Interprozesskommunikation

(IPC)

Gliederung

Interprozesskommunikation (IPC)

Übersicht:

- Übersicht von IPC-Möglichkeiten
- Charakteristika von IPC-Mechanismen
- Basis-Mechanismen:
  - (Signale -> Kap.2)
  - (Synchronisation prozessübergreifend -> Kap.5)
  - Pipes
  - Sockets
  - Shared-Memory
- Tips zur praktischen Anwendung

Übersicht an IPC-Möglichkeiten

Welche Möglichkeiten zur Interprozeßkommunikation gibt es ?

- Prozessdaten sind vor Zugriff anderer Prozesse geschützt
- Mechanismen zur **Interprozesskommunikation** (IPC) ermöglichen Datenaustausch !
- technische Möglichkeiten:

  - Basismechanismen in Linux/Unix:
    - Signale (siehe Kap.2 + Praktikum 3)
    - Synchronisation prozessübergreifend
    - Pipes
    - Sockets
    - Shared-Memory
  - aufsetzende Middleware-Lösungen (u.a.):
    - synchrone Kommunikation:
      - Remote-Porcedure-Calls (RPC)
      - Java Remote Method Invocation (RMI)
      - CORBA
    - asynchron:
      - SmartSockets
      - MqSeries Messaging
      - Java Message Service (JMS)

Charakteristika von IPC-Mechanismen

Welche Eigenschaften weisen IPC-Mechanismen auf ?

- **Kommunikationsmodell:**
  - Punkt-zu-Punkt (verbindungsorientiert)
  - publish-subscribe (verbindungslos)
  - broadcast-Kommunikation
- **Übertragungsrichtung:**
  - simplex / uni-direktional
  - duplex / bi-direktional
- **Synchronizität:**
  - synchron / blockierend
  - asynchron / nicht-blockierend / Nachrichten-basiert
- weitere Eigenschaften:
  - Plattformunabhängigkeit
  - Portierbarkeit
  - Reichweite:
    - systemgebunden
    - über Rechnergrenzen hinweg

Posix Semaphore

- **Posix Semaphore:**
  - auch prozeßübergreifend
  - **System-Calls:**
    - #include <semaphore.h>
      int sem_init(sem_t *sem, int pshared, unsigned int value);
      int sem_wait(sem_t * sem);
      int sem_trywait(sem_t * sem);
      int sem_post(sem_t * sem);
      int sem_getvalue(sem_t * sem, int * sval);
      int sem_destroy(sem_t * sem);
    - nicht zu verwechseln mit Unix-SysV Semaphoren:
      ipc(), semctl(), semop();

Pipes (1)

- **Pipes:**
  - Schnittstellen zum Aufbau eines lokale Kommunikationskanals
  - **Eigenschaften:**
    - FIFO-Prinzip: Datenaustausch als Byte-Stream (keine Satz- oder Bachrichtenstruktur)
    - kein Protokoll
    - unidirektional
    - blockierend (schreiben in volle / lesen aus leerer Pipe)
  - **weitere Eigenschaften:**
    - nicht plattformübergreifend / nicht rechnerübergreifend / portierbar unter Unix-Derivaten
    - typischerweise in Verbindung mit fork() o.ä. (verwandte Prozesse)

Pipes (2)

- ...
- Pipes werden über Dateideskriptoren angesprochen
- **System-Calls:**
  - #include <unistd.h>
    int **pipe**(int filedes[2]);
    - zwei File-Deskriptoren werden erzeugt (Lesen / Schreiben)
    - Lesen und Schreiben entsprechend wie bei Dateien
  - #include <stdio.h>
    FILE * **popen**(const char * command, const char * type);
    int **pclose**(FILE * stream);
  - Bsp: fd = popen("ls -tr1", 'r')

Pipes (3)

