adresse des Kommunikationspartners

6 TCP Verbindung: Verbindungsannahme durch Server

- Beispiel

```
...
listen(s, 5);              /* Allow queue of 5 connections */

struct sockaddr_in from;
socklen_t fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```


7 TCP Verbindung: Verbindungsaufbau durch Client

- **`connect`** meldet Verbindungswunsch an Server

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

  - **`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)

- Beispiel:

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


8 Lesen und Schreiben auf Sockets

- mit **`read`**, **`write`**

- Beispiel: Server, der alle Eingaben wieder zurückschickt

```
int fd = socket(PF_INET, SOCK_STREAM, 0); /* Fehlerabfrage */

struct sockaddr_in name;
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));/*.*/

listen(fd, 5); /* Fehlerabfrage */

int in_fd = accept(fd, NULL, 0); /* Fehlerabfrage */

char * buf = new char[100];
for(;;) {
    int n = read(in_fd, buf, sizeof(buf)); /* Fehler? */
    write(in_fd, buf, n); /* Fehlerabfrage */
}

close(in_fd);
```

8 Lesen und Schreiben auf Sockets (2)

- mit **send**, **recv**
- 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));/*.*/

listen(fd, 5); /* Fehlerabfrage */

in_fd = accept(fd, NULL, 0); /* Fehlerabfrage */

for(;;) {
    int n = recv(in_fd, buf, sizeof(buf), 0); /* Fehler? */
    send(in_fd, buf, n, 0); /* Fehlerabfrage */
}

close(in_fd);
```


9 Schließen einer Socketverbindung

- close

```
#include <unistd.h>

int close(int fd);
```

- shutdown

```
int shutdown(int s, int how);
```

- **how**:
  - SHUT_RD: verbiete Empfang
  - SHUT_WR: verbiete Senden
  - SHUT_RDWR: verbiete Senden und Empfangen

10 Probleme beim wiederverwenden des Ports

- Socketoptionen

```
int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);
```

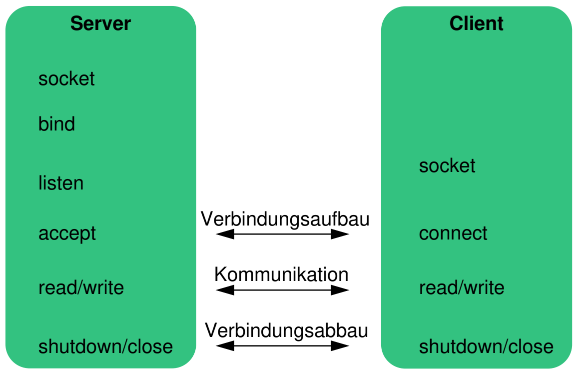
  - **level**: z.B.: SOL_SOCKET: Socket Ebene
  - **optname**: Name der Option (für SOL_SOCKET siehe **socket(7)**)
    z.B.: SO_REUSEADDR: **bind** soll lokale Adresse sofort wiederverwenden
  - **val**: Wert

- Binden an eine Portnummer, die sich im Status TIME_WAIT befindet

```
int sock;
int val = 1;

if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        perror ("socket open failed");
        exit (2);
}
setsockopt (sock,SOL_SOCKET,SO_REUSEADDR,&val,sizeof val);
```


11 TCP-Sockets: Zusammenfassung



12 UDP Verbindung

- Verbindungslos: kein **accept**, kein **connect**
- Empfänger bzw. Absender kann bei jedem Paket anders sein
- Senden

