## Datenaustausch zwischen Prozessen

Man muss unterscheiden wo sich die Prozesse befinden und wie sie voneinander abhängen:

1. Parallel ablaufende Teilprozesse die gemeinsamen Speicher besitzen (threads)
2. Prozesse am gleichen System, die miteinander verwandt sind aber jeweils einen eigenen Speicherbereich besitzen (unter Linux mit fork erzeugt, bei Windows gibt es fork nicht)
3. Prozesse am gleichen System, die nicht miteinander verwandt sind und von Haus aus einen eigenen Speicherbereich besitzen (einfach zwei getrennte Tasks)
4. Prozesse auf getrennten Systemen, wobei die Systeme über ein Netzwerk verbunden sind

Erzeugung von getrennten Prozessen am selben Rechner   
mit fork oder threads oder man startet gleich zwei verschiedene Programme

## Fork (verwandte Prozesse)

Für UNIX Systeme, nicht unter Windows.

Fork teilt einen Prozess in 2 Teilprozesse (Eltern und Kindprozess). Kind beendet sich nach erledigter Arbeit selbst mit exit. Wenn das Kind seinen exit Status nicht meldet, bleiben Ressourcen im OS belegt, bis der Elternprozess den Status des Kindes abruft. In der Zeit zwischen beenden und Status abrufen ist das Kind ein Zombie.

Eltern und Kindprozess haben getrennte Speicherbereiche, die aber identische Kopien des Speichers zum Zeitpunkt der Teilung sind.

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <ipc.h>

#define MAX\_COUNT 200

#define BUF\_SIZE 100

void main(void)

{

pid\_t pid;

int i;

char buf[BUF\_SIZE];

int status;

pid = fork();

if (pid==0) … Kind

{

…. Befehle, die das Kind ausführen soll

exit(status);

}

elseif (pid>0)

{

wait(&status);

}

if (pid>0) … Elternprozess, in pid steht die ID des Kindes

if (pid<0) … hat nicht funktioniert

}

C Beispiel einer Fork.

Elternprozess zB Warteschlange für Aufträge

while(1)

{

while (nichtszutun);

Variablen setzen, die die Arbeit des Kindes definieren

pid=fork();  
 if(pid==0)

{

Arbeitsaufträge des Kindes

exit(0); //Elternprozess über Beendigung informieren

}

if(pid>0) waitpid(pid\_des\_kindes, &status);

//auf exit des Kindes warten oder einfacher wait(&satus):

}

## Thread

Linux und Windows

Thread ist eine Ausführung eines Tasks. Ein Task kann mehrere Threads haben. Memory Objects werden von den Threads geteilt. Die Threads haben keinen getrennten Speicherbereich, deshalb muss man sich Gedanken über die Verwendung von gemeinsamen Ressourcen machen, damit kein Chaos entsteht (Problem des schlafenden Frisörs, Lösung mit lock, mutex, semaphoren, synchronize, volatile,…)

# IPC Kommunikation zwischen Prozessen

Austausch von Informationen zwischen Prozessen die keinen gemeinsamen Speicherbereich haben.

Möglichkeiten sind:

Unterscheidung:

Am gleichen System (verwandt oder nicht verwandt)

* Files
* Pipes (named (fifos) für nicht verwandte Prozesse oder unnamed pipes für verwandte)
* Shared Memory

Auf unterschiedlichen Systemen

* Files (nfs, samba, ftp)
* Sockets
* Datenbank

#### Files

Langsam, es gibt Schutzmechanismen also wer darf wann schreiben, evtl. Verschlüsselung. Ist auch im Netzwerk möglich. Trotzdem muss man sich ein Protokoll über den Ablauf überlegen zB ein zweites file mit Infos wer gerade was darf oder gemacht hat. Geht auch über Netzwerkfreigaben.

#### Shared Memory

Bereich auf den mehrere Prozesse zugreifen können. OS verwaltet einen Bereich der für alle zugänglich ist, wie Files nur schneller.

Task 1

#include <sys/ipc>

#include <sys/shm>

#include <sys/types.h>

#include <unistd.h>

void main(){

int shmid;

key-t key;

char \*shm;

key = 2222 //eindeutig

if(shmid = shmget(key, 100, IPC\_CREAT(0666))<0)

{

printf(“Fehler beim Anlegen”);

return 1;

}

shm=(char\*) shmat(shmid,NULL,0); //wie malloc

strcpy(shm,”Hallo”);

while(\*shm != ‘\*’) sleep(1);

return 0;

}

key = Konstante IPC\_PRIVATE

oder key = unique integer

Der Wert 0660 besagt dass alle Schreiben und Lesen dürfen.

Task 2

#include <sys/ipc>

#include <sys/shm>

#include <sys/types.h>

#include <unistd.h>

void main(){

int shmid;

key-t key = 2222;

char \*shm;

shmid = shmget(key,100,0666);

shm = (char\*)shmat(shmid,NULL,0);

printf(“%s”,shm);

strcpy(shm,”\*”);

return 0;

}

Für Schutzmechanismen (mutex, lock, semaphore,…) ist der Programmierer zuständig

Unter Windows wird shared memory “memory mapped file” genannt. Jeder, der den Namen des Files kennt, kann es verwenden, also auch nicht verwandte Tasks. Memory mapped files gibts persisted und non persisted:

Persisted: dahinter steckt ein echtes File, das in den Speicher geladen wird, dadurch kann man sehr schnell darauf arbeiten. Beim beenden wird das File wider aus dem memory auf die Festplatte persistiert

Non persisted: das File existiert nur im Speicher.

