

## 2 UNIX Interprozesskommunikation

- Austausch von Informationen zwischen kooperierenden Prozessen
- Synchronisation kooperierender Prozesse
- Kommunikationsmechanismen
  - gemeinsame Speicherbereiche (*shared memory*)
  - Kommunikationskanäle (*named / unnamed pipes, fifo files*)
  - Nachrichtenübertragung (*message queues*)
  - Nachrichtenwarteschlangen (*mailbox systems*)
  - *Sockets*
  - entfernter Prozeduraufruf (*remote procedure call, RPC*)
  - etc.
- Synchronisationsmechanismen
  - Sperren (*file locks, read-lock, write-lock*)
  - Semaphoren
  - Monitore
  - etc.

## 2.1 Prozesse erzeugen und beenden

- Programm: Sammlung von Anweisungen und Daten
  - befindet sich im Allgemeinen auf der Festplatte
  - ist im *i-node* als *ausführbar* gekennzeichnet
  - der Dateiinhalt entspricht den Regeln des Betriebssystems für ausführbare Programme (*a.out*-Format, *elf*-Format, ...)
  
- Ausführung des Programms
  - 1) Betriebssystemkern muss einen Prozess erzeugen, d. h. eine Umgebung bereitstellen, in der das Programm ausgeführt werden kann
  - 2) der Prozess besteht aus drei Bereichen:
    - *instruction segment*
    - *user data segment*
    - *system data segment*
  - 3) das Programm wird zur Initialisierung des *instruction segments* und *user data segments* benutzt

- 4) nach der Initialisierung besteht zwischen dem Prozess und dem Programm, das er ausführt, keine weitere Verbindung
  - 5) der Prozess kann weitere Betriebsmittel (mehr Speicher, neue Dateien, usw.) anfordern, die im Programm nicht vorhanden sind
- mehrere parallel ablaufende Prozesse können mit demselben Programm initialisiert werden
  - der Betriebssystemkern kann Hauptspeicher sparen, wenn solche Prozesse ein gemeinsames *instruction segment* verwenden  
(die beteiligten Prozesse können die gemeinsame Nutzung nicht feststellen, da die Segmente nur Lesezugriffe erlauben)
  - das *system data segment* enthält u. a. folgende Angaben:
    - aktuelles Dateiverzeichnis
    - Tabelle der offenen Dateien
    - akkumulierte CPU-Zeit
    - Prozessnummer des Vaterprozesses
  - ein Prozess kann sein *system data segment* nicht direkt manipulieren, da es außerhalb seines Adressraums liegt  
⇒ er benötigt Systemfunktionen zur Manipulation

- ein Prozess wird im Auftrag eines Prozesses vom Betriebssystemkern erzeugt
  - auftraggebender Prozess: Vaterprozess (*parent process*)
  - erzeugter Prozess: Sohnprozess (*child process*)
- der Sohnprozess erbt den größten Teil der Systemumgebung des Vaterprozesses (z. B. alle offenen Dateien, Semaphore, ...)
- jeder Prozess wird durch eine eindeutige Prozessnummer identifiziert (process-ID, PID)
- bis auf einen Prozess (den *init*-Prozess) hat jeder Prozess einen Vaterprozess
- der *init*-Prozess ist die Wurzel des Prozesssystems unter UNIX  
(falls ein Prozess endet, bevor seine Sohnprozesse geendet haben, wird der *init*-Prozess automatisch Vaterprozess der *Waisen*, d. h. die Vaterprozessnummer verwaister Prozesse ist 1)
- jeder Prozess gehört einer Prozessgruppe an, die durch eine Prozessgruppennummer identifiziert wird (process-group-ID, PGID)  
(Beispiel: alle Prozesse eines Datenbank-Managementsystems)

- ein Prozess der Gruppe ist der *Gruppenleiter*  
(bei diesem Prozess stimmen Prozessnummer und Gruppennummer überein)
- alle Prozesse einer Gruppe haben als Gruppennummer die Prozessnummer des Gruppenleiters
- es gibt Systemaufrufe, die alle Mitglieder einer Gruppe betreffen  
(auf diese Weise können z. B. alle Prozesse einer Gruppe beendet werden)
- ein Prozess kann seine Gruppennummer ändern und selbst Gruppenleiter einer eigenen Gruppe werden  
(siehe Handbuchseite *setpgid*)
- die Ausgaben aller Prozesse einer Gruppe erfolgen im Allgemeinen auf dem Sichtgerät, an dem der Gruppenleiterprozess gestartet wurde  
(d. h., dass der Bildschirm ein kritischer Bereich ist, auf dem sich die Ausgaben der Prozesse u. U. vermischen)
- der Gerätetreiber des Sichtgerätes sendet alle *interrupt* (<Ctrl-c>, <Strg-c>), *quit* (<Ctrl-|>, <Strg-|>) und *hangup* Signale (<Ctrl-Break>, <Strg-Untbr>) an alle Prozesse, die dieses Sichtgerät benutzen  
(u. U. werden auf diese Weise alle Prozesse beendet, wenn keine Vorkehrungen (z. B. *ignore hangups*) getroffen wurden)

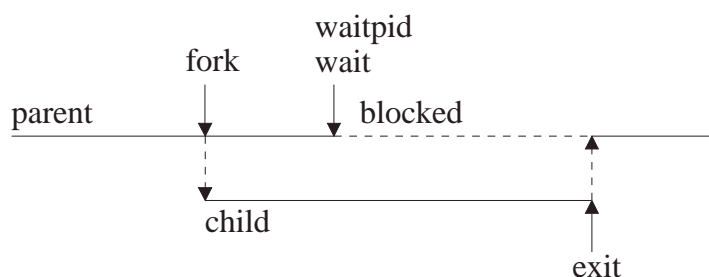
- erzeugen eines neuen Prozesses
  - `pid_t fork (void)`
  - der Sohnprozess ist im Wesentlichen eine Kopie des Vaterprozesses
  - der Vaterprozess erhält die Prozessnummer des Sohnprozesses als Ergebnis zurück
  - der Sohnprozess erhält den Wert 0 als Ergebnis zurück
  - `fork ( )` liefert `-1` zurück und es wurde kein Sohnprozess erzeugt, wenn ein Fehler aufgetreten ist  
(z. B. wenn die Prozesstabelle voll ist oder der Benutzer seine maximal erlaubte Anzahl an Prozessen erzeugt hat)
  - der Sohnprozess überlagert sich im Allgemeinen mit einem anderen Programm (`exec`-Funktion)
  - der Vaterprozess wartet entweder auf das Ende des Sohnprozesses (`wait`- oder `waitpid`-Funktion) oder erledigt irgendwelche sonstigen Arbeiten
  - der `fork`-Aufruf ist sehr aufwendig, da u. U. große Datenssegmente kopiert werden müssen
  - die Unterschiede in den Prozessumgebungen von Vater- und Sohnprozess können der Handbuchseite entnommen werden

- auf das Ende eines Sohnprozesses warten
  - `pid_t wait (int *stat_loc)`  
`pid_t waitpid (pid_t pid, int *stat_loc, int options)`
  - `wait ( )` liefert als Rückgabewert die Prozessnummer des Sohnprozesses oder `-1` im Fehlerfall zurück
  - der Rückgabewert/Statuswert des Sohnprozesses wird in die Variable gespeichert, auf die `stat_loc` zeigt
  - die Rückgabewerte und Parameter von `waitpid ( )` können der Handbuchseite entnommen werden
  - falls ein Sohnprozess endet, bevor sein Vaterprozess auf sein Ende wartet, wird er im Allgemeinen ein *Zombie*-Prozess
    - ◆ das *instruction*, *user data* und *system data segment* werden freigegeben
    - ◆ der Prozess behält seinen Eintrag in der Prozesstabelle
    - ◆ sobald der Vaterprozess `wait ( )` oder `waitpid ( )` aufruft, wird auch der Eintrag in der Prozesstabelle gelöscht
    - ◆ *Zombie*-Prozesse werden z. B. verhindert, wenn der Vaterprozess das Signal `SIGCHLD` ignoriert oder mit Hilfe der Funktion `sigaction ( )` das *Flag* `SA_NOCLDWAIT` setzt
    - ◆ *Zombie*-Prozesse werden spätestens beim Ende des Vaterprozesses oder durch einen Neustart des Rechners entfernt

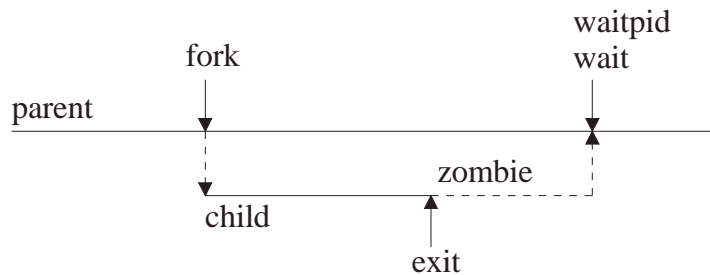
- einen Prozess beenden
  - void **exit** (int status)
  - beendet den Prozess und gibt einen Statuswert an den Vaterprozess
  - falls der Prozess noch Sohnprozesse besitzt, werden sie **nicht** abgebrochen, sondern erhalten den *init*-Prozess als Vaterprozess
  - offene Dateien werden automatisch geschlossen, nicht freigegebene Speicherbereiche werden freigegeben usw.  
(eine Beschreibung aller Aktionen kann der Handbuchseite entnommen werden)

- mögliche parallele Abläufe

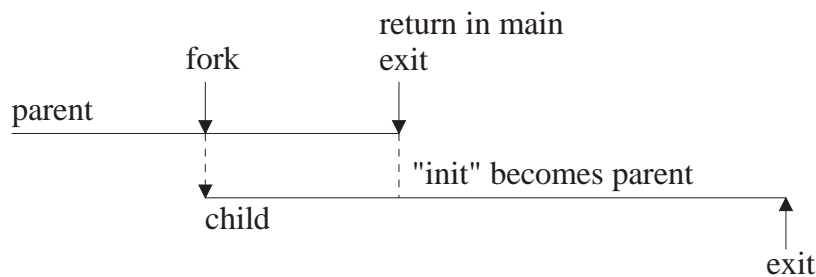
- 1) der Vaterprozess wartet bereits auf das Ende des Sohnprozesses  
(fork1.c, fork2.c)



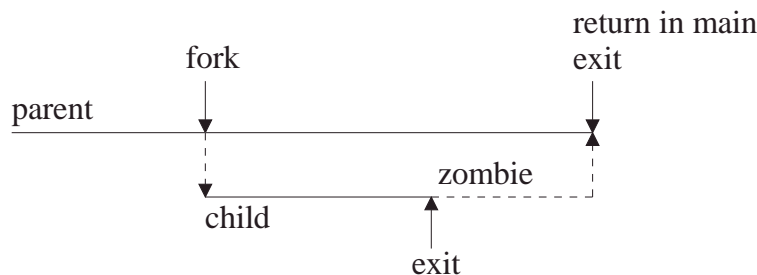
- 2) der Vaterprozess wartet noch nicht auf das Ende des Sohnprozesses (fork3.c)



- 3) der Vaterprozess wartet nicht auf seinen Sohnprozess und endet bevor der Sohnprozess endet (fork4.c)



- 4) der Vaterprozess wartet nicht auf seinen Sohnprozess und endet nachdem der Sohnprozess geendet hat (fork5.c)



der Sohnprozess bleibt **kein** dauerhafter *Zombie*-Prozess

- allgemeine Struktur eines Programms zur Prozesserzeugung (fork1.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

int eval_wait_stat (int stat);

int main (void)
{
    pid_t parent_pid, child_pid,          /* process ID's          */
          fork_pid, wait_pid,
          parent_grp, child_grp;        /* process groups       */
    int   child_stat;                  /* return status of child */

    parent_pid = getpid ();
    parent_grp = getpgrp ();
    printf ("\nParent process: process ID: %ld    group ID: %ld\n",
            (long) parent_pid, (long) parent_grp);
    fork_pid = fork ();
    switch (fork_pid)
    {
        case -1:                        /* error: no process created */
            perror ("fork failed");
            exit (EXIT_FAILURE);
            break;

        case 0:                          /* child process          */
            child_pid = getpid ();
            child_grp = getpgrp ();
            printf ("Child process:  process ID: %ld    group ID: %ld    "
                    "parent process ID: %ld\n", (long) child_pid,
                    (long) child_grp, (long) getpid ());
            printf ("Child process:  terminate with \"exit\"-value: %d\n",
                    EXIT_SUCCESS);
            exit (EXIT_SUCCESS);
            break;

        default:                          /* parent process          */
            printf ("Parent process: child process with ID %ld created.\n",
                    (long) fork_pid);
            wait_pid = wait (&child_stat);
            if (wait_pid == -1)
            {
                perror ("wait");
                exit (EXIT_FAILURE);
            }
            else
            {
                printf ("Parent process: child process %ld has terminated.\n",
                        (long) wait_pid);
                eval_wait_stat (child_stat);
            }
    }
    return 0;
}