- **Named Pipes:**
  - auch bidirektional
  - verwendbar zwischen beliebigen Prozessen, nur Kenntnis der Namen notwendig:
  - **Beispiel:**
    - mknod(pipenamefrom, S_IFIFO | 0666, 0); //Pipe1 erzeugen
      mknod(pipenameto, S_IFIFO | 0666, 0); //Pipe2 erzeugen
    - pcreatelp("/usr/toppic/src/servPgm", "servPgm", //ServerPgm starten
      pipenameto, pipenamefrom, (char *)0); //pcreate() ähnlich fork+exec
    - pipetos = open(pipenameto, O_WRONLY); //Pipes öffnen
      pipefroms = open(pipenamefrom, O_RDONLY);
    - unlink(pipenameto); //Dateien aus Verzeichnis,
      unlink(pipenamefrom); //da temporär, entfernen

Sockets (1)

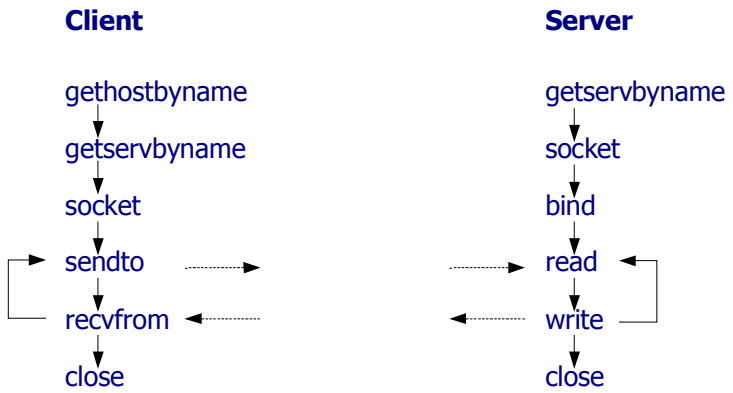
- **Sockets:**
  - Netzwerkschnittstellen zum Aufbau eines Kommunikationskanals
  - **Eigenschaften:**
    - mehrere IP-Protokolle (TCP, UDP, IPX, u.a.)
    - Kommunikationsmodelle: je nach Protokoll und Adressierung **alle** Modelle
    - bidirektional / duplex
    - sowohl blockierend als auch asynchron möglich
  - **weitere Eigenschaften:**
    - plattformübergreifend / portierbar / rechnerübergreifend

Sockets (2)

- **...**
- **Kommunikationsarten:**
  - Nachrichten-orientiert: `recvmsg() / sendmsg()`, oder
  - Bytestrom-orientiert: `sendto() / recvfrom(), read(), write()`
  - Standard-Pufferung: 8 kB (kann optional eingestellt werden):
    - Senden blockiert bei umfangreicheren Daten, bis Gegenstelle gelesen hat
    - nichtblockierendes (asynchrones) Verhalten kann optional eingestellt werden
- Sockets werden in Unix auf Dateideskriptoren abgebildet:
  - viele SystemCalls wie für Dateien (`read(), write() close()`)
  - viele Flags wie für Dateien

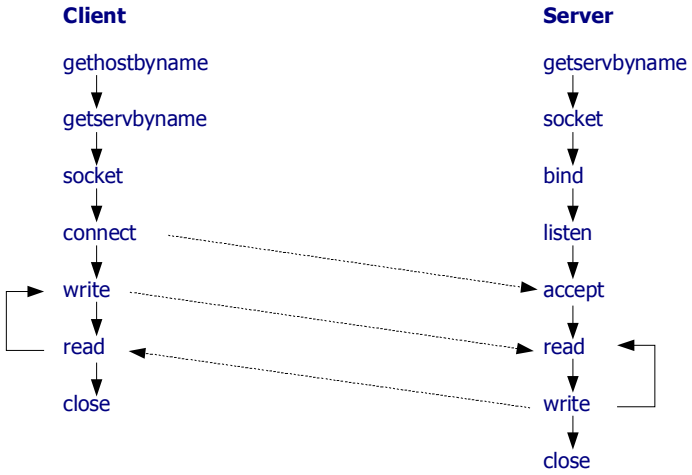
Sockets (3)

