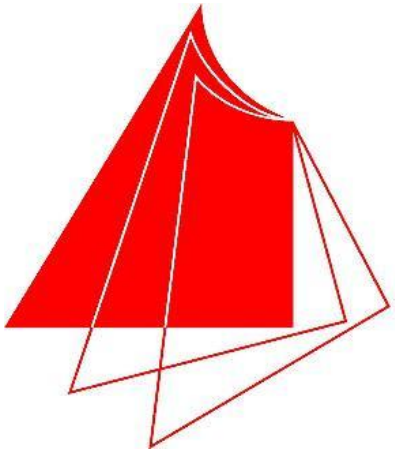
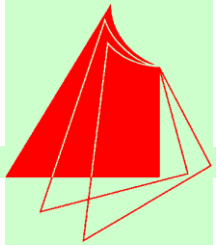


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

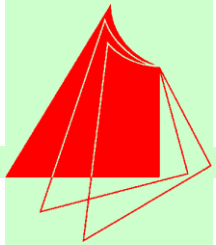
Labor Systemprogrammierung

Prof. Dr. Thomas Fuchß
Fakultät für Informatik und Wirtschaftsinformatik –
Fachgebiet Informatik



Übersicht

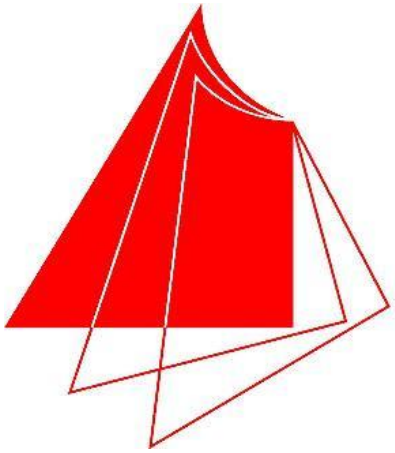
- **Allgemeines**
- **Teil I Lexikalische Analyse**
- **Teil II Syntaktische Analyse**
- **Teil III Prozesssynchronisation und Prozesskommunikation**



Allgemeines

- **Veranstaltungen**
  - jeweils mittwochs von 14.00 – 18.50 Uhr (LI 137)
  - Vorbesprechung der nächsten Aufgabe jeweils mittwochs um 14.00 Uhr im Seminarraum E 201

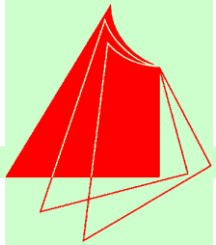
    Termine: 7.10., 11.11., 23.12.
- **Zeitplan**
  - Phase I 5 Termine (07.10. – 04.11.)
  - Phase II 6 Termine (11.11. – 16.12.)
  - Phase III 4 Termine (23.12. – 27.01.)
- **Werkzeuge und Sprachen**
  - C, C++



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

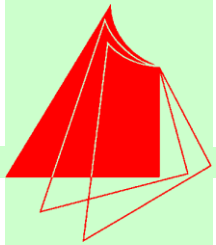
Systemprogrammierung
Teil III
Prozesssynchronisation und Prozesskommunikation

Prof. Dr. Thomas Fuchß
Fakultät für Informatik und Wirtschaftsinformatik –
Fachgebiet Informatik



Literatur

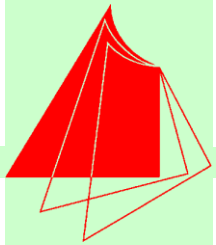
- Samuel J.Leffler.
  Das 4.3 BSD Unix Betriebssystem Daten und Design –
  Addison Wesley, 1989
- Jürgen Gulbins.
  Unix Version 7, bis System V.3 – Springer, 1988.
- Tim O'Reilly.
  Das BSD Unix Nutshell-Buch – Addison Wesley, 1989.
- J.Anton Illk.
  Programmieren in C unter Unix – Sybex, 1990.
- Andrew S. Tanenbaum.
  Betriebssysteme, Entwurf und Realisierung Teil 1 – Hanser, 1990.
- Marc J.Rochkind.
  Advanced Unix programing – Prentice Hall, 1985
- Eric Foxley.
  Unix für SuperUser – Addison Wesley 1988



Prozesse in UNIX

- **Umgebung (Environment):**
  Der Kontext eines ausführbaren Programms. Sie wird durch die Umgebungsvariablen (Umgebungsvektor) beschrieben.
  - das Home-Verzeichnis durch die Shell-Variable HOME,
  - die Standard-Suchpfadnamen durch die Shell-Variable PATH,
  - den Terminal-Typ durch die Shell-Variable TERM, ...

  Daneben sind auch alle offenen Dateien und Pipes, das aktuelle Verzeichnis, die Prozessgruppen-ID und die Benutzerkennung Bestandteil der Umgebung, ohne jedoch im Umgebungsvektor sichtbar zu sein.
- **Image:**
  Die Vereinigung von einem ausführbarem Objekt (Programm) und einer Umgebung.
- **Prozess:**
  Die Ausführung eines Images.