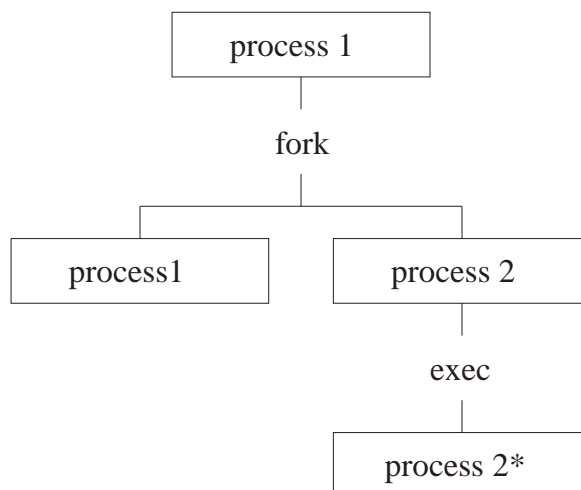
```

```
int eval_wait_stat (int stat)
{
    int return_val;
    if (WIFEXITED (stat) != 0)
    {
        printf ("\t\tChild has terminated with status %d.\n",
                WEXITSTATUS (stat));
        return_val = 0;
    }
    if (WIFSIGNALED (stat) != 0)
    {
        printf ("\t\tChild has been terminated using signal %d.\n",
                WTERMSIG (stat));
        return_val = 0;
    }
    if (WIFSTOPPED (stat) != 0)
    {
        printf ("\t\tChild has been stopped using signal %d.\n",
                WSTOPSIG (stat));
        return_val = 1;
    }
    #if defined(SunOS) && !defined(_POSIX_C_SOURCE)
    if (WIFCONTINUED (stat) != 0)
    {
        printf ("\t\tChild has continued.\n");
        return_val = 1;
    }
    #endif
    return return_val;
}
```

- **Aufgaben:**

- 2-1) Schreiben Sie ein Programm, in dem **ein** Prozess drei parallele Sohnprozesse erzeugt. Die Sohnprozesse sollen ihre PID, GID und PPID ausgeben und sich dann nach einer kurzen Wartezeit selbst beenden. Der Vaterprozess soll auf die Sohnprozesse warten.
- 2-2) Schreiben Sie ein Programm, in dem ein Prozess einen Sohnprozess erzeugt, der wieder einen Sohnprozess erzeugt, usw. Realisieren Sie eine Schachtelungstiefe von drei Prozessen. Jeder Prozess soll sich sonst wie in Aufgabe 2-1) verhalten.
- 2-3) Modifizieren Sie Aufgabe 2-1) in der Weise, dass jeder Sohnprozess eine eigene Prozessgruppe erzeugt. Zeigen Sie, dass sich die Lösungen 2-1) und 2-3) bei der Eingabe von <Ctrl-c> bzw. <Strg-c> unterschiedlich verhalten.

- überlagern eines Prozesses mit einem neuen Programm
  - `int execl (const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/)`
  - `int execv (const char *path, char *const argv[])`
  - `int execle (const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/, char *const envp[])`
  - `int execve (const char *path, char *const argv[], char *const envp[])`
  - `int execlp (const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/)`
  - `int execvp (const char *file, char *const argv[])`
- Vorgehensweise



- 1) zuerst einen neuen Prozess erzeugen (Kopie des Vaterprozesses)
- 2) der Sohnprozess überlagert sich mit einem neuen Programm und hat jetzt sein individuelles Code- und Datensegment

- die verschiedenen *exec*-Funktionen unterscheiden sich in
  - ◆ der Art der Parameterübergabe (explizite Liste, Feld)
  - ◆ der Übergabe der Umgebungsvariablen (automatisch, manuell)
  - ◆ der Suche nach dem Programm (expliziter Pfad, Auswertung der Umgebungsvariablen *PATH*)

Systemaufruf	Format der Argumente	Übergabe der Umgebungsvariablen	Suche über <i>PATH</i> -Variable
<i>execl</i>	Liste	automatisch	nein
<i>execv</i>	Feld	automatisch	nein
<i>execle</i>	Liste	manuell	nein
<i>execve</i>	Feld	manuell	nein
<i>execlp</i>	Liste	automatisch	ja
<i>execvp</i>	Feld	automatisch	ja

- l explizite Parameterliste
  - v variable Anzahl Parameter  
(notwendig, falls die Anzahl der Parameter zur Übersetzungszeit unbekannt ist)
  - e manuelle Übergabe der Umgebungsvariablen
  - p Programm über Pfadvariable suchen
- im Allgemeinen werden die Funktionen *execlp* und *execvp* verwendet

- die *exec*-Funktionen können auch ohne *fork* benutzt werden  
(Wenn ein großes Programm in mehreren Phasen abläuft, kann die nächste Phase durch *exec* geladen werden. Da das Datensegment des Prozesses durch das neue Programm initialisiert wird, müssen die Phasen im Allgemeinen unabhängig voneinander sein oder ihre Daten über temporäre Dateien austauschen.)
- das erste Argument der Parameterliste bzw. des Parameterfeldes (Parameter *arg0*) ist immer ein Programmname
- das letzte Argument der Parameterliste bzw. des Parameterfeldes ist immer *NULL*

– **Beispiel:** Programm mit "*exec ( )*" (forkexec.c)

```

...
int main (void)
{
    pid_t fork_pid, wait_pid;          /* process ID's          */
    int   child_stat,                 /* return status of child */
          ret_val;                    /* return value of a function */

    fork_pid = fork ();
    switch (fork_pid)
    {
        case -1:                       /* error: no process created */
            perror ("fork failed");
            exit (EXIT_FAILURE);
            break;
        case 0:                         /* child process          */
            ret_val = execl ("/bin/ls", "ls", "-al", NULL);
            if (ret_val == -1)
            {
                perror ("execl failed");
                exit (EXIT_FAILURE);
            }
            break;
        default:                        /* parent process        */
            printf ("Parent process: child process with ID %ld created.\n",
                    (long) fork_pid);
            wait_pid = wait (&child_stat);
            if (wait_pid == -1)
            {
                perror ("wait");
                exit (EXIT_FAILURE);
            }
            else
            {
                printf ("Parent process: child process %ld has ...\n",
                        (long) wait_pid);
                eval_wait_stat (child_stat);
            }
    }
    return 0;
}
...

```

- **Aufgabe 2-4:**

Schreiben Sie ein Programm, das vom Benutzer den Namen eines auszuführenden Programms und dessen Parameter erfragt und dieses Programm dann ausführt. Das Programm soll über die Variable *PATH* gesucht werden. Wenn das gewünschte Programm geendet hat, soll Ihr Programm dem Benutzer neue Eingaben erlauben (im Prinzip eine kleine *Shell*).

## 2.2 Signale

- Signale werden im Allgemeinen vom Betriebssystemkern gesendet, um dem Prozess Ausnahmebedingungen mitzuteilen
  - synchrone Signale (*traps*)
    - ◆ werden durch die Ausführung des Programmcodes erzeugt
    - ◆ werden im Allgemeinen direkt an den *Thread* ausgeliefert, der das Signal verursacht hat
    - ◆ Beispiele: SIGSEGV, SIGFPE, SIGILL
  - asynchrone Signale (*interrupts*)
    - ◆ werden durch den Benutzer oder ein Programm erzeugt
    - ◆ werden im Allgemeinen an einen beliebigen *Thread* ausgeliefert
    - ◆ Beispiele: SIGINT, SIGQUIT, SIGTERM, SIGUSR1
- die meisten Signale kann der Prozess ignorieren oder abfangen, d. h. durch eine eigene Routine bearbeiten lassen  
(in der Standardeinstellung beenden die meisten Signale den Prozess, der das Signal empfängt)
- Signale enthalten keine Informationen, d. h. der Empfänger weiß nicht, wer das Signal gesendet hat  
(Signale sollten nur für Ausnahmebedingungen verwendet werden und nicht zur Kommunikation. Nur der *Superuser* darf Signale an fremde Prozesse senden.)

- einige Signalnamen und Nummern (siehe auch Handbuchseite *signal(5)*)  
(im Programm dürfen nur die Namen benutzt werden)

Signalname	Signalnummer	Bedeutung
SIGHUP	1	<i>Hangup</i> . Abbruch der Dialogstationsleitung (analog zum Aufhängen beim Telefon). Kann im Allgemeinen durch <Ctrl-Break> bzw. <Strg-Untbr> erzeugt werden.
SIGINT	2	<i>Interrupt</i> . Wird jedem Prozess gesendet, der mit der Dialogstation verbunden ist, wenn die <i>Interrupt</i> -Taste vom Benutzer gedrückt wird (im Allgemeinen <Ctrl-c> bzw. <Strg-c>).
SIGQUIT	3	<i>Quit</i> . Analog SIGINT, wenn die <i>Quit</i> -Taste gedrückt wird (im Allgemeinen <Ctrl-\> bzw. <Strg-\>).
SIGFPE	8	<i>Floating-point exception</i> . Fehler bei einer Operation mit Gleitkommazahlen.
SIGKILL	9	<i>Kill</i> . Dieses Signal bricht einen Prozess unbedingt ab, da es nicht ignoriert oder abgefangen werden kann. Es sollte nur in Notfällen benutzt werden. Im Normalfall sollte SIGTERM verwendet werden.
SIGPIPE	13	<i>Write on a pipe not opened for reading</i> . Falls ein Prozess in einer <i>Pipeline</i> anormal endet, erhalten alle Prozesse auf der rechten Seite der <i>Pipeline</i> (Leserprozesse) ein <i>end-of-file</i> und alle Prozesse auf der linken Seite (Schreiberprozesse) dieses Signal. Die Schreiberprozesse können nur auf diese Weise über den "Tod" eines Leserprozesses informiert werden.
SIGALRM	14	<i>Alarm clock</i> . Dieses Signal zeigt an, dass sich der "Wecker" des Prozesses gemeldet hat, der mit der Systemfunktion <i>alarm</i> gestellt wurde.
SIGTERM	15	<i>Software termination</i> . Das normale Beendigungs-Signal für einen Prozess ( <i>kill</i> -Kommando, <i>shutdown</i> -Kommando). Dieses Signal sollte im Allgemeinen weder ignoriert noch abgefangen werden!
SIGUSR1 SIGUSR2	16 17	<i>User defined signal</i> . Obwohl diese Signale zur Interprozesskommunikation benutzt werden können, werden sie im Allgemeinen nicht verwendet, da Prozesse nicht über Signale miteinander kommunizieren sollen (Da Signale z. B. ignoriert werden können, sind sie kein sicheres Kommunikationsmittel.)

- falls ein Programm vor einer zwangsweisen Beendigung auf jeden Fall noch bestimmte *Aufräumarbeiten* erledigen muss, sollte es die Signale **SIGHUP**, **SIGINT** und **SIGTERM** abfangen  
(Die Aufräumarbeiten sollten **sehr schnell** erledigt werden. Danach sollte der Prozess sofort *exit* aufrufen! Damit der Benutzer das Programm noch über die Tastatur abbrechen kann, sollte das Signal **SIGQUIT** in diesem Fall nicht abgefangen werden.)
- die folgenden Signale beenden normalerweise (*default*) nicht nur die Prozesse, sondern erzeugen auch einen Speicherabzug (*core dump*):
  - **SIGQUIT** (quit)
  - **SIGILL** (illegal instruction)
  - **SIGTRAP** (trace trap)
  - **SIGABRT** (abort)
  - **SIGEMT** (emulator trap instruction)
  - **SIGFPE** (arithmetic exception)
  - **SIGBUS** (bus error)
  - **SIGSEGV** (segmentation violation)
  - **SIGSYS** (bad argument to system call)
  - **SIGXCPU** (CPU time limit exceeded)
  - **SIGXFSZ** (file size limit exceeded)

- einfaches Verfahren zur Signalbehandlung
  - `void (*signal (int sig_name, void (*sig_handler) (int))) (int)`
    - ◆ diese Funktion ist auch in ANSI C bzw. ISO-C definiert  
(*signal* ist eine Funktion mit zwei Parametern (*sig\_name* und *sig\_handler*), die einen Zeiger auf eine Funktion mit einem Parameter vom Typ *int* ohne Rückgabewert zurückliefert. *sig\_handler* ist ein Zeiger auf eine Funktion mit einem Parameter vom Typ *int* ohne Rückgabewert.)
    - ◆ *signal* ordnet einem Signal mit Namen *sig\_name* eine neue Bearbeitungsfunktion mit Namen *sig\_handler* zu
    - ◆ falls die neue Funktion installiert werden konnte, wird die Adresse der bisherigen Bearbeitungsfunktion als Rückgabewert geliefert und sonst *SIG\_ERR* (in diesem Fall wird *errno* gesetzt)
    - ◆ *sig\_handler* kann folgende Werte haben:
      - 1) **SIG\_DFL** wählt die Standard-Bearbeitungsroutine aus, d. h. der Prozess wird durch die meisten Signale beendet
      - 2) **SIG\_IGN** ignoriert das Signal  
(SIGKILL und SIGSTOP können nicht ignoriert werden)
      - 3) Adresse einer neuen Bearbeitungsfunktion  
(SIGKILL und SIGSTOP können nicht abgefangen werden)

- wenn ein Signal abgefangen werden soll, laufen beim Eintreffen des Signals folgende Aktionen ab:
  - 1) im Allgemeinen wird die Bearbeitungsroutine für das Signal wieder auf SIG\_DFL zurückgesetzt, bevor die eigene Bearbeitungsroutine ausgeführt wird
  - 2) der Bearbeitungsroutine wird die Nummer des abgefangenen Signals als Parameter übergeben
- die Bearbeitungsroutine kann mit *abort* ( ), *exit* ( ), *longjmp* ( ) oder *return* enden
- falls die Routine mit *return* endet, melden einige Systemfunktionen den Fehler *EINTR* (*error interrupted*)
- da die Funktionen der C-Laufzeitbibliothek u. U. nicht invariant (*reentrant*) programmiert sind, können sie nicht zuverlässig in einer Bearbeitungsroutine benutzt werden, die mit *return* endet
- es ist u. U. wichtig, wann die eigene Routine reinstalliert wird
  - ◆ am Anfang der Signalbehandlungsroutine
    - ⇒ kann z. B. bei SIGCHLD zu einem Fehler mit der Meldung *Segmentation Fault (core dumped)* führen  
(Solaris 2.5.1, nicht bei Solaris ab Version 7, Linux)
  - ◆ am Ende der Signalbehandlungsroutine
    - ⇒ der Prozess wird u. U. beendet, weil die eigene Routine nicht schnell genug reinstalliert wurde und das Signal noch einmal aufgetreten ist

- **Beispiel: Signale ignorieren (fork\_sig.c)**