Verbindungslose Kommunikation über Datagramme / UDP-Protokoll



Sockets (4)

Verbindungsorientierte Kommunikation über Streams / TCP-Protokoll



Sockets (5)

- **...**
- **Adressierung**, z.B.
  - lokal (AF_UNIX): Pfadname
  - Netzadresse (AF_INET): host+port
    - lokal über AF_INET: host = 'localhost' oder 127.0.0.1
- **Zuverlässigkeit:**
  - reliable, verbindungsorientiert (z.B. TCP-Protokoll): fehlerfreie Zustellung:
    - keine Verluste
    - keine Duplikate
    - korrekte Reihenfolge
  - unreliable, verbindungslos (z.B. UDP-Protokoll / Datagramm)

Anm.: C-Beispiele für eigene Experimente am Ende

Shared Memory (1)

- **Shared-Memory:**
  - #include <sys/types.h>
  - #include <sys/ipc.h>
  - #include <sys/shm.h>
  - int mode;
  - const int SHM_SIZE = 1024; //1 kB Region
  - Erzeugen eines eindeutigen Identifikations-Keys:

    key_t key = **ftok**("shmdemo.c", 'R');

  - Erzeugen einer neuen oder Erhalt der Identifikation einer bestehenden Shared-Memory Region:

    int shmid = **shmget**(key, SHM_SIZE, flags);

    flags: Zugriffsrechte, und ob neu zu erzeugen, z.B. 0644 | IPC_CREAT

Shared Memory (2)

- ...
  - Verknüpfen (*attach*) einer Shared-Memory Region mit einem Prozeß:

    char * data = **shmat**(shmid, (void *)0, flags);

    flags: Zugriffsrechte, und ob Adresse vom Kernel gerundet werden soll
  - nun Verwendung des Datenpointers ...
  - Lösen (*detach*) der Shared-Memory Region vom Prozess:

    shmdt(data);

    Eine Shared-Memory Region wird nicht gelöscht, wenn kein Prozeß mehr mit ihr verknüpft ist, außer sie wurde zuvor von einem prozeß freigegeben
  - Freigabe einer Shared-Memory Region:

    shmctl(shmid, IPC_RMID, 0);

Praxisbeispiele

Praxis: Übersicht

- **Pipes auf der Shell**
- **Sockets:**
  - Datagram-Server + -Client
  - verbindungsorientiert: Server- + Client
  - mögliche Probleme
-

Praxis: Pipes auf Shell

- **Beispiel:** Sortieren einer Datei und Ausgabe der ersten 30 Sätze:
  - „ohne“ Pipe:
    ```
    sort < datei > temp
    head -30 < temp
    rm temp
    ```
  - mit Pipe:
    ```
    sort < datei | head -30
    ```

  stdout des 1. Kommandos wird mit *stdin* des 2. Kommandos mit „|“ durch Pipe verbunden
- **Beispiel2:** Suche nach 'deprecated' auch in *stderr*
  - `(make 2>&1) | grep deprecated` //Umleitung von stderr nach stdout + grep stdout

Praxis: mögliche Probleme bei Sockets

- **mögliche Probleme:**
  - Port schon belegt: lsof („list of open files and sockets“)
  - Port noch belegt : setzen der LINGER-Option
    bspw. nach Stoppen des Servers, noch falls Daten anhängig sind
- **Nachrichtengrenzen:** bleiben ggf. nicht erhalten:

  z.B. send(170 Byte) + send(230 Byte) --> reveive(400 Byte)
- **asynchrones** Verhalten:

  API-Funktionen erlauben optional auch nicht-Blockierendes Verhalten

Praxis: Datagram-Server

```
#include <unistd.h>         // read(), close()
#include <arpa/inet.h>      // sockaddr_in, INADDR_ANY
#include <sys/socket.h>     // SOCK_DGRAM, socket(), bind()

const short port = 5242;
int         n, sockfd;
char        buf[256];
struct sockaddr_in serv_addr;

int main() {    //UDP_Socket
    if ((sockfd = socket( AF_INET, SOCK_DGRAM, 0 )) < 0)  perror("opening datagram");

    // Create name with wildcards
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons( port );

    if ( bind( sockfd, (sockaddr *)&serv_addr, sizeof(serv_addr) ) != 0 )
        perror( "binding to address" );

    n = read( sockfd, buf, sizeof(buf) ); printf( "Received: %s\n", buf );

    close( sockfd );

    return  0;
}
```


Praxis: Datagram-Client

```
#include <unistd.h>         // read(), close()
#include <arpa/inet.h>      // sockaddr_in, AF_INET
#include <sys/socket.h>     // SOCK_DGRAM, socket(), bind()
#include <sys/param.h>      // MAXHOSTNAMELEN
#include <netdb.h>          // gethostbyname()

const short port = 5242;
char        hostname[MAXHOSTNAMELEN+1] = "server";
int         sockfd;
struct sockaddr_in peer_addr;

int main() {    //UDP_Socket
    if ((sockfd = socket( AF_INET, SOCK_DGRAM, 0 )) < 0)  perror("opening datagram");

    struct hostent * hp = gethostbyname( hostname );
    bcopy( hp->h_addr, (char *)&peer_addr.sin_addr, hp->h_length );
    peer_addr.sin_family = AF_INET;
    peer_addr.sin_port = htons( port );

    if ( sendto( sockfd, "Hello World", 11, 0, (sockaddr *)&peer_addr,
        sizeof(peer_addr) ) < 0 )  perror( "sending data" );

    close( sockfd );

    return  0;
}
```


Praxis: Server verbindungsorientiert

```
#include <unistd.h>         // read(), close()
#include <arpa/inet.h>      // sockaddr_in, INADDR_ANY
#include <sys/socket.h>     // SOCK_DGRAM, socket(), bind()

const short port = 5242, waitqueuelen = 1;
int         n, sockfd, con;
char        buf[256];
struct sockaddr_in serv_addr;

int main() {    //TCP_Socket
    if ((sockfd = socket( AF_INET, SOCK_STREAM, 0 )) < 0)  perror("opening stream");
    serv_addr...;

    if ( bind( sockfd, (sockaddr *)&serv_addr, sizeof(serv_addr) ) != 0 )
        perror( "binding to address" );

    if ( listen( sockfd, waitqueuelen ) != 0 )  perror( "listening to address" );

    if ( ( con = accept( sockfd, (sockaddr *)&peer_addr, sizeof(serv_addr) ) < 0 )
        perror( "accepting client" );

    n = read(con, buf, sizeof(buf)); printf("Received: %s\n", buf); write(con, buf, n);

    close( con );
    close( sockfd );

    return  0;
}
```


Praxis: Client verbindungsorientiert

```
#include <unistd.h>         // read(), close()
#include <arpa/inet.h>      // sockaddr_in, AF_INET
#include <sys/socket.h>     // SOCK_DGRAM, socket(), bind()
#include <sys/param.h>      // MAXHOSTNAMELEN
#include <netdb.h>          // gethostbyname()

const short port = 5242;
char        hostname[MAXHOSTNAMELEN+1] = "server";
int         sockfd;
struct sockaddr_in peer_addr;

int main() {    //TCP_Socket
    if ((sockfd = socket( AF_INET, SOCK_STREAM, 0 )) < 0)  perror("opening datagram");
    peer_addr...;

    if ( connect( sockfd, (sochaddr *)&peer_addr, sizeof(peer) ) != 0 )
        perror( "connecting server" );

    write( sockfd, "Hello World", 11 );
    read( sockfd, buf, sizeof(buf)); printf("Received: %s\n", buf);

    close( sockfd );

    return  0;
}
```