Prozesse in UNIX

- **Prozesshierarchie:**

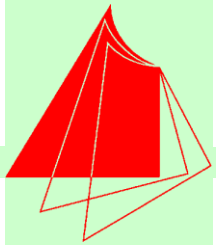
  Abgesehen vom ersten Prozess können Prozesse nur durch den Systemaufruf *fork()* als Kopie eines anderen Prozesses entstehen. Prozesse bilden eine Hierarchie

- **Prozessnummern (process-ID):**

  Jeder Prozess besitzt eine Prozessnummer, eine positive ganze Zahl

- **Väter und Waise:**

  Jeder Prozess kennt die Process-ID seines Vaters (Parent-Process-ID). Stirbt der Vater, so wird die Parent-Process-ID auf 1 gesetzt. Der Initialisierungsprozess adoptiert die Waise.

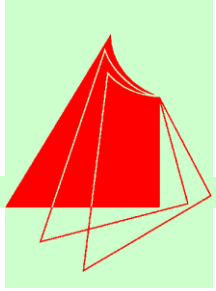


Prozessgruppen (process group)

Prozessgruppen sind Gruppen von miteinander verwandten Prozessen, die gemeinsam Aufgaben bearbeiten. Jede Gruppe hat einen Group-Leader, und jedes Mitglied hat dessen Prozessnummer als Prozessgruppennummer (Process-Group-ID). Über einen Systemaufruf kann an jedes Mitglied einer Gruppe ein Signal geschickt werden.

Ein Prozess kann aus einer Prozessgruppe ausscheiden und Leiter einer neuen Gruppe werden (Prozessnummer wird zur Gruppennummer *setpgrp()*). Durch das Abspalten von Sohn-Prozessen entsteht die neue Gruppe.

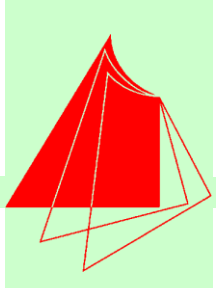
Hierdurch können applikationsspezifische Hierarchien gebildet werden.



Prozesse generieren, synchronisieren und beenden

- **generieren; fork():**
  Der Systemaufruf *fork()* spaltet einen aktiven Prozess in zwei voneinander unabhängige (asynchron) parallel ablaufende Prozesse auf, indem er vom aktiven Prozess (dessen Image) eine Kopie erzeugt. Eröffnete Dateien werden vererbt und haben einen gemeinsamen Lese-Schreib-Zeiger.

```
if ((pid = fork()) < 0) {
   /* Fehlerbehandlung:  Swap-Bereich zu klein, Prozesstabelle ist belegt, ... */
}
else if (pid > 0) { /* Vaterprozess  pid == process-ID des Tochterprozesses*/
   Aufgaben_des_Vaterprozesses
} else { /* Tochterprozess */
   execl(programmname, arg_1, arg_2, ... , arg_n, NULL); /* Programmwechsel */
}
```

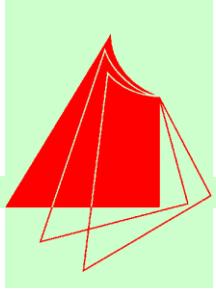


Prozesse generieren, synchronisieren und beenden

- **synchronisieren innerhalb von Prozessgruppen**
  - **process_id = wait(&status);**
    Der *wait()*-Systemaufruf versetzt den Vaterprozess in einen Wartezustand, bis ein Signal eintrifft oder einer seiner Tochterprozesse terminiert. In der ganzzahligen Variable *status* wird der Exit-Code des terminierten Prozesses, sowie die Nummer des Signals, das den Tochterprozess beendete abgelegt.

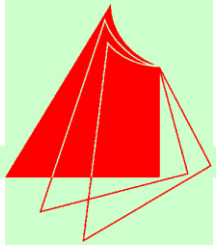
    status Byte 1 ist der Exit-Code Byte 0 die Signal-Nummer

  - **Signale SIGTERM, SIGUSR1, SIGUSR2 (u.a.)**
    Die Signale SIGUSR1 (Nr. 16) und SIGUSR2 (Nr. 17) werden nicht vom System verwendet und stehen somit dem Benutzer zur Prozess-synchronisation zur Verfügung. **Signale unterbrechen den empfangenen Prozess und erschweren somit (unnötig) das Programmieren.**
    (Empfangen wenn bereit)
- **Semaphore** sind sogenannte Flaggen, sie dienen der Zugriffsregelung auf Betriebsmittel und sollen Verklemmungen vermeiden.



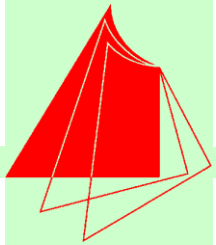
Prozesse generieren, synchronisieren und beenden

- **Prozesse beenden**
  - **void exit (int status);**
    exit() beendet einen Prozess und schließt alle dem Prozess gehörenden Dateien (Dateipuffer werden noch geschrieben). Falls der aufrufende Prozess mittels *wait()* wartet, wird dieser verständigt. Die niederwertigen acht Bits des Arguments *status* werden dem aufrufenden Prozess (Vaterprozess) übergeben.
  - **void _exit (int status);**
    _exit() beendet einen Prozess, ohne die Dateipuffer auf die Platte zu schreiben.



