12. Interprozesskommunikation

Michael Schöttner

Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

12.1 Überblick

- Kooperation zw. Prozessen, um Aufgaben gemeinsam zu bearbeiten
  → Interprozesskommunikation.
- Interprozesskommunikation:
  - Engl. Inter-Process Communication (IPC)
  - Datenaustausch umständlicher, als innerhalb eines Adressraums
- Verschiedene Techniken:
  - Pipes, Sockets (Domain, UDP, TCP),
  - Signale und Semaphore
  - Shared Memory
- Konkurrenz zw. Prozessen und Threads bzgl. gemeinsamer Ressourcen
  → Synchronisierung

12.2 Klassifikation

- Synchron / asynchron:
  - synchron: Sender wird bis zur Auslieferung der Nachricht blockiert
  - asynchron: Sender arbeitet sofort weiter
- Gepuffert / ungepuffert:
  - gepuffert: u.U. mehrere Nachrichten bündeln
  - ungepuffert: direktes Empfangen/Übertragen
- Verbindungsorientiert / verbindungslos:
  - verbindungsorientiert: expliziter Verbindungsaufbau vor der Kommunikation
  - verbindungslos: Zieladresse in jeder Nachricht; Multicast und Broadcast mögl.

12.2 Klassifikation

- Meldungs-/auftragsorientiert:
  - Meldung: Sender wird bis zur Bestätigung der Meldung blockiert
  - Auftrag: Sender wird bis zur Beendigung des Auftrages blockiert
- Kommunikationsarten:
  - nachrichtenbasiert: Socket, Message Passing Interface (MPI), Remote Method Invocation (RMI), ...
  - speicherbasiert: (Distributed) Shared Memory
- Übertragungsrichtung:
  - unidirektional (immer nur in eine Richtung)
  - bidirektional (voll-duplex vs. halb-duplex)

12.3 Kommunikation über Dateien (UNIX)

Prozess-Synchronisierung per Sperrdatei

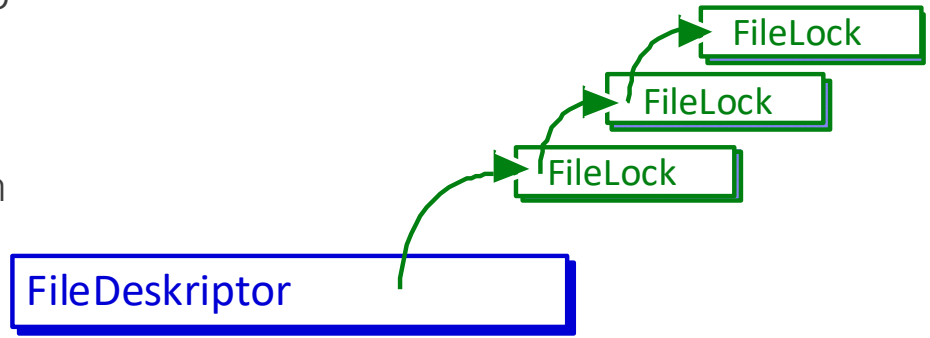
- Gewöhnliche Datei, typischerweise in `/tmp`
- Exklusiver Zugriff durch erfolgreiches Erzeugen der Datei (schreibgeschützt für andere Benutzer).
- Die Anfrage, ob eine Datei existiert und, wenn nicht, das Anlegen geschieht atomar durch den Systemaufruf **`creat`**.
- Versucht ein anderer Benutzer, die schreibgeschützte Datei zu erzeugen, wird ihm der Zugriff verweigert.
- Diese Zusicherung des atomaren **`creat`** (ohne e) ist eine alte UNIX- Synchronisierungstechnik (definiert in **`fcntl.h`**).

Prozess-Synchronisierung per Sperrdatei

- Funktioniert nicht für zur Synchronisierung gegenüber root, da root immer schreiben darf.
- Freigeben der Sperrdatei durch `unlink`-Aufruf (löscht Datei).
- Alternativ: Systemaufruf open mit Flags `O_CREAT | O_EXCL`
  → Fehlermeldung `EEXIST`, falls Datei schon existiert

Sperren von Dateiabschnitten

- Konkurrierende Schreibzugriffe auf Dateien werden zunächst durch **`open`** und **`close`**-Routinen geregelt.
  - Nur exklusiv öffnen
  - Nur im Read-Only Mode öffnen
  - Im Multiple Writer Modus öffnen
    → Risiko von Inkonsistenzen.