```

void set_sig_action (int sig, void (**old) (int), void (*new) (int),
                    char *err_msg);
static void (*old_sighup) (int); /* address of old signal handler*/
...

int main (void)
{ ...
  fork_pid = fork ();
  switch (fork_pid)
  {
    case -1: /* error: no process created */
      ...
    case 0: /* child process */
      ...
    default: /* parent process */
      set_sig_action (SIGHUP, &old_sighup, SIG_IGN,
                     "Couldn't ignore SIGHUP");
      ...
      printf ("Parent process: ignore SIGHUP, SIGINT, SIGQUIT, and "
              "SIGTERM\n");
      alarm (ALARM_TIME); /* force process termination */
      do
      {
        ...
        wait_pid = waitpid ((pid_t) -1, &child_stat, opt);
        ...
      } while (eval_wait_stat (child_stat) != 0);
      set_sig_action (SIGHUP, &old_sighup, old_sighup,
                     "Couldn't restore SIGHUP");
      ...
      printf ("Parent process: SIGHUP, SIGINT, SIGQUIT, and "
              "SIGTERM restored.\n");
    }
  }
  return 0;
}

void set_sig_action (int sig, void (**old) (int), void (*new) (int),
                    char *err_msg)
{
  register void (*tmp) (int); /* address of old handler */
  if ((tmp = signal (sig, new)) == SIG_ERR)
  {
    perror (err_msg);
    exit (1);
  }
  if ((tmp != SIG_IGN) && (tmp != SIG_DFL) &&
      (new != SIG_IGN) && (new != SIG_DFL))
  {
    if (tmp != new)
    {
      fprintf (stderr, "Error: Another signal handler is already "
               "installed.\n");
      exit (2);
    }
  }
  *old = tmp;
}

```

- **Beispiel: Signalbehandlungsroutine installieren/reinstallieren (sig.c)**

```

...
int main (void)
{
    pid_t fork_pid;                /* Prozessnummer          */
    if (signal (SIGCHLD, sig_handler) == SIG_ERR)
    {
        perror ("signal (SIGCHLD, ...)");
        exit (EXIT_FAILURE);
    }
    fork_pid = fork ();
    switch (fork_pid)
    {
        case -1:                    /* Fehler: kein Prozess erzeugt */
            ...

        case 0:                      /* Sohnprozess              */
            printf ("Sohnprozess: Schlafe %d Sekunden.\n", SLEEP_TIME_CHILD);
            sleep (SLEEP_TIME_CHILD);
            printf ("Sohnprozess: Rufe \"exit (0)\" auf.\n");
            exit (EXIT_SUCCESS);
            break;

        default:                      /* Vaterprozess              */
            printf ("Vaterprozess: Schlafe %d Sekunden.\n",
                    SLEEP_TIME_PARENT);
            sleep (SLEEP_TIME_PARENT);
            printf ("Vaterprozess: Beende mich.\n");
    }
    return 0;
}

void sig_handler (int sig)
{
    #ifdef ANFANG
        if (signal (SIGCHLD, sig_handler) == SIG_ERR)
        {
            perror ("sig_handler");
            exit (EXIT_FAILURE);
        }
    #endif
    switch (sig)
    {
        case SIGCHLD:
            while (waitpid ((pid_t) -1, NULL, WNOHANG) > 0)
            {
                ;
            }
            break;

        default:
            ;
    }
    #ifndef ANFANG
        if (signal (SIGCHLD, sig_handler) == SIG_ERR)
        {
            perror ("sig_handler");
            exit (EXIT_FAILURE);
        }
    #endif
}

```

- detailliertes Verfahren zur Signalbehandlung
  - `int sigaction (int sig_name, const struct sigaction *action, struct sigaction *old_action)`

- ◆ Struktur von *struct sigaction*