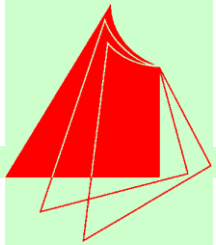
Prozesskommunikation

- **Gemeinsam benützte Positionszeiger** (selten)
  Es wird ein Positionszeiger einer Datei von einem Prozess gesetzt und von einem anderen gelesen. Die Position dient dann als übermitteltes Datum.
- **Prozess-Tracing** (kompliziert)
  Der Vaterprozess kann die Ausführungen seines Sohnes kontrollieren, d.h er kann dessen Daten lesen und schreiben.
- **Message-Queues (Nachrichten-Schlangen)**
  Jeder Prozess kann Nachrichten aus der Message-Queue empfangen, vorausgesetzt er hat die Zugriffsrechte dafür.
- **Dateien**
  Sie sind ein verbreitetes Kommunikationsmittel zwischen Prozessen. Ein Prozess schreibt auf eine Datei, die ein anderer liest. Schwierig, wenn beide gleichzeitig laufen.



Prozesskommunikation

- **Pipes** (lösen das Synchronisationsproblem der Dateien)
  Die Pipe selbst ist keine Datei. Sie besitzt aber eine I-Node. Das Lesen von und das Schreiben auf eine Pipe funktioniert ähnlich wie bei Dateien. Die Synchronisation wird aber der Pipe überlassen.
  Nachteile:
  - Kommunikation mittels Pipes ist nur bei verwandten Prozessen möglich.
  - die atomare Ausführung der Lese-Schreiboperationen ist nicht garantiert
  - Pipes sind langsam.
- **FIFO-Dateien (named pipe)**
  Auch fremde Prozesse können über FIFO's miteinander kommunizieren. FIFO's sind wie Pipes langsam.
- **Shared-Memory** (gemeinsamer Speicher)
  Wohl die schnellste Art der Prozesskommunikation unter UNIX. Es werden Daten von mehreren Prozessen in einen gemeinsamen Speicher geschrieben, wobei der Zugriff über Semaphore oder Messages synchronisiert wird.

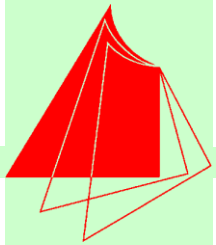


Message-Queues

Unter einer Nachrichtenwarteschlange versteht man eine Einrichtung, an die ein Prozess (Produzent) eine Nachricht sendet, die dann ein anderer Prozess (Konsument) zu einem späteren Zeitpunkt abholt.

Mehrere Produzenten können gleichzeitig an ein und dieselbe Nachrichtenwarteschlange Nachrichten senden und mehrere Konsumenten können gleichzeitig von dieser einen Warteschlange Nachrichten abholen. Auf diese Weise können über eine einzige Nachrichtenwartenschlange zahlreiche kooperierende Prozesse miteinander kommunizieren.

Warteschlangen sind reihenfolgetreu



Message

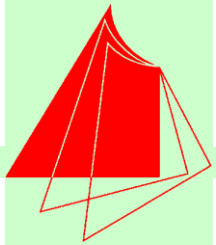
Bei Messages handelt es sich um kleine Portionen von Daten, die zwischen Prozessen ausgetauscht werden.

Eine Nachricht besteht aus einem Nachrichtentext und einem Nachrichtentyp.

Nachrichtentypen haben keine feste Bedeutung, jeder Produzent kann Nachrichten verschiedener Typen an ein und dieselbe Nachrichtenwarteschlange senden. Jeder Konsument kann nun eine Nachricht aus einer Message-Queue anfordern (Typangabe optional)

Ein Prozess kann beim Anfordern einer Nachricht angeben,

- ob er suspendiert werden will, sofern noch keine Nachricht vorliegt, oder
- ob er durch einen Fehlercode informiert werden will.



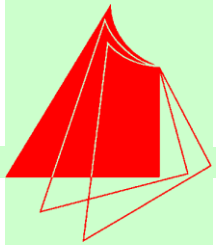
Arbeiten mit Messages

1. **Warteschlange anlegen (msgget())**
   Die Warteschlange muss als erstes definiert werden, d.h. durch die Definition wird die Warteschlange durch einen Datensatz, den Queue-Header, im System repräsentiert.

2. **Warteschlange einem Prozess verfügbar gemacht (msgget())**
   Jedem Prozess, der eine Warteschlange benutzen will, muss diese bekannt gemacht werden.

3. **Nachrichten in die Warteschlange schreiben (msgsnd())**

4. **Nachrichten aus der Warteschlange lesen (msgrcv())**



Warteschlange anlegen

Eine Warteschlange wird über die Funktion *msgget()* definiert oder bekannt gemacht. Dabei wird die Message-Queue mit einem „externen“ Namen (Key) und mit Zugriffsrechten (*Flag*) versehen und an einen „internen“ Namen gebunden (*id*). Über diesen hat der Prozess Zugriff auf die Queue. Es können nun Messages in die Queue eingetragen (*msgsnd()*) oder aus dieser gelesen (*msgrcv()*) werden.