Zugriffe erfolgen, in dem ein stream mit reader oder writer erzeugt wird. In C# elegant gelöst.

#### Pipes

**Named Pipes:**

In der Konsole:

mkfifo nameDerPipe

Terminal 1 Terminal 2  
mkfifo mypipe cat < mypipe  
Befehl > mypipe

Gegenseitiges Blockieren ist wichtig.

Terminal 1 wartet bis Terminal 2 die pipe gelesen hat  
Terminal 2 wartet bis Terminal 1 etwas reingeschrieben hat  
nach dem Lesen ist die pipe leer.   
Das Ganze ist ziemlich praktisch, weil man sich durch das automatische Blockieren nicht mehr selber um den Ablauf kümmern muss.

In C:

Befehl ***mkfifo(char \*);*** wobei die Zeichenkette den Pfad und den Namen der pipe enthält.  
Danach schreiben und lesen wie bei files, aber wieder mit automatischem Blockieren, um das man sich nicht selber kümmern muss. Geht auch für nicht verwandte Prozesse, wenn sie den Namen der pipe kennen.

**Unnamed Pipes**im Terminal:

| … unnamed Pipe. Befehl 1 | Befehl2

Ausgabe des ersten Befehls wird dem 2ten Befehl als Parameter mitgegeben.

In C:

Nur für verwandte Prozesse (als Beilage zum Erklären bei der mRDP, ohne Kommentare)

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

void main(){

int fd[2];

pid\_t childpid;

char string[] = "Hello, world!\n";

char readbuffer[80];

pipe(fd);

if((childpid = fork()) == -1)

{

perror("fork");

exit(1);

}

if(childpid == 0)

{

/\* Child process closes up input side of pipe \*

close(fd[0]);

/\* Send "string" through the output side of pipe \*/

write(fd[1], string, (strlen(string)+1));

exit(0);

}

else

{

/\* Parent process closes up output side of pipe \*/

close(fd[1]);

/\* Read in a string from the pipe \*/

nbytes = read(fd[0], readbuffer, sizeof(readbuffer));

printf("Received string: %s", readbuffer);

}

return(0);

}

fd[0] = Input Seite der Pipe

fd[1] = Output Seite der Pipe

Unter Windows gibt es pipes ebenfalls, Logik dahinter ist gleich: CreatePipe, GetNamedPipe, WaitPipe usw.

**Sockets**

noch ein paar Worte zu sockets (API für TCP/IP Verbindungen)

Für Tasks auf verschiedenen Rechnern (Netzwerkverbindung)

Ablauf

Server:

Socket erstellen

Binden an einen Port

Listen (Warten auf eine Verbindungsanfrage)

Accept (Einkommende Anfrage akzeptieren)

read/write wie bei Files

Client:

Socket erstellen

Connect an IP Adresse des Servers:Port

Warten auf accept

read/write wie bei Files

Bedeutung des Ports:  
Legt fest welche Anwendung für die Auswertung der Daten verantwortlich ist  
Well known ports (kleine Nummern unter 1024)

Port 80 (http -> Browser ist verantwortlich)

Port 22 (ftp - >ftpClient zb Filezilla)

Rest für eigene Verwendung, alle Teilnehmer müssen das Protokoll kennen, damit sie Daten beim Senden entsprechend „verpacken“ und beim Empfangen „entpacken“ können bzw damit sie überhaupt wissen wo und wann ein Datenaustausch beginnt und endet.

## Problem des „schlafenden Frisörs“

Tritt auf, wenn Prozesse abwechselnd ausgeführt werden und vom scheduler zur falschen Zeit unterbrochen werden:

Die Prozesse haben jeweils zwei Aufgaben:

1. Information sammeln
2. Reagieren auf die Information

Probleme entstehen, wenn der Prozess nach dem Sammeln der Information unterbrochen wird und erst später darauf reagieren kann und sich die Lage inzwischen verändert hat:

Frisör:

1. Information sammeln: Schaut ob eine Kunde da ist
2. Reaktion: Wenn keine Kunde da ist, schlafen gehen und warten bis er geweckt wird

Kunde:

1. Information sammeln: Schauen ob Frisör schläft
2. Reaktion: Wenn er der erste Kunde ist den Frisör aufwecken sonst warten bis man er dran kommt

Problematischer Ablauf weil die Prozesse unterbrochen werden und der jeweils andere Prozess die CPU Ressourcen bekommt

Es ist gerade kein Kunde im System

1. Frisör schaut ob Kunde da ist: Keiner da – Prozess wird unterbrochen
2. Kunde schaut ob Frisör schläft: Frisör ist munter – Prozess unterbrochen
3. Frisör reagiert auf seine letzte Information (kein Kunde da) und geht schlafen – Prozess unterbrochen
4. Kunde reagiert auf seine letzte Information (Frisör ist munter) und wartet bis er dran kommt

Jetzt schläft der Frisör und der Kunde wartet bis er dran kommt –> dead lock  
Auch ein weiterer Kunde würde den Frisör nicht aufwecken weil er ja nicht der erste Kunde ist.  
Lösung atomare Prozesse: = Verbieten, dass Prozesse an solchen kritischen Stellen unterbrochen werden können. Schlüsselwort mutex oder synchronize oder lock,….