```
struct sigaction {
    void      (*sa_handler) (int);
    void      (*sa_action)  (int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int       sa_flags;
};
```

(Solaris packt die beiden Zeiger auf eine Funktion noch in eine Verbundstruktur (*union*))

- ◆ über die Komponenten *sa\_mask* und *sa\_flags* kann detailliert festgelegt werden, wie ein Signal behandelt werden soll
- ◆ das *Flag* `SA_SIGINFO` bestimmt, ob eine Signalbehandlungsroutine mit einem Parameter oder eine mit drei Parametern benutzt werden soll
- ◆ das *Flag* `SA_NOCLDWAIT` verhindert, dass *Zombie*-Prozesse erzeugt werden
- ◆ das *Flag* `SA_RESTART` sorgt dafür, dass einige unterbrochene Systemfunktionen transparent wieder gestartet werden (in diesem Falle würden sie nicht den Fehler *EINTR* melden)
- ◆ weitere Einzelheiten stehen in der Handbuchseite

- ein Signal an den eigenen Prozess senden
  - int **raise** (int sig\_name)
  - diese Funktion ist auch in ANSI C bzw. ISO-C definiert
  - sie führt die Aufgabe mit *kill (getpid ( ), sig\_name)* aus
  
- Signal an einen bestimmten Prozess oder eine Prozessgruppe senden
  - int **kill** (pid\_t pid, int sig\_name)
  - falls *pid* größer als 0 ist, wird das Signal an den Prozess gesendet, dessen ID mit *pid* identisch ist
  - falls *pid* kleiner als -2 ist, wird das Signal an alle Prozesse gesendet, deren PGID mit dem absoluten Wert von *pid* identisch ist
  - *kill ((pid\_t) -1, SIGTERM)* beendet unabhängig von der Prozessgruppe alle Prozesse, die einem Prozess gehören  
(Falls der *Superuser* diesen Funktionsaufruf ausführt, werden alle Prozesse bis auf den *swap*- und *init*-Prozess (PID 0 und 1) beendet. Damit können alle Prozesse auf einfache Weise vor dem Herunterfahren des Systems (*shutdown*) beendet werden.)
  - weitere Einzelheiten stehen in der Handbuchseite

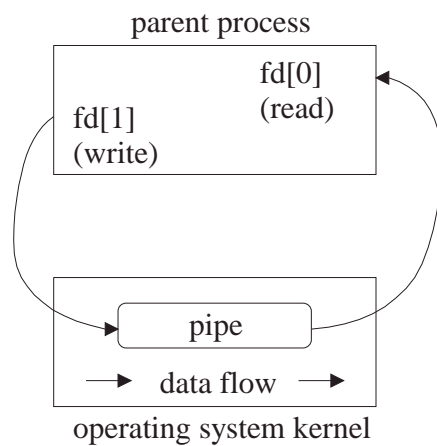
- auf das Eintreffen eines Signals warten
  - int **pause** (void)
  - der Prozess wird bis zum Eintreffen eines Signals blockiert
  
- setzen einer "Weckzeit"
  - unsigned int **alarm** (unsigned int seconds)
  - falls der Prozess nach dem Aufruf von *alarm* auf ein Ereignis wartet, muss die Uhr im Allgemeinen gestoppt werden, wenn der Prozess durch ein anderes Ereignis geweckt wird  
(andernfalls erhält der Prozess das Signal SIGALRM u. U. zu einem Zeitpunkt, zu dem er mit diesem Signal nicht mehr rechnet)
  - jeder neue Aufruf von *alarm* überschreibt den vorhergehenden
  - ein Aufruf mit 0 Sekunden löscht die Weckzeit
  - weitere Einzelheiten stehen in der Handbuchseite
  
- **Aufgabe 2-5:**

Schreiben Sie ein Programm, das  $n$  Sohnprozesse erzeugt. Jeder Sohnprozess soll den vorher erzeugten Sohnprozessen für eine gewisse Zeit wahlweise die Signale SIGUSR? senden. Jedes Signal soll bei jedem Sendevorgang sehr schnell mit einer zufälligen Häufigkeit gesendet werden. Alle Prozesse zählen, wie viele Signale sie gesendet und empfangen haben. Das Ergebnis soll auf geeignete Weise ausgegeben werden.

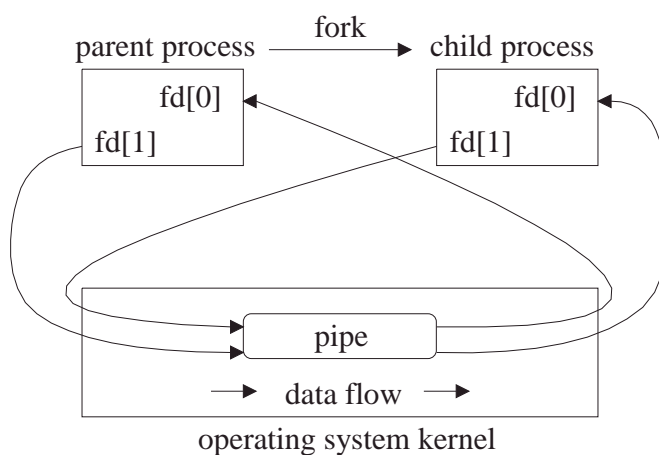
## 2.3 Pipes

- eine *Pipe* erlaubt einen unidirektionalen Datenfluss zwischen zwei Prozessen
- Aufsetzen einer *Pipe*

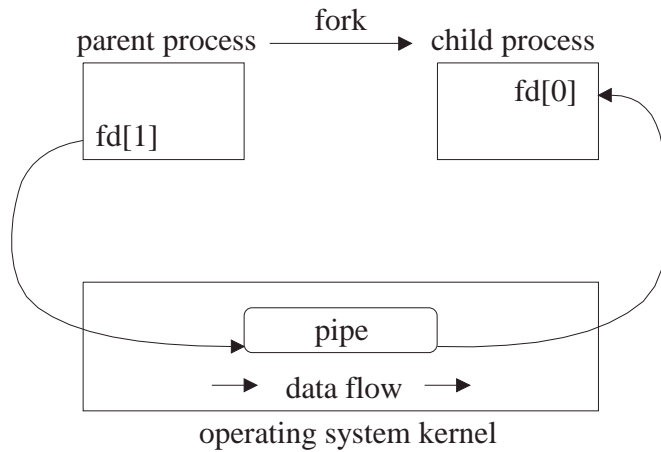
### 1) der Prozess erzeugt eine *Pipe*



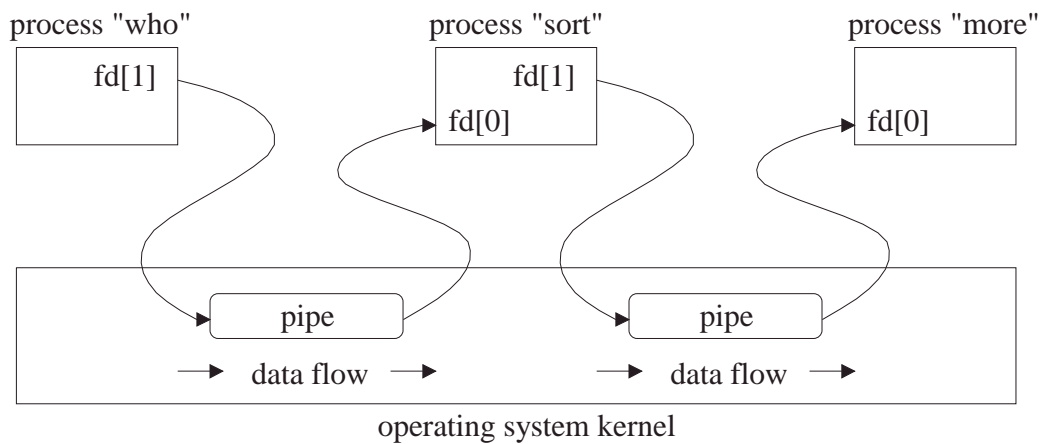
### 2) der Prozess erzeugt einen Sohnprozess



- 3) der Vaterprozess schließt den Leseausgang und der Sohnprozess den Schreibeingang der *Pipe*

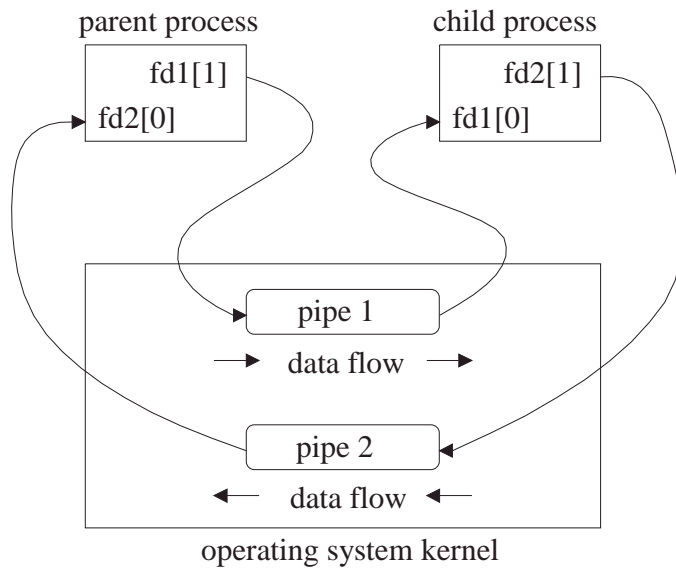


- für das Kommando "who | sort | more" würde auf diese Weise der folgende Ablauf erzeugt werden



- ein bidirektionaler Datenfluss zwischen Vater- und Sohnprozess erfordert zwei unidirektionale *Pipes*, die folgendermaßen aufgesetzt werden

(obwohl einige Betriebssysteme auch bidirektionale *Pipes* unterstützen, sollen hier nur unidirektionale *Pipes* benutzt werden)



- eine *unnamed Pipe* zur Interprozesskommunikation zwischen verwandten Prozessen erzeugen (Vater - Sohn, Sohn - Sohn, ...)
  - `int pipe (int file_descriptor[2])`
  - Datenübermittlung: *first-in-first-out*
  - der Zugriff auf die *Pipe* erfolgt nur über die Datei-Deskriptoren

- einige Systemaufrufe zur Handhabung von *Pipes*:
  - ◆ **write**

Mit Hilfe dieser Funktion können Daten in die *Pipe* geschrieben werden. Falls die *Pipe* voll ist, wird der schreibende Prozess im Allgemeinen solange blockiert bis der lesende Prozess so viele Daten gelesen hat, dass der Schreibauftrag **vollständig** ausgeführt werden kann (atomare Operation). Die Größe der *Pipe* hängt von der Implementierung ab (mindestens 4096 Bytes). Der schreibende Prozess kann ein *end-of-file* erzeugen, indem er den Schreibeingang schließt.
  - ◆ **read**

Mit Hilfe dieser Funktion können Daten aus der *Pipe* gelesen werden. Falls die *Pipe* leer ist, wird der lesende Prozess im Allgemeinen solange blockiert bis mindestens ein Byte in der *Pipe* zur Verfügung steht oder der Schreibeingang geschlossen wurde.
  - ◆ **close**

Mit Hilfe dieser Funktion können die Enden der *Pipe* geschlossen werden. Falls der Schreibeingang geschlossen wird, wird für den Leser ein *end-of-file* erzeugt. Falls der Leseausgang geschlossen wird, führt ein Schreibauftrag zu einem Fehler. In diesem Fall wird das Signal SIGPIPE erzeugt.
  - ◆ **fcntl**

Mit Hilfe dieser Funktion kann u. a. eingestellt werden, ob ein Prozess bei einem erfolglosen Schreib-/Leseauftrag blockiert werden soll oder nicht.
  - ◆ **dup**

Dupliziert einen Datei-Deskriptor.

(Vorgehensweise damit ein Prozess immer vom/auf Datei-Deskriptor 0/1 lesen/schreiben kann: Schließe Deskriptor 0/1. Rufe *dup* mit dem Lese-/Schreib-Deskriptor der *Pipe* auf.)

- eine *named Pipe* (spezielle FIFO-Datei) zur Interprozesskommunikation zwischen Prozessen erzeugen
  - int **mknod** (const char \*path, mode\_t mode, dev\_t dev)
  - erlaubt den Datenaustausch zwischen **beliebigen** Prozessen, die den Namen der FIFO-Datei kennen und die erforderlichen Zugriffsrechte besitzen
  - die FIFO-Datei muss zuerst mit der *open*-Funktion geöffnet werden
  - für *read*, *write*, ... gilt dasselbe wie bei *unnamed Pipes*
  - Schreib- und Lese-Operationen sind auf jeden Fall atomar, wenn die Blockgröße 4096 Bytes nicht überschreitet
  - *unnamed* und *named Pipes* für einige Anwendungen zu langsam
  - das Ergebnis des Shell-Kommandos "ls | wc" kann auch durch die folgenden Kommandos erreicht werden:

```
/etc/mknod mypipe p
ls > mypipe & wc < mypipe
rm mypipe
```

- **Aufgabe 2-6:**

Implementieren Sie einen Erzeuger- und einen Verbraucherprozess, die über eine FIFO-Datei kommunizieren können. Der Erzeugerprozess generiert Datensätze, die eine Prozessnummer (z. B. über einen Parameter auf der Kommandozeile oder seine PID) und eine fortlaufende Nummer enthalten. Die Anzahl der zu erzeugenden Datensätze wird ihm auf jeden Fall als Parameter über die Kommandozeile übergeben. Der Verbraucherprozess gibt die Datensätze aus, nachdem er seine eigene Prozessnummer ausgegeben hat.

Nachdem Sie z. B. mit `"/etc/mknod erz_ver p"` eine FIFO-Datei erzeugt haben, können Sie z. B. folgendes Kommando eingeben, um drei Erzeuger- (Kommando: `erz`) und zwei Verbraucherprozesse (Kommando: `ver`) zu starten:

```
erz 20 & erz 30 & erz 25 & ver & ver
```

Erhalten Sie immer dieselbe Ausgabenreihenfolge ihrer Verbraucherprozesse?

## 2.4 Shared Memory

- es gibt verschiedene Programmierschnittstellen
  - System V IPC (wird in diesem Kapitel vorgestellt)
    - ◆ benutzt nur den Hauptspeicher
    - ◆ ermöglicht effizienten Datenaustausch zwischen parallelen Prozessen  
(schnellste Möglichkeit der Datenübertragung zwischen zwei Prozessen)
    - ◆ benutzt einen *Schlüsselwert* zur Bestimmung der Identifikationsnummer des gewünschten Speicherbereichs
  - mmap ( )
    - ◆ erlaubt einen impliziten Zugriff auf eine Datei über den virtuellen Speicher (*memory mapped object*)
    - ◆ Vorgehensweise
      - eine normale Datei wird geöffnet, um einen Dateideskriptor zu erhalten
      - *mmap ( )* wird mit dem Dateideskriptor und weiteren Parametern aufgerufen, um den Dateiinhalte in den virtuellen Speicher einzubinden
      - die Datei kann über normale Speicherzugriffe gelesen bzw. geändert werden

- ◆ *mmap* ist besser in UNIX integriert als *System V IPC*, da Dateien benutzt werden
  - ◆ *mmap* verwendet Dateinamen und Pfade als Namen, da die gemeinsamen Objekte Dateien sind
  - ◆ gemeinsame Objekte werden mit normalen Systemfunktionen erzeugt bzw. entfernt (*open ( )*, *unlink ( )*)
  - ◆ der Eigentümer und die Zugriffsrechte können mit normalen Systemfunktionen geändert werden (*chown ( )*, *chmod ( )*)
  - ◆ die gemeinsamen Objekte *überleben* einen Stromausfall oder Neustart der Maschine, falls der Speicherbereich vorher in die Datei zurückgeschrieben wurde
  - ◆ *mmap* ist wegen der Festplattenzugriffe sehr langsam
- POSIX IPC
- ◆ ähnelt *mmap ( )*, da es ebenfalls auf dem Dateisystem basiert
  - ◆ es gibt zwei Funktionen: *shm\_open ( )* und *shm\_unlink ( )*  
(sie stellen nur eine Schnittstelle für die POSIX IPC Namensgebung zur Verfügung, d. h. sie ersetzen im Wesentlichen *open ( )* und *unlink ( )* ⇒ die eigentliche Einbindung des gemeinsamen Objekts in den virtuellen Speicher erfolgt dann über *mmap ( )*)

– Überblick über die Programmierschnittstellen

	System V IPC	UNIX mmap	POSIX IPC
Speicherbereich in Betriebssystemtabelle anlegen	shmget ()	open () lseek () write ()	shm_open () lseek () write ()
Speicherbereich an virtuellen Adressraum anbinden	shmat ()	mmap ()	mmap ()
Speicherbereich aus virtuellem Adressraum entfernen	shmdt ()	munmap ()	munmap ()
Speicherbereich aus Betriebssystemtabelle entfernen	shmctl ()	close () unlink ()	close () shm_unlink ()

(Bei *mmap* kann die Größe einer neuen Datei dadurch festgelegt werden, dass mit *lseek* eine Position in der Datei festgelegt wird, an der dann mit *write* ein Zeichen geschrieben wird.)

- mehrere Prozesse können einen gemeinsamen Speicherbereich gleichzeitig in ihren virtuellen Speicher einbinden

- Vorgehensweise bei *System V IPC*
  - 1) Speicherbereich anlegen/öffnen (*shmget*)
  - 2) Speicherbereich an eigenen Datenbereich anbinden (*shmat*)
  - 3) Datenübertragung
  - 4) Speicherbereich freigeben (*shmdt*)
  - 5) Speicherbereich löschen/schließen (*shmctl*)
  
- der Zugriff auf den Speicherbereich kann gesteuert werden (*shmctl*)  
(Zugriffsrechte, Sperren, ...)
  
- neue Segmente belegen noch keinen Speicherplatz  
(es erfolgt nur ein Eintrag in der *Shared Memory* Tabelle des Betriebssystems)
  
- der Speicherplatz wird erst dann belegt, wenn ein Prozess den Bereich in seinen virtuellen Speicher einbindet
  
- der Zugriff auf den gemeinsamen Speicherbereich muss synchronisiert werden  
(im Allgemeinen über *Semaphore*)

- allgemeine Kommandos für einen gemeinsamen Speicherbereich

- int **shmctl** (int shmid, int cmd, struct shmids \*buf)

- ◆ shmid Identifikationsnummer des Speicherbereichs  
(wird von *shmget* geliefert)

- ◆ cmd auszuführendes Kommando
    - IPC\_RMID ID in *Shared Memory* Tabelle löschen
    - IPC\_SET Werte ändern
    - ...

- ◆ buf Zeiger auf Datenpuffer, z. B. bei IPC\_SET

- Struktur des Datensatzes hängt vom Betriebssystem ab

(siehe *shm.h*)