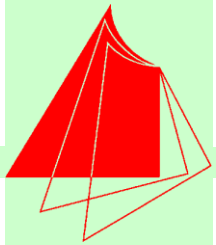
- **Parameter Key, Name der Warteschlange**
- **Parameter Flag, Zugriffsrechte auf die Queue in oktaler Form angegeben.**
- **IPC_CREAT zeigt an, dass die Queue erzeugt werden soll.**

Beispiel: Anlegen einer Queue mit dem Namen „12“

```
key = 12;
msqid = msgget(key, 0664 | IPC_CREAT);
```

Beispiel: Anfordern einer Queue mit dem Namen „12“
(Gibt es die Queue noch nicht, so liefert *msgget()* einen Fehler zurück)

```
key = 12;
msgget(key, 0664);
```



Aufbau von Nachrichten

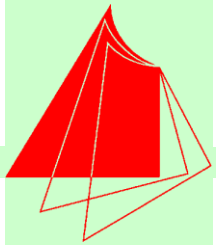
Messages können die verschiedensten Bedeutungen und Inhalte haben, ihr Aufbau im IPC-Konzept unter UNIX wird durch zwei Punkte charakterisiert:

- **den Typ der Message und**
- **den Text der Message.**

Über den Typ der Message können z.B. Dringlichkeitsstufen realisiert werden.

Senden von Nachrichten: **int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflag);**

- **msqid:**
  Interner Name der Queue
- **msgp:**
  Die eigentliche Message, ein Pointer auf eine Variable vom Typ „*struct msgbuf*“. (Typ und Text)
- **msgsz:**
  Die Länge des Messagetextes
- **msgflag:**
  Flags zur Steuerung von *msgsnd()*



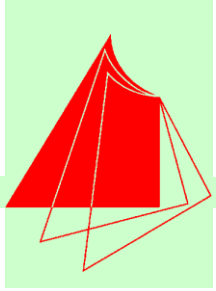
Nachrichten lesen

Mit der Funktion *msgrcv()* kann ein Prozess lesend auf eine Queue zugreifen.

Lesend bedeutet dabei:
Die Funktion *msgrcv()* wählt eine Nachricht nach bestimmten Kriterien aus, kopiert sie in den lokalen Adressraum und löscht sie aus der Queue.

ssize_t = msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflag);

- **msqid:** Der interne Name der Queue.
- **msgp:** Pointer auf eine Variable vom Typ „*struct msgbuf*“. In diese Variable wird der Typ der Message und der ihr zugehörige Text kopiert.
- **msgsz:** Maximal gelesene Zeichen des Nachrichtentextes (ist die Nachricht länger, so wird die Funktion abgebrochen und keine Message aus der Queue gelesen. (die Konstante *MSG_NOERROR* verhindert den Abbruch
- **msgtyp:** Typ der Nachricht, die gelesen werden soll.
  - typ == 0, die erste Message in der Queue wird gelesen.
  - typ > 0, Typ der Message, erste Message dieses Typs wird gelesen.
  - typ < 0, Die erste Message mit msgType < |typ| wird gelesen.
- **msgflag:** Zugriffsrechte und Steuerparameter

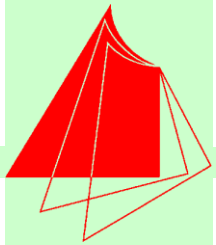


Queues verwalten msgctl()

Über die Funktion *msgctl()* können Zugriffsrechte und Besitzverhältnisse (owner und group) verändert und die gesamte Queue gelöscht werden. Außerdem kann der Status einer Queue („*struct msqid_ds*“) abgefragt werden.

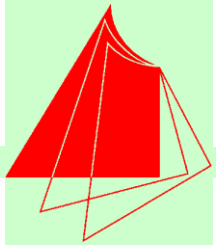
msgctl(int msqid, int com, struct msqid_ds *buf)

- msqid: Die zu verwaltende Queue
- com: Das Kommando bestimmt, was die Funktion msgctl() machen soll.
  - IPC_STAT, die Informationen über die Queue, die im Kopf der Oueue abgespeichert sind, werden in der Variablen msgq qespeichert.
  - IPC_SET, Zugriffsrechte und Besitzverhältnisse können verändert werden, die neuen Werte werden mittels *buf* übergebenden.
  - IPC_RMID, die Queue wird gelöscht.
- buf: Daten die den Queue-Header beschreiben oder in die Werte des Queue-Headers geschrieben werden.



Datenstrukturen

```
struct msg{
   struct msg *msg_next;     /* Pointer auf nächstes Element    */
   long msg_type;            /* Typ der Message                 */
   short msg_ts;             /* Länge des Textes                */
   short msg_spot;           /* Verweis auf Map-Bereich         */
 };
struct msqid_ds{
   struct ipc_perm msg_perm; /* Zugriffsrechte                      */
   struct msg *msg_first;    /* Pointer auf erste Message           */
   struct msg *msg_last;     /* Pointer auf letzte Message          */
   ushort msg_cbytes;        /* Anzahl Bytes in der Queue           */
   ushort msg_qnum;          /* Anzahl Messages in der Queue */
   ushort msg_qbytes;        /* max. Anzahl Bytes in der Queue      */
   ushort msg_lspid;         /* PID von letztem msgsnd()            */
   ushort msg_lrpid;         /* PID von letztem msgrcv()            */
   time_t msg_stime;         /* letzter Zugriff von msgsnd()        */
   time_t msg_rtime;         /* letzter Zugriff von msgrcv()        */
   time_t msg_ctime;         /* letzte Änderung                     */
};
struct msgbuf {
   long mtype;       /* Typ der Message  */
   char mtext[1];    /* Text der Message */ };
```



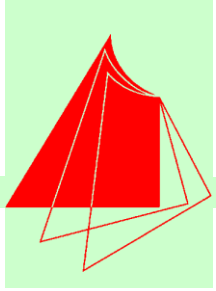
Shared-Memory

Shared-Memory dient dem Austausch von beliebigen Daten zwischen zwei oder mehreren Prozessen. Es kann sich dabei um beliebige Prozesse handeln (auch von verschiedenen Benutzern gestartet).