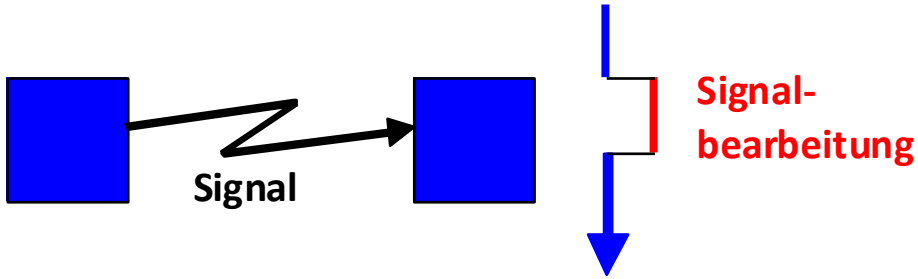
- Im Multiple Writer Modus
  → Sperrung von Dateibereichen:

```
int fcntl( unsigned int fileDeskriptor,
           unsigned int command,        /* F_GETLK, ... */
           struct flock *fileLock
         );
```

- **`struct flock`**:
  - Lese-/Schreibsperre, Dateibereich, PID

12.4 Signale in UNIX

- Signale: kurze wichtige Meldungen über async. Ereignisse → Exceptions.
- Generiert von Kern- oder Benutzerprozessen.
- Führen typischerweise zur Terminierung.
- Beim Auftreten des Signals wird die normale Ausführung der Befehlssequenz unterbrochen, und der Handler wird asynchron aufgerufen:



UNIX Signale (Auszug)

Signal	Ursache
SIGABRT	Sent to abort process and force a core dump
SIGILL	The process has exectured an illegal machine instruction
SIGINT	The user has hit the DEL key to interrupt the process
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written on a pipe with no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Signale Senden

- Mit `kill –SIG PID` (SIG = Signalname bzw. ID)
- Oder Aufruf `sigsend(P_PID, ID, SIG);`
  (P_PID = type process ID, ID = process ID)
- Setzt entsprechendes Bit in Variable `signal` in `task_struct`
- Falls Signal nicht geblockt und Prozess nicht in der Bereit-Queue ist, ihn dorthin verschieben.
- Berechtigung:
  - Kern- und root-Prozesse dürfen an alle Prozesse Signale schicken
  - Normale Prozesse dürfen nur an Prozesse mit gleicher User-/Group-ID senden

Implementierung in Linux (Auszug)

- Wichtige Strukturen:

```
struct task_struct {
   ...
   unsigned long signal;            /* wartende Signale                    */
   unsigned long blocked;           /* blockierte Signale                  */
   struct sigaction sigaction[];    /* Behandlung der Signale, siehe unten  */
   ...
}

struct sigaction  {
   void (*sa_handler) (int);        /* Funktions-Zeiger auf Handler,
                                       Parameter = Signalnummer            */
   sigset_t    sa_mask;             /* Signale, die während der Ausführung
                                       des Handlers blockiert werden sollen */
   int         sa_flags;            /* für Sonderzwecke        */
}
```

Beispiel: eigener Signal-Handler

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

/* Signal auslösen mit: kill -SIGUSR1 pid  */
void myHandler(int parm) {
   printf("received SIGUSR1\n");
}

int main() {
   struct sigaction action,old;

   action.sa_flags = 0;                  /* Standardverhalten für Signale          */
   sigemptyset(&action.sa_mask);         /* Alle Signale unterdrücken              */
   action.sa_handler = &myHandler;