```

struct shmids {
    struct ipc_perm shm_perm; /* operation permission struct */
    int shm_segsz; /* size of segment in bytes */
    struct anon_map *shm_amp; /* segment anon_map pointer */
    ushort shm_lkcnt; /* number of times it is being locked */
    pid_t shm_lpid; /* pid of last shmop */
    pid_t shm_cpid; /* pid of creator */
    ulong shm_nattch; /* used only for shminfo */
    ulong shm_cnattch; /* used only for shminfo */
    time_t shm_atime; /* last shmat time */
    long shm_pad1; /* reserved for time_t expansion */
    time_t shm_dtime; /* last shmdt time */
    long shm_pad2; /* reserved for time_t expansion */
    time_t shm_ctime; /* last change time */
    long shm_pad3; /* reserved for time_t expansion */
    long shm_pad4[4]; /* reserve area */
}

```

- jedes Speicher-Segment muss explizit in der *Shared Memory* Tabelle des Betriebssystems gelöscht werden (IPC\_RMID)

- gemeinsamen Speicherbereich anlegen/öffnen und die Identifikationsnummer des Bereichs zurückliefern
  - `int shmget (key_t key, int size, int shmflg)`
    - ◆ `key` vom Benutzer wählbarer eindeutiger numerischer Schlüssel oder `IPC_PRIVATE`  
(bei `IPC_PRIVATE` kennen fremde Programme den Schlüssel nicht und können das Segment daher nicht benutzen; ein numerischer Schlüssel kann als symbolische Konstante oder Parameter auf der Kommandozeile definiert werden)
    - ◆ `size` Größe des Segments
    - ◆ `shmflg` Zugriffsrechte und Funktionssteuerung  
(falls beim Eröffnen eines neuen Segments keine Zugriffsrechte gesetzt werden, ist das Segment wertlos, da in der Standard-Einstellung nicht einmal der Erzeuger selbst irgendwelche Zugriffsrechte besitzt)
  - Funktionssteuerung
    - `IPC_CREAT` neues Segment erzeugen, falls das Segment mit dem angegebenen Schlüssel noch nicht existiert
    - `IPC_EXCL` liefert einen Fehler, falls das Segment neu angelegt werden soll und der angegebene Schlüssel schon existiert

- *Shared Memory* Operationen

- Speicherbereich an eigenen Datenbereich anbinden und Adresse des Segments zurückliefern

void \***shmat** (int shmid, void \*shmaddr, int shmflg)

- ◆ shmid            Identifikationsnummer des Speicherbereichs
  - ◆ shmaddr        (*void \**) 0, falls das System die Adresse wählen darf (Einzelheiten stehen in der Handbuchseite)
  - ◆ shmflg         Zugriffsrechte (siehe Handbuchseite)
- Speicherbereich freigeben

int **shmdt** (void \*shmaddr)

- ◆ shmaddr        Adresse des freizugebenden Speicherbereichs
- **vor jedem Logout muss überprüft werden, ob Shared Memory Segmente nicht gelöscht worden sind**

- Kommando *ipcs -m* gibt aus, welche Segmente vorhanden sind
- ggf. müssen diese Segmente manuell gelöscht werden
  - ◆ *ipcrm -m <shmid>*            (IRIX, SunOS)
  - ◆ *ipcrm -M <shmkey>*            (IRIX, SunOS)
  - ◆ *ipcrm shm <shmid>*            (Linux)

- **Beispiel:** Nutzung eines gemeinsamen Speicherbereichs ohne Synchronisation

Ein Prozess gibt ein Zeichen n-mal auf dem Bildschirm aus. Ein anderer Prozess kann sowohl das Zeichen ändern als auch die Zahl, die angibt, wie oft das Zeichen ausgegeben werden soll. Wenn zwischen dem Lesen dieser beiden Werte ein Prozesswechsel stattfindet, wird das alte Zeichen mit der neuen Häufigkeit ausgegeben.

– gemeinsame *Header*-Datei für beide Programme (shm\_definitions.h)

```
struct shared_seg { char c;           /* character to print      */
                   int  frq;        /* frequency of character */
                   };

#define SEGSIZE (sizeof (struct shared_seg))
#define SHM_KEY ((key_t) 0x2206)      /* arbitrary value       */
#define SHM_PERM (0600)              /* access permissions    */
```

## – Programm zur Ausgabe der Werte (shm\_output\_2.c)

```

...
#define SLEEP_TIME      2          /* 2 seconds          */
#define ESHMGET         1          /* error calling "shmget()" */
...

int main (void)
{
    struct shared_seg    *shm_adr; /* address of shm segment */
    struct shmctl        shmctl;   /* for shmctl            */
    int                  shm_id,   /* ID of shm segment     */
                    shm_frq,     /* temporary value       */
                    i;           /* loop variable         */
    char                 shm_c;    /* temporary value       */

    shm_id = shmget (SHM_KEY, SEGSIZE, IPC_CREAT | SHM_PERM);
    if (shm_id < 0)
    {
        perror ("shm_output_2: shmget failed");
        exit (ESHMGET);
    }
    /* let the system choose an appropriate address */
    shm_adr = (struct shared_seg *) shmat (shm_id, NULL, SHM_PERM);
    if (shm_adr < (struct shared_seg *) 0)
    {
        perror ("shm_output_2: shmat failed");
        exit (ESHMAT);
    }
    /* initialize the shared memory segment */
    shm_adr->c = 'x';
    shm_adr->frq = 70;
    /* display a new line every "2 * SLEEP_TIME" seconds */
    while ((shm_adr->c != 'q') && (shm_adr->frq > 0))
    {
        shm_c = shm_adr->c;
        printf ("... I'll read the frequency in %d s\n", SLEEP_TIME);
        sleep (SLEEP_TIME); /* allow shm inconsistency */
        shm_frq = shm_adr->frq;
        for (i = 0; i < shm_frq; ++i)
        {
            putchar (shm_c);
        }
        putchar ('\n');
        sleep (SLEEP_TIME);
    }
    /* release the shared memory segment */
    #ifndef XPG4
    ...
    #else
        if (shmdt (shm_adr) < 0)
        {
            perror ("shm_output_2: shmdt failed");
            exit (ESHMDT);
        }
    #endif
    if (shmctl (shm_id, IPC_RMID, &shmctl) < 0)
    {
        perror ("shm_output_2: shmctl failed");
        exit (ESHMCTL);
    }
    return 0;
}

```

– Programm zum Ändern der Werte (shm\_change\_contents.c)

```

...
int main (int argc, char *argv[])
{
    struct shared_seg    *shm_adr; /* address of shm segment    */
    int                  shm_id;   /* ID of shm segment    */

    if (argc != 3)
    {
        fprintf (stderr, "\nThis program changes the \"char\" and \"
                    \"frequency\" in a shared memory segment.\n"
                    "The \"char\" will be displayed \"frequency\" times \"
                    \"on a line.\n\n");
        fprintf (stderr, "Usage: %s <char> <frequency>\n\n", argv[0]);
        exit (EXIT_FAILURE);
    }
    shm_id = shmget (SHM_KEY, SEGSIZE, IPC_CREAT | SHM_PERM);
    if (shm_id < 0)
    {
        perror ("shm_change_contents: shmget failed");
        exit (ESHMGET);
    }
    /* let the system choose an appropriate address */
    shm_adr = (struct shared_seg *) shmat (shm_id, NULL, SHM_PERM);
    if (shm_adr < (struct shared_seg *) 0)
    {
        perror ("shm_change_contents: shmat failed");
        exit (ESHMAT);
    }
    /* change the contents of the shared memory segment */
    shm_adr->c = argv[1][0];
    shm_adr->frq = atoi (argv[2]);
    /* detach the shared memory segment */
    #ifndef XPG4
        if (shmdt ((char *) shm_adr) < 0)
        {
            perror ("shm_change_contents: shmdt failed");
            exit (ESHMDT);
        }
    #else
        ...
    #endif
    return 0;
}

```

- jedes Zeichen wird immer mit der richtigen Häufigkeit ausgegeben, wenn der Zugriff auf den gemeinsamen Speicherbereich durch Semaphore geschützt wird

- *System V IPC* Mechanismen können über Systemparameter konfiguriert werden

SHMMNI maximale Anzahl Identifikationsnummern für *Shared Memory* Segmente

SHMMAX maximale Größe eines *Shared Memory* Segments in Bytes

SEMMNI maximale Anzahl Identifikationsnummern für Semaphore-Kollektionen

SEMMSL maximale Anzahl Semaphore pro Kollektion

SEMOPM maximale Anzahl Operationen pro *semop*-Aufruf

MSGMNI maximale Anzahl Identifikationsnummern für Nachrichtenwarteschlangen

MSGMNB maximale Anzahl Bytes aller Nachrichten der Nachrichtenwarteschlange

MSGTQL maximale Anzahl Nachrichten in der Nachrichtenwarteschlange

- unter *Linux 2.6.x* werden die Werte in den folgenden Dateien festgelegt:
  - `/usr/include/linux/shm.h`
  - `/usr/include/linux/sem.h`
  - `/usr/include/linux/msg.h`

falls Werte geändert werden, muss ein neuer Betriebssystemkern erzeugt werden

die aktuellen Werte können mit folgendem Kommando ausgegeben werden

```
/sbin/sysctl -a | grep -e shm -e sem -e msg
```

- unter Solaris werden die Standardwerte vermutlich direkt in den folgenden Moduldateien festgelegt:
  - shmsys
  - semsys
  - msgsys

(Die Dateien befinden sich im Verzeichnis "/kernel/sys" (32-Bit Betriebssystem), "/kernel/sys/sparcv9" oder "/kernel/sys/amd64" (64-Bit Betriebssysteme).)

bis Solaris 9 werden Änderungen über die Datei */etc/system* vorgenommen (siehe Handbuchseite "*man -s 4 system*")

**z. B.:** set shmsys:shminfo\_shmmni = <value>  
set semsys:seminfo\_semmni = <value>  
set msgsys:msginfo\_msgmni = <value>

falls Werte geändert werden, muss das System neu gestartet werden

die aktuellen Werte können mit folgendem Kommando ausgegeben werden (siehe Handbuchseite *sysdef*)

```
/usr/sbin/sysdef | /usr/xpg4/bin/grep -e SHM -e SEM -e MSG
```

(Solaris 10 führte "Container" und prozessspezifische Ressourcen ein, so dass jetzt alles ein wenig anders gemacht werden muss. Benutzen Sie "man prctl", "man getrctl" und "man rctladm" als erste Informationsquelle, wenn Sie am neuen Konzept interessiert sind. Die aktuellen Werte können ab Solaris 10 mit dem Kommando "rctladm -l" angezeigt werden. Der Vorteil des neuen Konzepts ist, dass die Werte jetzt für jeden Prozess individuell gesetzt werden können und dass das System nicht mehr neu gestartet werden muss, wenn Werte geändert werden.)

## 2.5 Semaphore

- es gibt verschiedene Programmierschnittstellen
    - System V IPC (wird in diesem Kapitel vorgestellt)
      - ◆ benutzt nur den Hauptspeicher
      - ◆ benutzt einen *Schlüsselwert* zur Bestimmung der Identifikationsnummer der gewünschten Semaphore-Kollektion
    - POSIX IPC
      - ◆ namenlose Semaphore
        - benutzen nur den Hauptspeicher
        - entsprechen den Semaphoren von Dijkstra
- ```
#define SEM_INITVAL 1
...
sem_t  s                /* semaphore                               */
...
sem_init (&s, TRUE, SEM_INITVAL);
...
sem_wait (&s);         /* one-to-one correspondence to P(s) */
...
sem_post (&s);         /* one-to-one correspondence to V(s) */
...
```
- ◆ Semaphore mit Namen: basieren auf Dateien im Dateisystem

## – Überblick über die Programmierschnittstellen

|                     | POSIX named<br>semaphores                         | POSIX unnamed<br>semaphores                       | System V<br>semaphores |
|---------------------|---------------------------------------------------|---------------------------------------------------|------------------------|
| Semaphore erzeugen  | sem_open ()                                       | sem_init ()                                       | semget ()              |
| Initialisieren      | beim Erzeugen                                     | beim Erzeugen                                     | semctl ()              |
| P (...)             | sem_wait ()<br>sem_timedwait ()<br>sem_trywait () | sem_wait ()<br>sem_timedwait ()<br>sem_trywait () | semop ()               |
| V (...)             | sem_post ()                                       | sem_post ()                                       | semop ()               |
| Wert lesen          | sem_getvalue ()                                   | sem_getvalue ()                                   | semctl ()              |
| Semaphore entfernen | sem_close ()<br>sem_unlink ()                     | sem_destroy ()                                    | semctl ()              |

- Semaphore aus *System V IPC* sind sehr mächtig
  - sie können um beliebige Werte erhöht/erniedrigt werden
    - ◆ eine Subtraktion wird nur dann durchgeführt, wenn das Ergebnis größer oder gleich Null ist
    - ◆ der Prozess wird ggf. solange blockiert, bis die Subtraktion ausgeführt werden kann
  - mehrere Semaphore können mit **einem** Funktionsaufruf manipuliert werden
    - ◆ die Operation ist atomar, d. h., für Subtraktionen müssen **alle** Ergebnisse größer oder gleich Null sein
    - ◆ der Prozess wird ggf. solange blockiert, bis alle Operationen gleichzeitig ausgeführt werden können
  - in **einem** Funktionsaufruf kann zusätzlich zur Manipulation mehrerer Semaphore darauf gewartet werden, dass einige Semaphore den Wert Null annehmen
    - ◆ die Operation ist atomar, d. h., für Subtraktionen müssen **alle** Ergebnisse größer oder gleich Null sein
    - ◆ der Prozess wird ggf. solange blockiert, bis alle Operationen gleichzeitig ausgeführt werden können und die spezifizierten Semaphore den Wert Null angenommen haben

- der Prozess kann festlegen, dass er nicht blockiert wird, wenn eine Semaphore-Operation nicht sofort ausgeführt werden kann  
(in diesem Fall liefert der Funktionsaufruf einen Fehler zurück und *errno* wird entsprechend gesetzt)
- der Prozess kann festlegen, dass das System alle Operationen "protokolliert" und ggf. automatisch rückgängig macht, wenn der Prozess abbricht

- Darstellung von Semaphoren

- jedem Semaphor ist folgende Datenstruktur zugeordnet