Shared-Memories sind Bereiche im Hauptspeicher, die nicht im Adressbereich eines speziellen Prozesses liegen, sondern global sind, und auf die beliebige Prozesse zugreifen und die dort abgelegten Daten bearbeiten können.

Arbeit mit Shared-Memory:

1. Das Shared-Memory vom Betriebssystem anfordern (oder anlegen) und dem Prozess mit der Funktion *shmget()* verfügbar machen.
2. Das Shared-Memory mit *shmat()* in den Adressraum des aktuellen Prozesses einbinden (einen Pointer auf den Speicherbereich)
3. Mit beliebigen Funktionen auf diesen HSP-Bereich operieren (lesen oder schreiben).
4. Das Shared Memory mit *shmdt()* aus dem Adressbereich herauslösen.

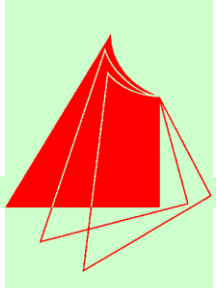


Shared-Memory anfordern bzw. anlegen

Ein Shared-Memory besteht aber nicht von vornherein, sondern muss irgendwann angelegt werden.

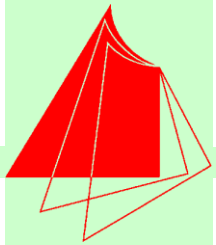
- **Shared-Memory anlegen int shmget(key_t key, int size, int shmflg)**
  - Der erste Parameter ist ein frei zu vergebender Name
  - Als zweiter Parameter wird die Größe des Shared-Memory übergeben.
  - Mit dem dritten Parameter werden die Zugriffsrechte definiert und es wird angegeben, dass ein Shared-Memory neu angelegt werden soll (IPC_CREATE)
- **Zugriffsrechte und symbolische Konstanten**
  - 00X00 Rechte Besitzer (4 Lesen 2 Schreiben)
  - 000X0 Rechte Gruppe
  - 0000X Rechte Andere
  - 01000 IPC_CREAT
  - 02000 IPC_EXCL

```
shm_id = shmget(12, 5000, IPC_CREAT | 0644);
```



Shared-Memory an- und abkoppeln

- **Ankoppeln an den Adressraum**
  void *shmat (int shmid, const void *shmaddr, int shmflg)
  Nachdem ein Shared-Memory mit *shmget()* einem Prozess bekannt gemacht wurde, kann es mit der Funktion *shmat()* in das Datensegment des virtuellen Adressraumes des Prozesses eingebunden werden. Das Shared Memory ist für diesen Prozess damit lokal verfügbar.(Vergleich bar einem Char-Array)
  - Rückgabe einer Adresse über die auf die Daten des Shared-Memory zugegriffen werden kann.
  - Über *shmid* wird das Shared-Memory angesprochen. *shmid* ist die Kennung, die *shmget()* als Returnwert lieferte.
  - *shmaddr*, die Adressvorgabe. Eine 0 bedeutet, dass diese Adresse, von der UNIX-Prozessverwaltung freivergeben werden kann. (Sicher und empfehlenswert)
  - Der Parameter *shmflg* definiert den Zugriff. 0 bedeutet, dass auf das Shared-Memory lesend und schreibend zugegriffen werden kann.

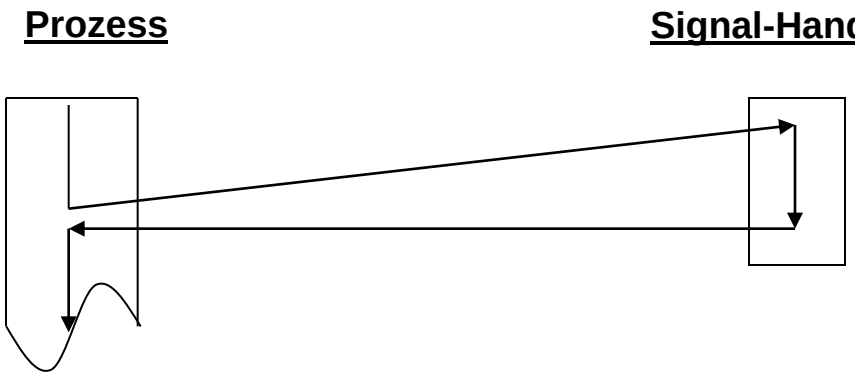


Shared-Memory an- und abkoppeln

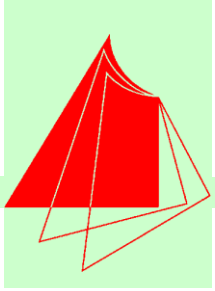
- **Abkoppeln aus dem Adressraum**
  int shmdt (const void *shmaddr)
  Wird das Shared-Memory nicht mehr benötigt, so wird es vom lokalen Adressraum abgekoppelt. Das Shared-Memory ist aber dem Prozess noch bekannt und kann zu einem späteren Zeitpunkt mit *shmat()* wieder eingebunden werden.
- **Kontrollieren eines Shared-Memorys**
  int shmctl(int shmid, int cmd, struct shmid_ds *buf)
  - Kommandos: IPC_STAT, IPC_SET, IPC_RMID (siehe Queues)
    SHM_LOCK und SHM_UNLOCK verbietet und erlaubt Swapping (nur für Root)