   sigaction(SIGUSR1, &action,  &old); /* Signaltyp, neuer Handler, alter Handler */
   sleep(60);                          /* 60s schlafen -> unterbrechbar per Signal*/
   sigaction(SIGUSR1, &old, NULL);     /* alter Handler wieder einrichten         */
}
```

12.5 Pipes in UNIX

- Pipe: zuverlässige Kommunikation über Kanal:
  - Ordnung der Zeichen bleibt erhalten
  - Blockierung bei voller/leerer Pipe
  - Unidirektionale Kommunikation
- **Anonyme Pipes**: temporäre Spezialdateien, die Prozesse verbinden
  - z.B. Umlenken der Ausgabe
- **Benannte Pipes**: Spezialdateien mit Berechtigungen
- Bem.: Klassische IPC Methode unter UNIX.

Anonyme Pipes

- Eine temporäre memory-mapped Datei
- Inode der Datei zeigt auf eine Speicherseite
- Datenaustausch über diese Datei/Seite
- Zwei Handles für read/write

- Anwendungsbeispiel: `% ls -l | more`

Beispiel: anonyme Pipe (Wiederholung)

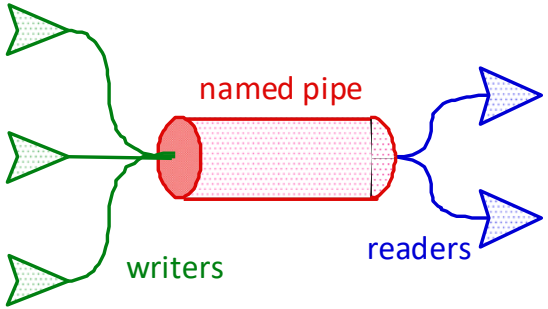
```
#include <stdio.h>
#include <unistd.h>           /* different standard constants              */

int main() {
   char   data[80];
   int    rb;                 /* nr of read bytes                          */
   int    pipe_ends[2];       /* handles: read=0; write=1                  */

   pipe(pipe_ends);           /* create anonymous pipe                     */

   if (fork()==0) {           /* child process                             */
      close(pipe_ends[1]);    /* close write end -> process wants to read  */
      rb = read(pipe_ends[0], data,79);
      data[rb]='\0';          /* terminate string                          */
      printf ("%s\n",data);
      close(pipe_ends[0]);    /* done, close read end                      */
   } else {                   /* parent process                            */
      close(pipe_ends[0]);    /* close read end -> process wants to writen */
      write(pipe_ends[1],"hello",5);
      close(pipe_ends[1]);    /* done, close write end                     */
   }
}
```

Benannte Pipes (engl. named pipe)

- Pipe erscheint im Dateisystem, ist jedoch keine echte Datei
- Lesen & Schreiben auf eine benannte Pipe wie auf einer Datei (halbduplex)
- Beispiel:

```
% mkfifo  nmPip
% ls  -l >nmPip & more <nmPip
```

- Bemerkung: Named-Pipes erlauben auch mehrfache Leser und Schreiber:

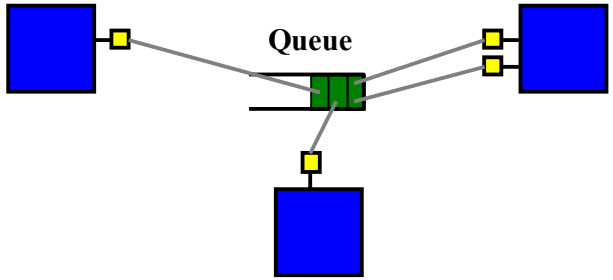


Beispiel: benannte Pipe

```
#include <fcntl.h>      #include <stdio.h>      #include <string.h>
#include <sys/stat.h>   #include <sys/wait.h>   #include <unistd.h>

int main(void) {
   char  *msg = "A message from mother process";
   char  *fifo = "myFifo";
   char  buffer[64];
   int   fd, status;

   mkfifo(fifo, 0666);          /* create named pipe -> read/write for everyone */
   pid_t pid = fork();

   if (pid > 0) {               /* parent                                       */
      fd = open(fifo, O_WRONLY);
      write(fd, msg, strlen(msg);
      wait( status );           /* wait until child terminates                  */
   } else {                     /* child                                        */
     fd = open(fifo, O_RDONLY);
     read(fd, buffer, 64);
     printf("%s\n", buffer);
   }
   close(fd);
   unlink(fifo);                /* delete special file                          */
}
```

12.6 Message Queues in UNIX

- Message Queue: **asynchroner Nachrichtenaustausch** (Briefkasten).
- Operationen:
  - Queue anfordern: `msgget(key, flags);`
    - flags: `IPC_CREAT` …
  - `msgsnd(…)`
  - `msgrcv(…)`



- Key: rechnerlokale Adresse (Integer) zur Identifikation eines Puffers:
  - Den Key müssen alle beteiligten Prozesse kennen
  - Erzeugen eines Keys beispielsweise mit: `key = ftok(path, projektID);`
    - `path`: referenziert existierende Datei
    - `projektID`: Projektnummer

Beispiel: Message Queue Sender (UNIX)