```
struct sem {
    ushort semval;           /* current value, nonnegative */
    pid_t  sempid;          /* PID of last operation */
    /* perhaps more operating system dependent variables */
};
```

- Semaphore werden in Kollektionen (*semaphore set*) zusammengefasst

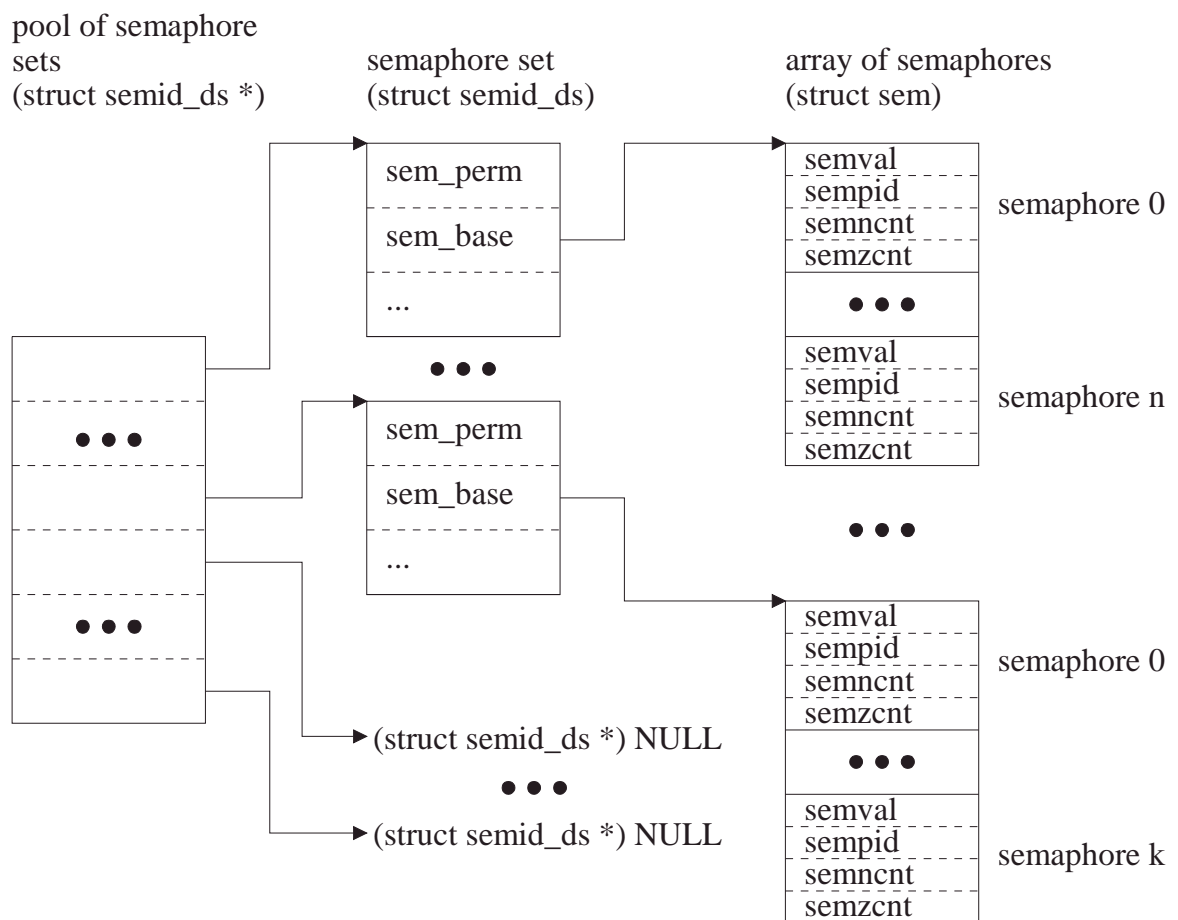
(auch für ein einziges Semaphor muss eine Kollektion erzeugt/initialisiert werden)

- eine Semaphore-Kollektion wird durch folgende Datenstruktur repräsentiert (abhängig vom Betriebssystem; siehe *sem.h*)

```
struct semid_ds {
    struct ipc_perm sem_perm;           /* operation permission struct */
    struct sem      *sem_base;          /* ptr to first semaphore in set */
    ushort          sem_nsems;          /* # of semaphores in set */
    time_t          sem_otime;          /* last semop time */
    long            sem_pad1;           /* reserved for time_t expansion */
    time_t          sem_ctime;          /* last change time */
    long            sem_pad2;           /* time_t expansion */
    long            sem_pad3[4];        /* reserve area */
};
```

diese Datenstruktur gibt es für jede *semid*

- Zusammenfassung der Datenstrukturen für Semaphore



- *semid* wird als Index in das Feld *Semaphor ID Pool* benutzt
- die Datenstruktur spart sehr viel Speicher im Betriebssystemkern, solange keine oder wenige Semaphore benutzt werden

- Vorgehensweise:

- 1) Semaphore anlegen und initialisieren (*semget*)
- 2) Semaphore benutzen (*semop*, *semctl*)
- 3) Semaphore freigeben (*semctl*)

- allgemeine Kommandos für Semaphore

- `int semctl (int semid, int semnum, int cmd, union semun arg)`

- ◆ `semid` Identifikationsnummer der Semaphore-Kollektion

(wird von `semget` geliefert)

- ◆ `semnum` Nummer des Semaphors in der Kollektion

- ◆ `cmd` auszuführendes Kommando

- `GETVAL` Wert eines Semaphors liefern

- `SETVAL` Wert eines Semaphors setzen

- `GETALL` Werte aller Semaphore liefern

- `SETALL` Werte aller Semaphore setzen

- `IPC_SET` u. a. Zugriffsrechte setzen

- `IPC_RMID` Semaphore-Kollektion löschen

- usw. siehe Handbuchseite

- ◆ `arg` abhängig von `cmd`

```
union semun
{
    int          val;          /* z. B. bei SETVAL */
    struct semid_ds *buf;     /* z. B. bei IPC_SET */
    ushort      *array;      /* z. B. bei SETALL */
};
```

- ◆ Rückgabewert ist abhängig von `cmd` (siehe Handbuchseite)

- die Datenstrukturen für Semaphore müssen explizit gelöscht werden (`IPC_RMID`)

- **vor jedem *Logout* muss überprüft werden**, ob Semaphore-Kollektionen nicht gelöscht worden sind
  - Kommando *ipcs -s* gibt aus, welche Kollektionen vorhanden sind
  - ggf. müssen diese Kollektionen manuell gelöscht werden
    - ◆ *ipcrm -s <semid>* (IRIX, SunOS)
    - ◆ *ipcrm -S <semkey>* (IRIX, SunOS)
    - ◆ *ipcrm sem <semid>* (Linux)
  
- Semaphore-Kollektion anlegen und die Identifikationsnummer zurückliefern
  - `int semget (key_t key, int nsems, int semflg)`
    - ◆ `key` vom Benutzer wählbarer eindeutiger numerischer Schlüssel oder `IPC_PRIVATE`  
(bei `IPC_PRIVATE` kennen fremde Programme den Schlüssel nicht und können die Semaphore-Kollektion daher nicht benutzen; ein numerischer Schlüssel kann als symbolische Konstante oder Parameter auf der Kommandozeile definiert werden)
    - ◆ `nsems` gewünschte Anzahl Semaphore in der Kollektion
    - ◆ `semflg` Zugriffsrechte und Funktionssteuerung
  - Funktionssteuerung:
    - `IPC_CREAT` neue Kollektion erzeugen
    - `IPC_EXCL` liefert einen Fehler, falls die Semaphore-Kollektion neu angelegt werden soll und der angegebene Schlüssel schon existiert

- atomare Operationen auf Semaphore einer Kollektion durchführen

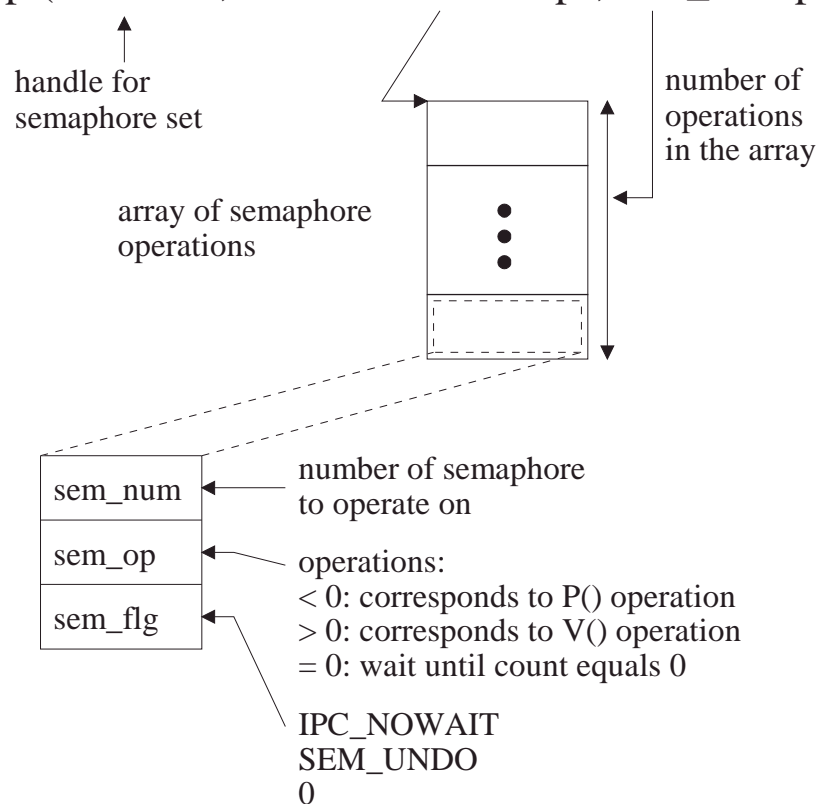
- `int semop (int semid, struct sembuf *sops, size_t nsops)`