Signale

- **Signale ähneln Hardware-Interrupts oder Traps. Sie haben ihren Ausgangspunkt meist im Kernel, sie können aber auch von jedem andern Prozess gesendet werden.**
- **Signale enthalten keine Informationen.**
- **Signale können zu jedem beliebigen Zeitpunkt gesendet werden.**
- **Auf Signale muss nicht zwangsläufig reagiert werden. Ein Prozess kann durch explizite Angaben veranlasst werden, Signale bestimmten Typs zu ignorieren oder in gesonderten Routinen zu behandeln (sonst terminiert er).**

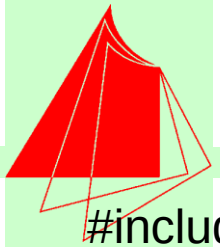


Programmfluss, bei abgefangenem Signal



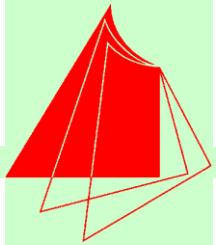
Übersicht der Systemaufrufe für den Umgang mit Signalen:

- **sighandler_t signal(int signum, sighandler_t handler)**
  typedef void (*sighandler_t)(int)
  Einem Signal die Behandlungsroutine zuordnen.
  Standard-Handler
  - SIG_IGN
  - SIG_DFL
- **unsigned int alarm(unsigned int seconds)**
  Weckalarm einstellen.
- **int pause(void)**
  Den Prozess anhalten, bis ein Signal eintrifft.
- **int kill(pid_t pid, int sig)**
  Ein Signal an einen Prozess senden.
  Besonderheiten pid == 0, pid == -1 und pid < -1
- **int setjmp(jmp_buf env)**
  Vorbereitung eines nicht lokalen Sprungs.
- **longjmp(jmpenv, val)**
  Nichtlokaler Sprung wird ausgeführt.



Beispiel

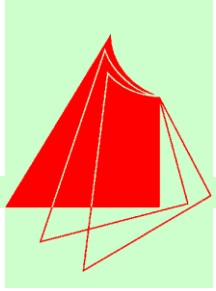
```
#include <signal.h>
#include <stdio.h>

void INThandler(int i) /* fängt Interrupt-Taste permanent ab */ {
   if(signal(SIGINT,INThandler)==SIG_ERR) perror("SignalFehler!");
   printf("Signal SIGINT abgefangen!\n");
 }

void QUIThandler(int i) { /* einmaliges abfangen, danach Standardreaktion */
   printf("Signal SIGQUIT abgefangen!\n");
}

main() {
   if(signal(SIGINT, INThandler)   == SIG_ERR) perror("SignalFehler!");
   if(signal(SIGQUIT, QUIThandler) == SIG_ERR) perror("SignalFehler!");

   printf("Das Signal INTERRUPT kann diesen Prozess nicht beenden!\n");
   printf("Das Signal QUIT wird nur einmal abgefangen!\n");
   for(;;);
}
```



Semaphore

Ein Semaphor ist ein Mechanismus, der zwei oder mehr Prozesse daran hindert,auf ein exklusiv benötigtes, gemeinsames Betriebsmittel (z. B. einen Drucker) gleichzeitig zuzugreifen.

- **binäres Semaphor**
  Ein binäres Semaphor hat nur zwei Zustände: blockiert (locked) und frei (unlocked).
- **allgemeines Semaphor**
  Ein allgemeines Semaphor hat unendlich viele (oder zumindest sehr viele) Zustände. Es ist ein Zähler, der bei Erwerb eines Semaphors (locked) um eins erniedrigt und bei Freigabe (unlocked) um eins erhöht wird. Hat das Semaphor den Wert Null, so muss ein Prozess, der es erwerben will, auf einen anderen Prozess warten, der es erhöht (es kann niemals negativ werden).



Der Umgang mit Semaphoren

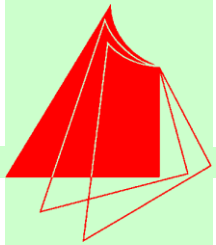
Auf ein Semaphor wird üblicherweise über zwei Funktionen zugegriffen

- die Funktion P (Erwerb)
- die Funktion V (Freigabe)

```
void P(int *sema) { /* Semaphor erwerben */
   while (*sema <= 0); /* nichts tun */
   (*sema)--;
 }

void V(int *sema) { /* Semaphor freigeben */
   (*sema)++ ;
}
```

Das Semaphor muss durch Aufruf der Funktion V initialisiert werden andernfalls startet das Semaphor ohne Erwerbsmöglichkeit!
Falls das Semaphor zum Beispiel die Anzahl freier Puffer zählt und am Anfang fünf davon existieren, so könnte man beginnen, indem man fünf Mal die Funktion V aufruft.