```c
#include <sys/ipc.h>     #include <sys/msg.h>
#include <stdio.h>       #include <string.h>

#define MSGSIZE       20      /* fixed message size  */
#define MQ_KEY      2404      /* hard-coded queue ID */

struct myMsg {
     long mtype;
     char mtext[MSGSIZE];
} dataMsg;

int main(int argc, char **argv) {
   int  msgID;
   long msgTyp = 0;

   dataMsg.mtype = atol( argv[1] );          /* argument 1: message type    */
   strncpy(dataMsg.mtext, argv[2], MSGSIZE); /* argument 2: message string */
   msgID = msgget(MQ_KEY, IPC_CREAT | 0666); /* 0666: readable & appendable by all */

   msgsnd(msgID, &dataMsg, MSGSIZE, 0);      /* MSGSIZE = size of mtext     */
   printf("data sent\n");
}
```

Beispiel: Message Queue Empfänger (UNIX)

```
#include <sys/ipc.h>     #include <sys/msg.h>
#include <stdio.h>       #include <string.h>

#define MSGSIZE       20      /* fixed message size  */
#define MQ_KEY      2404      /* hard-coded queue ID */

struct myMsg {
     long mtype;
     char mtext[MSGSIZE];
} dataMsg;

int main(int argc, char **argv) {
   int  msgID;
   long msgTyp = 0;

   msgTyp = atol(argv[1]);                  /* argument 1: message type */

   msgID = msgget(MQ_KEY, IPC_CREAT | 0666); /* 0666: readable & appendable by all */
   msgrcv(msgID, &dataMsg, MSGSIZE, msgTyp, 0); /* blocking receive */
   printf("data received: %s\n", dataMsg.mtext);
}
```

12.7 Sockets (UNIX-Domain)

- **Socket**: allgemeine Kommunikationsendpunkte.
  - bidirektionale gepufferte Kommunikation (blockierend und asynchroner Modi)
- Auswahl einer **Protokollfamilie**
  - UNIX-Domain: IPC auf einem Rechner
  - Internet-Domain: TCP bzw. UDP
- UNIX-Domain Sockets
  - Prozesse müssen nicht verwandt sein
  - **Adressierung durch Pfad + Datei**
    → Spezialdatei für socket (`ls -l` → Typ='s').
  - Adressformat in `sys/un.h`

```
struct sockaddr_un {
   /* Address family, hier AF_UNIX */
   unsigned short sun_family;

   /* Address Data : Dateipfad */
   char sun_data[108];
};
```

Beispiel: Client

```
#include <sys/un.h>       #include <sys/socket.h>
#include <stdio.h>        #include <string.h>

#define UNIXSTR_PATH   "mySock"    /* file sock path                  */

int main(int argc, char **argv) {
   int                    cSock, lenAddr;
   struct sockaddr_un     servAddr;
   char                   *buffer = ”hello world”; /* we just send a hello world     */

   cSock = socket(PF_UNIX, SOCK_STREAM, 0);       /* 0 = standard protocol = stream */

   bzero(&servAddr, sizeof(servAddr));            /* init struct -> set me to 0     */
   servAddr.sun_family = AF_UNIX;                 /* set address family             */
   strcpy(servAddr.sun_path, UNIXSTR_PATH);       /* set file path                  */
   lenAddr = sizeof(servAddr.sun_family) + strlen(servAddr.sun_path);

   connect(cSock, (struct sockaddr ) &servAddr, lenAddr);
   send(cSock, buffer, strlen(buffer), 0); /* socket, data, byte len, flags=default */
   close(cSock);
}
```

Beispiel: Server

```
#include <sys/un.h>      #include <sys/socket.h>
#include <stdio.h>       #include <string.h>