```

struct sembuf
{
    ushort  sem_num;          /* Nummer des Semaphors    */
    short   sem_op;          /* Operation                */
    short   sem_flg;          /* Flags                    */
};

```

`semop (int semid, struct sembuf *sops, size_t nsops)`



- die maximale Anzahl Operationen pro `semop`-Aufruf ist abhängig vom Betriebssystem

- $sem\_op > 0$       Semaphorwert um diesen Wert erhöhen
- $sem\_op < 0$       Semaphorwert um den Wert dekrementieren,  
falls möglich  
(ein negatives Ergebnis würde den Prozess ggf. blockieren,  
ohne die Operation auszuführen)
- $sem\_op = 0$       warten bis der Zähler des Semaphors Null  
wird
  
- $sem\_flg$  kann folgende Werte annehmen:
  - **IPC\_NOWAIT** kann für jede einzelne Operation gesetzt werden  
Falls **eine** Operation, für die IPC\_NOWAIT gesetzt ist, nicht erfolgreich durchgeführt werden kann, wird überhaupt keine Operation für die Kollektion durchgeführt,  $semop$  liefert den Wert  $-1$  zurück und setzt  $errno$  entsprechend.
  - **SEM\_UNDO** alle Semaphor-Operationen des Prozesses können vom Betriebssystem rückgängig gemacht werden, wenn der Prozess endet und nicht selbst für einen korrekten Zustand sorgt  
**Achtung:** Darf nicht benutzt werden, wenn Prozesse endlich lange laufen und die  $P$ -Operation in einem Prozess und die zugehörige  $V$ -Operation in einem anderen Prozess ausgeführt wird. In der Lehrveranstaltung Betriebssysteme werden beim Erzeuger-/Verbraucherproblem z. B. in einem Lösungsversuch die Sätze im Puffer über eine Semaphor-Variable gezählt. In diesem Fall würden z. B. alle  $V$ -Operationen am Ende des Erzeugerprozesses rückgängig gemacht werden, so dass der Verbraucherprozess die Sätze nicht mehr bearbeiten kann.
  - **0** in den meisten Fällen der Standardwert  
(falls der Prozess aufgrund eines Fehlers abbricht, kann er u. U. Betriebsmittel für andere Prozesse sperren, die dieselben Semaphore-Kollektionen benutzen)

- **Beispiel: Realisierung der P()- und V()-Operation von Dijkstra mit Hilfe der Semaphore von *System V IPC* (sem\_implement\_P\_V.c)**

```

...
#define SEM_KEY          IPC_PRIVATE          /* create private semaphore */
#define SEM_PERM         0600                /* access permissions       */

typedef int      semaphore;                  /* semaphore (handle) type  */
#if (_SEM_SEMUN_UNDEFINED == 1) || defined(SunOS)
    typedef union semun
    {
        int          val;                    /* cmd: SETVAL              */
        struct semid_ds *buf;                /* cmd: IPC_STAT, IPC_SET  */
        ushort       *array;                /* cmd: GETALL, SETALL     */
    } semunion;
#endif

semaphore init_sem (int value);              /* create/initialize semaphore */
void rel_sem (semaphore sem);                /* release semaphore         */
void P (semaphore sem);                      /* semaphore operations      */
void V (semaphore sem);

int main (void)
{
    semaphore      s;
    ...

    srand ((unsigned int) time ((time_t) NULL));
    s = init_sem (1);                          /* create sem with value 1 */
    for (i = 0; i < MAX_CHILD; ++i)
    {
        child_pid [i] = fork ();
        switch (child_pid [i])
        {
            case -1:                            /* error: no process created */
                ...
            case 0:                              /* child process              */
                /* up to Linux 2.0.x the process didn't terminate after
                 * LIFETIME seconds because "alarm()" and "sleep()" have
                 * used the same clock. This problem is solved in Linux 2.2.x
                 */
                alarm (LIFETIME);
                while (1)
                {
                    /* simulate some normal work */
                    P (s);
                    /* simulate some critical work */
                    V (s);
                }
                break;
            default:                             /* parent process            */
                if (i == (MAX_CHILD - 1))        /* all childs created ?     */
                {
                    /* yes -> wait for termination */
                    ...
                    printf ("All child processes have terminated.\n");
                    rel_sem (s);
                }
        }
    }
    /* end switch */
    /* end for */
    return 0;
}

```

```

semaphore init_sem (int value)
{
    union semun arg;                /* parameter for semctl      */
    int      semid;                /* semaphore handle         */

    semid = semget (SEM_KEY, 1, IPC_CREAT | SEM_PERM);
    if (semid == -1)
    {
        perror ("init_sem: semget failed");
        exit (ESEMGET);
    }
    arg.val = value;
    if (semctl (semid, 0, SETVAL, arg) == -1)
    {
        perror ("init_sem: semctl failed");
        exit (ESEMCTL);
    }
    return semid;
}

void rel_sem (semaphore sem)
{
    union semun tmp;                /* for Linux up to 2.0.x    */

    if (semctl (sem, 0, IPC_RMID, tmp) == -1)
    {
        perror ("rel_sem: semctl failed");
        exit (ESEMCTL);
    }
}

void P (semaphore sem)
{
    struct sembuf tmp;

    tmp.sem_num = 0;
    tmp.sem_op  = -1;
    tmp.sem_flg = 0;
    if (semop (sem, &tmp, (size_t) 1) == -1)
    {
        perror ("P: semop failed");
        exit (ESEMOP);
    }
}
...

```

- **Aufgabe 2-7:**

Implementieren Sie Lösung 1 des Philosophenproblems aus der Lehrveranstaltung *Betriebssysteme*. Die obige Simulation darf nicht benutzt werden! Nutzen Sie alle Möglichkeiten der Semaphore von *System V IPC*, d.h., nehmen Sie alle erforderlichen Gabeln in einer Semaphore-Operation auf.

## 2.6 Message Queues

- es gibt verschiedene Programmierschnittstellen
  - System V IPC (wird in diesem Kapitel vorgestellt)
    - ◆ benutzt einen *Schlüsselwert* zur Bestimmung der Identifikationsnummer der gewünschten Nachrichtenwarteschlange
  - POSIX IPC
  - Überblick über die Programmierschnittstellen

|                                    | POSIX message queues                                                                 | System V message queues |
|------------------------------------|--------------------------------------------------------------------------------------|-------------------------|
| Nachrichtenwarteschlange erzeugen  | mq_open ()                                                                           | msgget ()               |
| Operationen                        | mq_send ()<br>mq_receive ()<br>mq_timedsend ()<br>mq_timedreceive ()<br>mq_notify () | msgsnd ()<br>msgrcv ()  |
| Attribute lesen/ändern             | mq_setattr ()<br>mq_getattr ()                                                       | msgctl ()               |
| Nachrichtenwarteschlange entfernen | mq_close ()<br>mq_unlink ()                                                          | msgctl ()               |

- Möglichkeit zur Datenübertragung zwischen beliebigen Prozessen
- alle Nachrichten werden im Betriebssystemkern gespeichert  
(Beim Senden einer Nachricht wird sie aus dem Benutzerdatenbereich in den Betriebssystemdatenbereich kopiert und beim Empfangen umgekehrt wieder in den Benutzerdatenbereich.)
- Nachrichtenwarteschlangen sind nur für die Interprozesskommunikation auf einem Rechner geeignet und nicht für den Datenaustausch in einem Rechnernetz
- Nachrichtenwarteschlangen werden als FIFO-Warteschlangen realisiert  
(Einfach verkettete Liste mit je einem Zeiger auf den Anfang und das Ende der Warteschlange. Am Kopf wird gelesen und am Ende geschrieben.)
- der Kopf einer Nachrichtenwarteschlange wird durch folgende Datenstruktur repräsentiert (abhängig vom Betriebssystem; siehe *msg.h*)

```
struct msqid_ds {
    struct ipc_perm  msg_perm;           /* operation permission structure */
    struct msg       *msg_first;         /* first message on queue */
    struct msg       *msg_last;         /* last message in queue */
    ushort          msg_cbytes;         /* current number of bytes on queue */
    ushort          msg_qnum;           /* number of messages in queue */
    ushort          msg_qbytes;         /* max number of bytes on queue */
    ushort          msg_lspid;          /* pid of last msgsnd */
    ushort          msg_lrpid;          /* pid of last msgrcv */
    time_t          msg_stime;          /* last msgsnd time */
    time_t          msg_rtime;          /* last msgrcv time */
    time_t          msg_ctime;          /* last msgctl time */
};
```

diese Datenstruktur gibt es für jede *msqid*

- die Köpfe der einzelnen Nachrichten in der Warteschlange werden durch folgende Datenstruktur repräsentiert

```
struct msg {
    struct msg *msg_next;           /* next message on queue      */
    long      msg_type;             /* message type, must be >0  */
    short     msg_ts;              /* message text size          */
    char      *msg_spot;           /* message text address       */
};
```

- über den Nachrichtentyp können unterschiedliche Nachrichten in **einer** Warteschlange verwaltet werden  
(Nachrichtentypen kleiner gleich Null sind für *msgrcv* reserviert)
- es können Text- oder beliebig strukturierte Binärdaten übertragen werden  
(solange die Prozesse wissen, wie sie die Daten interpretieren müssen)

- der Benutzer übergibt oder empfängt die Nachrichten über folgende Datenstruktur (siehe *msg.h*)

```
struct msgbuf {
    long mtype;           /* type of message          */
    char mtext[1];      /* message text             */
};
```

da er im Allgemeinen längere Nachrichten senden will, kann er eine der beiden folgenden Methoden wählen

#### a) statische Nachrichtenpuffer

```
#define MAX_MSG_LEN ... /* max. Laenge einer Nachricht */

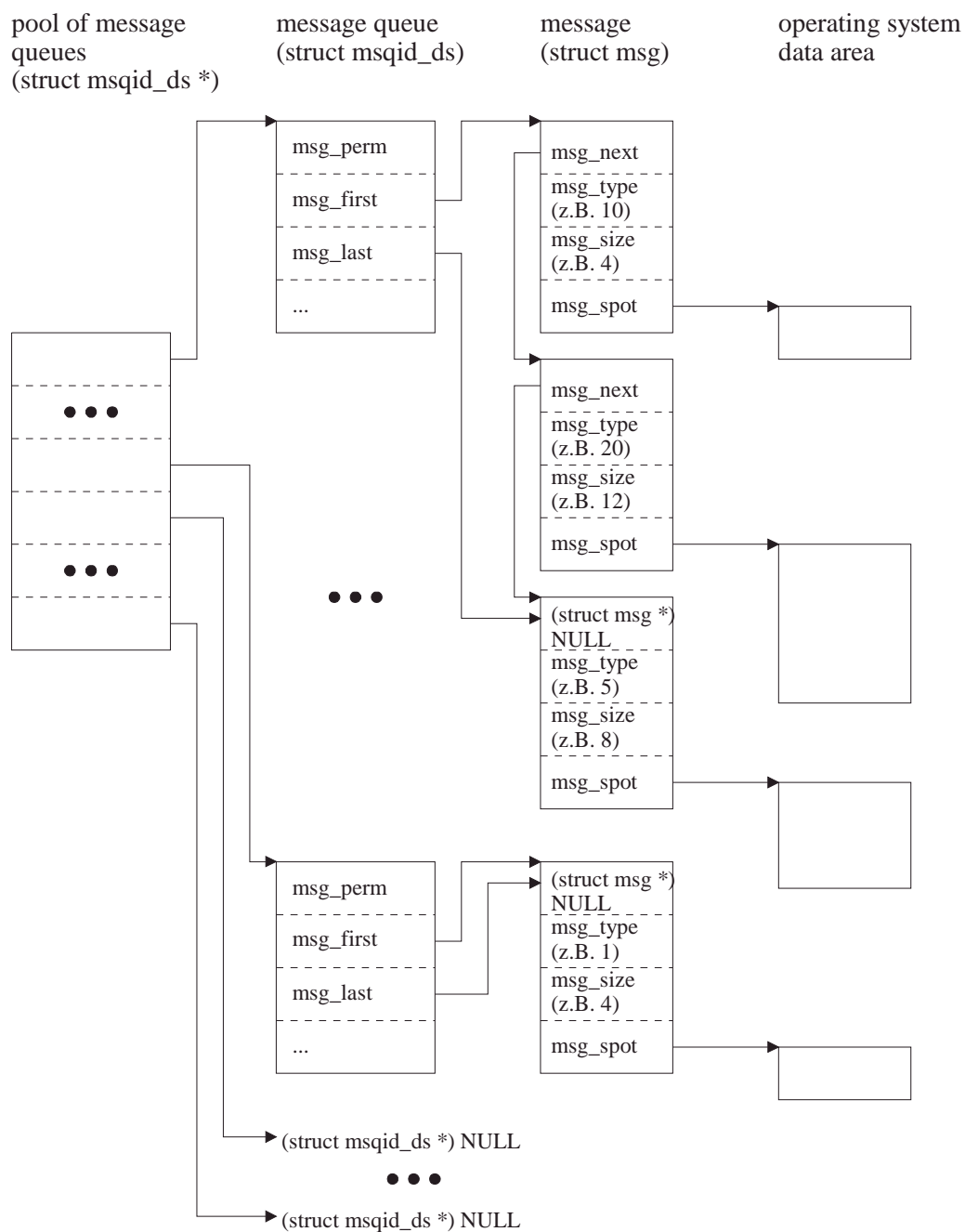
typedef struct my_msgbuf {
    /* ...                irgendwelche Komponenten          */
    long mtype;           /* mtype und mtext muessen in dieser */
    char mtext[MAX_MSG_LEN]; /* Reihenfolge hintereinander stehen */
    /* ...                irgendwelche Komponenten          */
} my_msg;
```

#### b) dynamische Nachrichtenpuffer