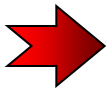
(alternativ kann man auch einfach die Semaphor-Variable auf fünf setzen).



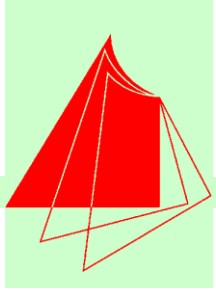
Der Umgang mit Semaphoren

Weshalb können die Funktionen P und V nicht im Adressraum des Benutzers programmiert werden?

- Die Semaphor-Variable, auf die "sema" verweist, kann nicht von mehreren Prozessen gemeinsam benutzt werden, die unterschiedliche Datensegmente haben.
- Die Funktionen werden nicht atomar ausgeführt, d.h. der Kern kann einen Prozess zu jedem beliebigen Zeitpunkt unterbrechen.
  - Prozess 1 beendet die while-Schleife in P und wird unterbrochen,bevor er das Semaphor herunterzählen kann.
  - Prozess 2 beginnt mit P, findet das Semaphor mit dem Wert 1 vor, beendet seine while-Schleife und erniedrigt den Wert des Semaphors.
  - Prozess 1 macht weiter und erniedrigt den Wert des Semaphors auf -1
- P führt ein sogenanntes busy-waiting aus, falls "sema" den Wert Null besitzt (keine intelligente Art, eine CPU zu benutzen).



Semaphore müssen durch den Kern (Prozesssteuerung) unterstützt werden.



Unix und Semaphore

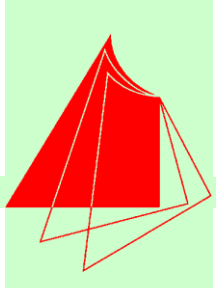
Statt eines Semaphors wird immer ein Satz von (allgemeinen) Semaphoren (Array) angelegt.

Neben dem Namen gibt es mehrere Variablen, die einen solchen Satz und die einzelnen Semaphoren beschreiben.

- der Wert des Semaphors (0, 1 oder größer),
- die Prozessnummer des Prozesses, der das Semaphor zuletzt manipuliert hat,
- die Anzahl der Prozesse, die den Wert des Semaphors verändern wollen oder auf einen bestimmten Wert des Semaphors warten,
- Zugriffsrechte und Besitzverhältnisse.

```
struct sem {
ushort semval;    /* Wert des Semaphors        */
short sempid;     /* PID des letzten Prozesses*/
                  /* Prozesse die warten bis */
ushort semncnt; /* semval > aktuellen Wert    */
ushort semzcnt; /* bis semval = 0             */
};
```

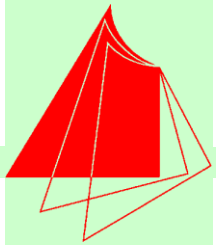
```
struct semid_ds {
struct ipc_perm sem_perm; /* Zugriffsrechte */
struct sem *sem_base;     /* Zeiger auf Sema. */
ushort sem_nsems;         /* Anz Semaphore */
time_t sem_otime;         /* Zugriffszeit      */
time_t sem_ctime;         /* Zugriffszeit      */
};
```



Umgang mit Semaphoren

- **int semget(key_t key, int nsems, int semflg)**
  semget() legt einen Satz von Semaphoren an oder macht eine Satz verfügbar. (IPC_CREATE)
- **int semop(int semid, struct sembuf * operand, unsigned nsops)**
  frägt das Semaphor ab und setzt ihn falls möglich oder gibt ihn frei.
  1. mittels *semop()* feststellen, ob Zugriff möglich ist ggf. Semaphor setzen
  2. die "kritischen Region" betreten
  3. die "kritischen Region" verlassen
  4. mittels *semop()* das Semaphor wieder freigeben
- wurde keine Freigabe erteilt, dann kann der Prozess warten oder den Zugriff abbrechen.

```
struct  sembuf {
short   sem_num; /* Nummer des Semaphors im Satz */
short   sem_op;   /* Operator +x –x                */
short   sem_flg;   /* Flag  IPC_NOWAIT,...         */   }
```



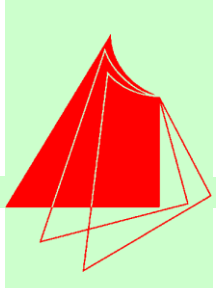
Beispiel

Aktivitäten, die in einem Programm durchgeführt werden müssen, bevor die Funktion semop() genutzt werden kann.

```
struct sembuf  feld[1];          /* Array definiert   */
struct sembuf *pointer_feld; /* Pointer definiert */
pointer_feld    = feld;          /* Pointer setzen    */
```

Im nächsten Schritt werden die Komponenten der Struktur belegt. Danach kann die Funktion semop() ausgeführt werden.

```
pointer_feld->sem_num =  0;
pointer_feld->sem_op  = -1;
pointer_feld->sem_flg =  0;

return_value = semop(semid, pointer_feld, 1);
```



Kontrollieren und Steuern der Semaphore

Mit der Funktion *semctl()* kann der Status eines Semaphors kontrolliert werden.

```
int semctl(int semid, int semnum, int com, arg)

union semun {
int                 val;
struct semid_ds     *buf;       /* Puffer für SETVAL, GETVAL, ... */
unsigned short      *array;     /* Puffer für IPC_STAT und IPC_SET */
struct seminfo      * _ _buf;   /* linux spezifisch, Puffer für IPC_INFO*/
 } arg
```

Die Funktion semctl() wird mit vier Parameter aufgerufen. Der erste Parameter identifiziert das Semaphor. Der zweite Parameter gibt die Nummer des zu bearbeitenden Semaphors an. Der dritte Parameter enthält ein Kontrollkommando, das die Funktion und Wirkungsweise der Funktion semctl() steuert.