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

  ◆ mittels **to** wird die Zieladresse festgelegt

- Empfangen

```
int recvfrom(int s, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

  ◆ in **from** steht der Absender

D.3 Threads

- Vergleich von Prozess und Thread-Konzepten
- POSIX-Threads

1 Motivation von Threads

- UNIX-Prozesskonzept ist für viele heutige Anwendungen unzureichend
- in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
- zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
- typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server für jeden Client zu erzeugen

  ➥ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)

2 Vergleich von Prozess- und Thread-Konzepten

- mehrere **UNIX-Prozesse** mit gemeinsamen Speicherbereichen

Bewertung:

+ echte Parallelität möglich
– viele Betriebsmittel zur Verwaltung eines Prozesses notwendig; Prozessumschaltungen aufwendig → teuer
– innerhalb einer solchen Prozessfamilie wäre häufig ein anwendungsorientiertes Scheduling notwendig: schwierig realisierbar

2 Vergleich von Prozess- und Thread-Konzepten (2)

- ***User-Level-Threads*** (Koroutinen) — Realisierung von Threads auf Benutzerebene innerhalb eines Prozesses

  Bewertung:

  + Erzeugung von Threads und Umschaltung extrem billig
  + Verwaltung und Scheduling anwendungsorientiert möglich
  – Systemkern hat kein Wissen über diese Threads
    - ➥ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
    - ➥ in Multiprozessorsystemen keine parallelen Abläufe möglich
    - ➥ wird eine Koroutine wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert

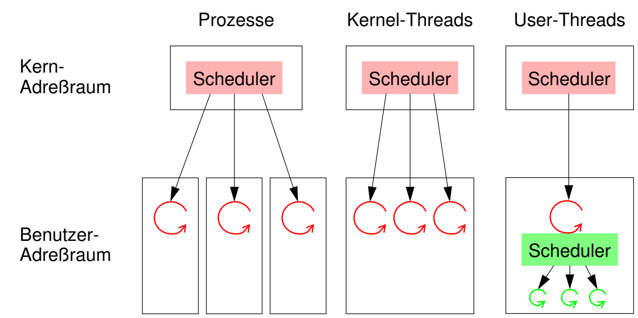
2 Vergleich von Prozess- und Thread-Konzepten (3)

- ***Kernel-Threads***: leichtgewichtige Prozesse *(lightweight processes)*

  Bewertung:

  + eine Gruppe leichtgewichtiger Prozesse nutzt gemeinsam eine Menge von Betriebsmitteln
  + jeder leichtgewichtige Prozess ist aber als eigener Aktivitätsträger dem Betriebssystemkern bekannt
    - eigener Programmzähler
    - eigener Registersatz
    - eigener Stack

2 Vergleich von Prozess- und Thread-Konzepten (5)



2 Vergleich von Prozess- und Thread-Konzepten (3)

... Bewertung *Kernel-Threads (lightweight processes)*

+ Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung
  - ➥ es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
  - ➥ Adreßraum muss nicht gewechselt werden
  - ➥ alle Systemressourcen bleiben verfügbar
– Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei “schwergewichtigen” Prozessen, aber erheblich teuerer als bei *user-level-Threads*
– Verwaltung und Scheduling meist durch Kern vorgegeben

2 Vergleich von Prozess- und Thread-Konzepten (4)

- Vergleich

	Prozesse	Kernel-Threads	User-Threads
Kosten	– teuer	○ mittel	+ billig
Betriebssystemeingliederung	+ gut	+ gut	– schlecht
Interaktion untereinander	– schwierig	+ einfach	+ einfach
Benutzerkonfigurierbarkeit	– nein	– nein	+ ja
Gerechtigkeit	– nein	+ ja	± teils

– Gerechtigkeit bedeutet:
wie kommt das System damit klar, wenn eine Anwendung eine große Anzahl von Aktivitätsträgern erzeugt, eine andere dagegen eine geringe — werden Zeitscheiben an Anwendungen oder an Aktivitätsträger vergeben?

3 UNIX — Prozesse, LWPs & Threads

- Thread-Konzept zunehmend auch in UNIX-Systemen realisiert
  - ◆ Solaris
  - ◆ HP UX
  - ◆ Digital UNIX
  - ◆ Linux
  - ◆ ...
- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE POSIX Standard P1003.4a
- Pthreads-Implementierungen aber sehr unterschiedlich!
  - ➤ reine User-level-Threads (HP-UX)
  - ➤ reine Kernel-Threads (Linux, MACH, KSR-UNIX, Digital UNIX)
  - ➤ parametrierbare Mischung (Solaris)
- Daneben z. T. auch andere Thread-Bibliotheken (z. B. Solaris-Threads)

4 pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

4 pthread-Benutzerschnittstelle (2)

- Threaderzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

- ◆ `thread` Thread-Id
- ◆ `attr` modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). `NULL` für Standardattribute.
- ◆ Thread wird erzeugt und startet mit der Ausführung der Funktion `start_routine` mit Parameter `arg`
- ◆ Rückgabewert: 0; im Fehlerfall -1 außerdem wird `errno` gesetzt

4 pthread-Benutzerschnittstelle (3)

- explizites beenden eines Threads:

```
void pthread_exit(void *retval)
```

  - ◆ Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe `pthread_join`)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

  - ◆ Wartet auf den Thread mit der Thread-ID `thread` und liefert dessen Rückgabewert über `retvalp` zurück.

5 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                              (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

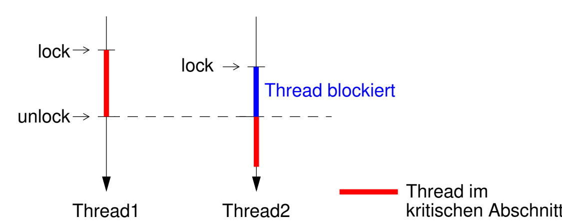

6 Pthreads-Koordinierung

- UNIX-Semaphore für Koordinierung von leichtgewichtigten Prozessen zu teuer
  - ◆ Implementierung durch den Systemkern
  - ◆ komplexe Datenstrukturen
- Bei Koordinierung von Threads reichen meist einfache ***mutex***-Semaphore
  - ◆ gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

6 Pthreads-Koordinierung (2)

★ **Mutexes**

- Koordinierung von kritischen Abschnitten



6 Pthreads-Koordinierung (3)

- Mutex erzeugen

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)
```

- Lock & unlock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Beispiel:

```
pthread_mutex_t m1;
pthread_mutex_init(&m1, pthread_mutexattr_default);
...
pthread_mutex_lock(&m1);
... kritischer Abschnitt
pthread_mutex_unlock(&m1);
```


7 Zusammenfassung - Pthreads

- standardisierte Thread API
- unterschiedliche Implementierung
- einfache Threaderzeugung mittels `pthread_create`
- Koordinierung mit Hilfe von Mutexes
  - `pthread_mutex_lock, pthread_mutex_unlock`