```
struct msgbuf *ptr_msgbuf; /* Zeiger auf Nachrichtenpuffer */
int          msg_size;    /* aktuelle Nachrichtengroesse */

msg_size = ...;
ptr_msgbuf = (struct msgbuf *) malloc (sizeof (struct msgbuf) -
  sizeof ptr_msgbuf->mtext + msg_size);
```

- Zusammenfassung der Datenstrukturen für Nachrichtenwarteschlangen



- die *msqid* wird als Index in das Ankerfeld benutzt
- Datenstruktur spart viel Speicher im Betriebssystemkern, solange keine oder wenige Nachrichtenwarteschlangen benutzt werden

- Vorgehensweise:
  - 1) Nachrichtenwarteschlange anlegen (*msgget*)
  - 2) Nachrichtenwarteschlange benutzen (*msgsnd*, *msgrcv*, *msgctl*)
  - 3) Nachrichtenwarteschlange freigeben (*msgctl*)
  
- allgemeine Kommandos für Nachrichtenwarteschlangen
  - int **msgctl** (int msqid, int cmd, struct msqid\_ds \*buf)
    - ◆ msqid      Identifikationsnummer der Nachrichtenwarteschlange (wird von *msgget* geliefert)
  
    - ◆ cmd      auszuführendes Kommando
      - IPC\_RMID    Nachrichtenwarteschlange löschen
      - IPC\_STAT    Werte des Warteschlangenkopfes liefern
      - ...
  
    - ◆ buf      Zeiger auf Datenpuffer, z. B. bei IPC\_STAT
  
  - die Datenstrukturen für Nachrichtenwarteschlangen müssen explizit gelöscht werden (IPC\_RMID)

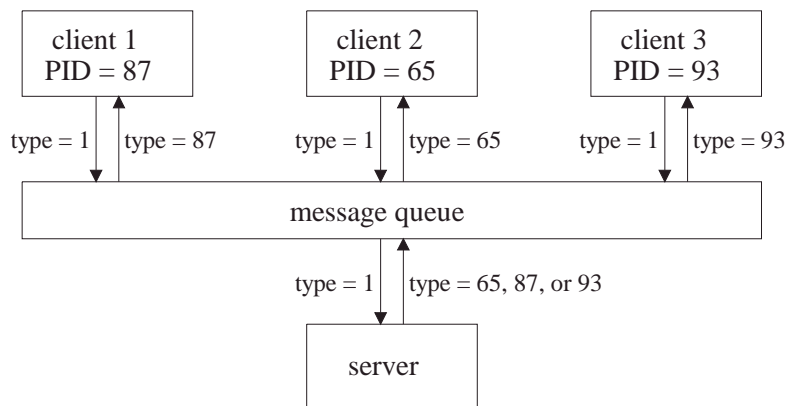
- **vor jedem *Logout* muss überprüft werden**, ob Nachrichtenwarteschlangen nicht gelöscht worden sind
  - Kommando *ipcs -q* gibt aus, welche Warteschlangen vorhanden sind
  - ggf. müssen die Warteschlangen manuell gelöscht werden
    - ◆ *ipcrm -q <msqid>* (IRIX, SunOS)
    - ◆ *ipcrm -Q <msqkey>* (IRIX, SunOS)
    - ◆ *ipcrm msg <msqid>* (Linux)
  
- Nachrichtenwarteschlange anlegen und die Identifikationsnummer zurückliefern
  - `int msgget (key_t key, int msgflg)`
    - ◆ `key` vom Benutzer wählbarer eindeutiger numerischer Schlüssel oder `IPC_PRIVATE`  
(bei `IPC_PRIVATE` kennen fremde Programme den Schlüssel nicht und können die Nachrichtenwarteschlange daher nicht benutzen; ein numerischer Schlüssel kann als symbolische Konstante oder Parameter auf der Kommandozeile definiert werden)
    - ◆ `msgflg` Zugriffsrechte und Funktionssteuerung
  - Funktionssteuerung:
    - `IPC_CREAT` neue Nachrichtenwarteschlange erzeugen
    - `IPC_EXCL` liefert einen Fehler, falls die Nachrichtenwarteschlange neu angelegt werden soll und der angegebene Schlüssel schon existiert

- Nachricht an die Nachrichtenwarteschlange senden
  - int **msgsnd** (int msqid, const void \*msgp, size\_t msgsz, int msgflg)
    - ◆ msqid Identifikationsnummer der Nachrichtenwarteschlange
    - ◆ msgp Zeiger auf einen vom Benutzer definierten Puffer, der als Erstes eine Komponente vom Typ *long* mit dem Nachrichtentyp und danach die Daten enthält  
(Linux verwendet hier *struct msgbuf \*msgp* (mindestens bis Version 2.2.x))
    - ◆ msgsz Länge des Datenbereichs im Puffer in Bytes  
(msgsz <= MSGMAX)
    - ◆ msgflg **IPC\_NOWAIT** falls die Nachricht nicht in die Warteschlange eingekettet werden kann, weil dann irgendwelche Begrenzungen überschritten werden, liefert *msgsnd* den Wert *-1* zurück und setzt *errno* entsprechend  
**0** der Prozess wird ggf. blockiert, bis die Nachricht in die Warteschlange eingefügt werden kann

- Nachricht aus der Nachrichtenwarteschlange entnehmen
  - int **msgrcv** (int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg)
    - ◆ msqid Identifikationsnummer der Nachrichtenwarteschlange
    - ◆ msgp Zeiger auf einen vom Benutzer definierten Puffer, in den die Nachricht kopiert wird. Der Puffer muss als Erstes eine Komponente vom Typ *long* für den Nachrichtentyp und danach den Bereich für die Daten enthalten  
(Linux verwendet hier *struct msgbuf \* msgp* (mindestens bis Version 2.2.x))
    - ◆ msgsz Länge des Datenbereichs im Puffer in Bytes
    - ◆ msgtyp gibt den Typ der Nachricht an, die empfangen werden soll
      - 0: die erste (älteste) Nachricht der Warteschlange wird entnommen (unabhängig vom Typ)
      - >0: die erste Nachricht mit dem Typ "msgtyp" wird entnommen
      - <0: die erste Nachricht mit dem kleinsten Typ, der kleiner oder gleich dem absoluten Wert von "msgtyp" ist, wird entnommen  
(Hiermit können auf einfache Weise Prioritätswarteschlangen implementiert werden. Die Priorität ist umso höher je kleiner der Nachrichtentyp ist. **Nachteil:** Da alle Nachrichten in einer verketteten Liste gespeichert sind, muss jedes Mal die gesamte Liste durchsucht werden.)

- ◆ **msgflg**      **IPC\_NOWAIT**      falls sich die Nachricht nicht in der Warteschlange befindet, liefert *msgrcv* den Wert *-1* zurück und setzt *errno* entsprechend  
  
**MSG\_NOERROR**      zu große Nachrichten sollen auf "msgsz"-Bytes gekürzt werden. Der abgeschnittene Teil der Daten geht verloren. Der Prozess wird hierüber nicht informiert.  
  
**0**      der Prozess wird ggf. blockiert, bis die Nachricht in der Warteschlange zur Verfügung steht
- ◆ **Rückgabewert:** Anzahl der im Puffer abgelegten Bytes oder *-1* im Fehlerfall

- mehrere unabhängige *Client*-Prozesse können mit einem *Server*-Prozess über eine Nachrichtenwarteschlange im *Multiplex*-Betrieb Daten austauschen, wenn sie als Nachrichtentyp z.B. ihre PID benutzen und den Nachrichtentyp für den *Server*-Prozess kennen



- **Beispiel**

Realisierung eines Prozesses, der Nachrichten in einer Warteschlange ablegt und eines weiteren Prozesses, der die Nachrichten aus der Warteschlange entnimmt und auf dem Bildschirm ausgibt.

– gemeinsame *Header*-Datei (msg\_definitions.h)

```

#define MBUF_KEY      (key_t) 0x2412) /* arbitrary value          */
#define MBUF_PERM     (0666)         /* "read/write" for everybody */
#define MAX_MSG_LEN   16             /* max. message length        */

typedef struct my_msgbuf {
    int    msg_len;                  /* message length in bytes    */
    long   msg_type;                 /* message type, > 0          */
    char   msg_data [MAX_MSG_LEN];  /* message text                */
} my_msg;

/* Linux 2.6.x or newer, Cygwin, and Darwin don't support
 * "struct msgbuf"
 */

#if defined(Cygwin) || defined(Darwin) || defined(Linux)
    struct msgbuf {
        long   mtype;                /* message type                */
        char   mtext [1];            /* message text                */
    };
#endif
  
```

– Nachricht an Nachrichtenwarteschlange senden (msg\_send.c)

```

...
int main (int argc, char *argv[])
{
    int          msqid,          /* message queue id          */
              msize,          /* message length           */
              ret_val,        /* return value of a function */
              another;        /* another message?         */
    long        msgtyp;        /* message type             */
    struct msgbuf *msgp;      /* pointer to message buffer */
    char        buffer[MAX_LEN + 2]; /* +2 for '\n' and '\0'     */
    signed char tmp;          /* in IRIX "char" is unsigned */

    msqid = msgget (MBUF_KEY, IPC_CREAT | MBUF_PERM);
    if (msqid < 0)
    {
        ...
    }
    printf ("\n\nYou can put a message into a message queue.\n\n");

    do
    {
        do
        {
            printf ("Please enter a message type (1,...,1000): ");
            scanf (" %ld", &msgtyp);
            while (((tmp = getchar ()) != '\n') && (tmp != EOF))
            {
                ; /* clean up after "scanf ()" */
            }
        } while ((msgtyp < 1) || (msgtyp > 1000));
        printf ("Please enter your message (at most %d characters): ",
                MAX_LEN);
        if (fgets (buffer, MAX_LEN + 2, stdin) == NULL)
        {
            buffer[0] = '\0';
        }
        msize = strlen (buffer) + 1; /* +1 for '\0' */
        if ((strchr (buffer, '\n') == NULL) && (msize > MAX_LEN))
        {
            buffer[MAX_LEN] = '\n';
            buffer[MAX_LEN + 1] = '\0';
        }
        /* allocate dynamic message buffer and create message */
        msgp = (struct msgbuf *) malloc (sizeof (struct msgbuf) -
                                       sizeof msgp->mtext + msize);

        if (msgp == NULL)
        {
            ...
        }
        msgp->mtype = msgtyp;
        strncpy (msgp->mtext, buffer, (size_t) msize);
        /* put message into message queue */
        ret_val = msgsnd (msqid, msgp, (size_t) msize, 0);
        if (ret_val < 0)
        {
            ...
        }
        free (msgp);
        printf ("\nDo you want to send ...? ");
        ...
    } while (another != 0);
}

```



**Aufgabe 2-8:**

Implementieren Sie einen *Broadcast-Server*. Ein *Client* kann sich beim *Server* registrieren und später wieder abmelden. Beide Aktionen werden bestätigt. Der *Server* darf eine Registrierung ablehnen, wenn sich bereits eine vorgegebene, maximale Anzahl von *Clients* registriert hat. Ein registrierter *Client* kann dem *Server* Nachrichten senden, die seinen "Namen" und einen Text enthalten. Der Text wird vom Benutzer über die Tastatur eingegeben. Jeder Text ist durch einen Zeilenvorschub abgeschlossen. Der *Server* sendet jede empfangene Nachricht an alle registrierten *Clients*. Jeder *Client* gibt die empfangenen Nachrichten in geeigneter Form aus. Nachrichtenausgabe und Texteingabe sollen parallel erfolgen können, d. h., dass der *Client* z. B. alle zwei Sekunden überprüft, ob eine Nachricht oder ein vollständiger Text vorliegt (er darf bei keiner Aktion blockiert werden).

Der *Server* kann durch das Kommando *quit* oder *exit* (in beliebiger Schreibweise) beendet werden. Bevor der *Server* sich selbst beendet, sendet er allen registrierten *Clients* ein Ende-Kommando, das die *Clients* bestätigen. Falls in dieser Phase weitere Registrierungswünsche eingehen, werden sie abgelehnt. Der *Server* darf sich erst beenden, wenn er alle Quittungen auf sein Ende-Kommando empfangen hat. Ein *Client*, der ein Ende-Kommando empfängt oder dessen Registrierung abgelehnt wird, beendet sich nach einer entsprechenden Meldung.

Starten Sie den *Server* und die *Clients* in verschiedenen Fenstern. Zeigen Sie, dass Ihre Programme das oben beschriebene Protokoll einhalten.