- **Status lesen: Werte des Semaphors werden abgefragt und im vierten Parameter gespeichert**
- **Status schreiben: Werte aus dem vierten Parameter werden in die Variable des Semaphors geschrieben.**

Aufgabe: allgemein

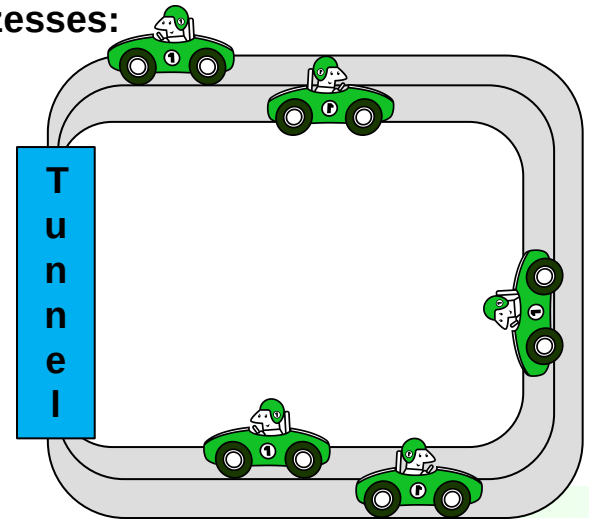
Schreiben Sie eine einfache Simulation in C bzw. C++ unter Linux für folgendes Problem:
(angelehnt an Betriebssysteme WS 0304)

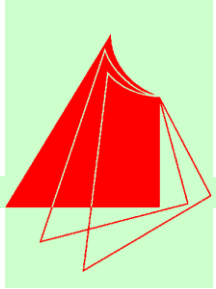
In einer zweispurigen Nord-Süd-Straße befindet sich ein einspuriger Tunnel, in dem sich zeitgleich mehrere Autos aufhalten können. Ein südwärts fahrendes Auto, das an den Tunnel heranfährt darf in den Tunnel einfahren sofern der Tunnel frei ist oder bereits Fahrzeuge in südlicher Richtung im Tunnel unterwegs sind. Solange weitere Autos in südlicher Richtung ankommen, haben diese Vorrang, selbst wenn zwischenzeitlich Autos aus der nördlichen Richtung warten. Bei leerem Tunnel bestimmt das erste Auto die Vorzugsrichtung im Tunnel.

Synchronisieren Sie den simulierten Verkehr durch den einspurigen Tunnel mittels eines Monitors `TUNNEL_MANAGER` mit den Monitorprozeduren `arriveSouth`, `leaveSouth`, `arriveNorth`, `leaveNorth` und modellieren Sie jedes Auto in Form eines eigenen Prozesses:

Ein südwärts fahrendes Auto sieht in Pseudocode folgendermaßen aus:

```
Process Auto {
  while( Simulation aktiv ){
    % an Tunnel heranfahren
    TUNNEL_MANAGER.arriveNorth
    % Fahrzeug im Tunnel
    TUNNEL_MANAGER.leaveSouth
  }
}
```



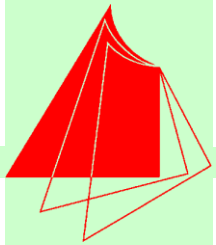


Aufgabe: Details I

- Entwerfen Sie den Monitor und seine Prozeduren in Pseudocode. Welche Conditions und Datenstrukturen sind erforderlich?

- Realisieren Sie den Monitor und die an den Tunnel heranfahrenden Autos als eigenständige Programme in C/C++.
  - implementieren Sie die Aufrufe der Monitorfunktionen mittels Interprozesskommunikation – Shared-Memories, Message-Queues und Semaphoren
  - nutzen Sie Signale, um die Prozesse (Autos) anzuhalten und zu aktivieren

- Überwachen Sie die Simulation durch ein eigenes Programm, das jede „Traffic-Änderung im Tunnel“ protokolliert – 3 Autos NS, 0 Autos SN. Speichern Sie hierzu die entsprechenden Daten des Monitors in einem Shared-Memory und sichern Sie dies geeignet ab.



Aufgabe: Details II

- Sorgen Sie dafür, dass bei der Simulation der Autos sowohl das Heranfahren als auch die Zeit im Tunnel eine zufällige Dauer von etwa 3-5 Sekunden aufweist. Geben Sie jeweils aus, wo sich das Fahrzeug befindet und was es macht. (wartet vor dem Tunnel N, im Tunnel NS, usw.)
- Starten Sie die Simulation mit unterschiedlich vielen Autos (Prozessen) und beschreiben Sie das zu beobachtende Verhalten.
  Idealerweise ordnen Sie jedem Auto ein eigenes Terminal zu:

  \> xterm -e auto NS &

Die Simulation sollte sich nach etwa 100 Sekunden selbständig beenden.