#define UNIXSTR_PATH   "mySock"    /* file sock path                  */

int main(int argc, char **argv) {
   int                   lSock, cSock, lenAddr, numrcv;
   socklen_t             clilen;
   struct sockaddr_un    cliAddr, servAddr;
   char                  buffer[BUFF_SIZE];

   lSock = socket(PF_UNIX, SOCK_STREAM, 0);       /* 0 = standard protocol = stream */

   bzero(&servAddr, sizeof(servAddr));            /* init struct -> set me to 0     */
   servAddr.sun_family = AF_UNIX;                 /* set address family             */
   strcpy(servAddr.sun_path, UNIXSTR_PATH);       /* set file path                  */
   lenAddr = sizeof(servAddr.sun_family) + strlen(servAddr.sun_path);
   clilen  = sizeof( cliAddr );
   bind(lSock, (struct sockaddr *) &servAddr, lenAddr); /* bind to listener socket  */
   listen(lSock, 5);                          /* allow max. 5 pending connections    */
   cSock = accept(lSock, (struct sockaddr *)&cliAddr, &clilen);  /* block           */
   numrcv = recv(cSock, buffer, BUFF_SIZE, 0);    /* receive on cSock data          */
   close(cSock); close(lSock);
}
```

12.8 Shared-Memory (UNIX-Domain)

- Schnellste Möglichkeit der Inter-Prozess-Kommunikation
- Vermeidet Umkopieren von Daten
- Sharing von Datenstrukturen mit Zeigern
  - Sofern Speicherbereich jeweils an gleicher Adresse eingeblendet wird
  - Dann entfällt auch De-/Serialisierung von Datenstrukturen, wie sie bei Sockets und Pipes notwendig ist

- Anzeigen von Shared-Memory-Bereichen mit dem Befehl `ipcs`

Shared-Memory-Funktionen

- Anlegen: `shmget(key, size, flags)`
  - `key`: zur Identifikation des Shared-Memory zwischen Prozessen
  - `size`: Größe des Bereichs
  - `flags`: `IPC_CREAT`, Zugriffsberechtigung (9-Bits)
- Freigeben: `shmdt(ptr)`
- Einbinden: `ptr = shmat(key, addr, flags)`
  - `key`: zur Identifikation des shared memory zw. Prozessen
  - `addr`: 0 oder Speicheradresse vorschlagen
  - `flags`: 0 oder `SHM_RDONLY`
- Zerstören: `shmctl(key, IPC_RMID, flags)`
  - `flags`: Zeiger auf Struktur (Ownership, Zeitstempel etc.)

Beispiel: 1. Prozess (erzeugt Shared-Memory)

```
#include <sys/ipc.h>     #include <sys/shm.h>
#include <stdio.h>       #include <string.h>

#define MAXMYMEM    30
#define SHM_KEY     0x964

int main(int argc, char **argv) {
   int  shID;
   char *myPtr;
   int  i;

   shID  = shmget(SHM_KEY, MAXMYMEM, IPC_CREAT | 0666);
   myPtr = shmat(shID, 0, 0);           /* map shared memory (key, addr, rw)    */

   for (i=0; i<MAXMYMEM; i++)
      myPtr[i] = 'A'+i;

   printf("data written in shared memory, waiting for key\n");
   getchar();

   shmdt(myPtr);                        /* release shared memory mapping        */
   shmctl(shID, IPC_RMID, 0);           /* delete shared memory mapping         */
}
```

Beispiel: 2. Prozess (greift auf Shared-Memory zu)

```
#include <sys/ipc.h>      #include <sys/shm.h>
#include <stdio.h>        #include <string.h>

#define MAXMYMEM    30
#define SHM_KEY     0x964

int main(int argc, char **argv) {
   int  shID;
   char *myPtr;
   int  i;

   shID  = shmget(SHM_KEY, MAXMYMEM, IPC_CREAT | 0666);
   myPtr = shmat(shID, 0, 0);           /* map shared memory (key, addr, rw)    */

   for (i=0; i<MAXMYMEM; i++)
      putchar(myPtr[i]);
   puts("\n");

   shmdt(myPtr);                        /* release shared memory mapping        */
}
```

12.9 Hörsaal-Aufgabe

- Ziel: Datenaustausch zw. Eltern-Kind-Prozess mithilfe von Shared-Memory
- Erzeugen Sie mit `fork()` einen Kindprozess
- Der Kindprozess soll einen Shared-Memory-Bereich einblenden und einen String in diesen hineinkopieren
- Der Elternprozess:
  - Wartet mit `waitpid` auf die Terminierung des Kindprozesses
  - Und liest danach die Daten aus dem Shared-Memory-Bereich und gibt diese auf dem Bildschirm aus