Interprozeßkommunikation unter Unix

- Eine Einführung -

Stefan Freinatis
Gerhard Mercator Universität –Gesamthochschule– Duisburg
1994 / 1996

Inhaltsverzeichnis

1. Vorwort und Einführung

Prozesskommunikation ist die Fähigkeit, zwischen Prozessen Daten auszutauschen und die Ausführung zu synchronisieren. Dafür stehen unter Unix verschiedene Mechanismen zur Verfügung. Die wichtigsten Mechanismen werden hier vorgestellt und anhand von Fallbeispielen einführend erläutert. Dies setzt Kenntnisse des Lesers im Betriebssystem UNIX, sowie in der Programmiersprache C voraus.

Viele der vorgestellten Mechanismen verfügen über Optionen, deren Behandlung den Rahmen dieser Einführung sprengen würde. Aus diesem Grund beschränken sich die Beispiele auf gängige Anwendungen der Prozeßkommunikation. Der Schritt ins Detail wird nach erfolgreicher Lektüre dann nicht mehr schwer fallen.

Jeder vorgestellte Mechanismus wird anhand von C-Programmen besprochen. Soweit nötig, sind die dazu erforderlichen Systemaufrufe beschrieben. Es empfiehlt sich dennoch, ein Handbuch über die Systemaufrufe zur Verfügung zu haben.

Alle Beispiele wurden auf einer SUN[1] geschrieben, dabei wurde der Standard *cc*-Compiler verwendet, der auf allen Unix-Systemen verfügbar ist. Das Erzeugen der ausführbaren Datei geschieht durch `make <dateiname>`, wobei die Extension nicht mit angegeben wird. Ein Makefile ist nicht erforderlich.

In den Beispielen wird die Datei `defs.h` eingebunden, sie wird im Anhang A besprochen. Im Anhang B werden ein paar Hilfsprogramme vorgestellt, die beim Experimentieren mit den Beispielen nützlich sein können. Alle Beispiele und die Hilfsprogramme befinden sich in Form von Quelltexten auf beiliegender Diskette.

Diese Arbeit entstand im Rahmen meiner Arbeit als studentische Hilfskraft im Fachgebiet Datenverarbeitung an der Mercator Universität GH Duisburg. Ich danke Sakis Papathanasiou und Joachim Zumbrägel, daß sie mich diese Arbeit selbständig haben durchführen lassen. Diese Arbeit sei Ihnen gewidmet.

Stefan Freinatis, Duisburg, 9. Mai 1994

---

[1] Sun-OS 4.2.1

Nachtrag

Aufgrund der Nachfrage habe ich die erste Version etwas überarbeitet und die Hilfsprogramme als Listings in den Anhang aufgenommen, da erfahrungsgemäß die Diskette bei der Weitergabe auf der Strecke bleibt. Eigentlich müßte eine Generalüberholung der Beispielprogramme erfolgen, da diese nicht dem heutigen Programmierstandard entsprechen (ANSI-C). Die Beispiele und Hilfsprogramme, zumeist abgewandelte Beispiele aus Büchern oder Manuals, sind noch im Kernighan & Ritchie Stil gehalten, der heute überholt ist. So werden beispielsweise Konstanten heute nicht mehr mit `#define`, sondern mit `const` festgelegt. Ebenso werden Funktionsargumente komplett im Funktionskopf definiert. Dennoch haben die Beispiele nichts von ihrer Aktualität verloren. Durch die Verbreitung von Linux hat das Betriebsystem Unix mittlerweile Einzug in den privaten Bereich gefunden. Der Kreis der Unix-Softwareentwickler wird zunehmend größer und damit das Interesse an der Interprozeßkommunikation.
Empfehlenswert für das tiefere Verständnis sind die Werke von Richard Stevens (siehe Literatur S.60). Er behandelt u.a. auch die Kommunikation über Netzwerke und geht genauer auf die Unterschiede zwischen System V und BSD ein.

Stefan Freinatis, Duisburg, 1.03.1996

2. Programme und Prozesse

Ein Programm besteht aus Anweisungen und Daten, die in einer Datei auf Platte gespeichert sind. Diese Datei ist als ausführbar markiert und nach bestimmten Regeln angeordnet. Um ein Programm auszuführen muß der Betriebssystemkern einen Prozeß erzeugen. Der Prozeß ist eine Umgebung, in der das Programm ablaufen kann. Er besteht aus 3 Segmenten: Anweisungssegment, Benutzer-Datensegment und System-Datensegment. Die Programmdatei wird dazu benutzt, das Anweisungssegment und das Benutzer-Datensegment zu initialisieren. Danach gibt es zwischen dem Programm und dem Prozeß, der dieses Programm gerade ausführt, keine feste Verbindung mehr. In der Terminologie objektorientierter Programmentwicklung kann man auch sagen: Ein Prozess ist eine Instanz eines Programmes. Es ist möglich, daß mehrere gleichzeitig ablaufende Prozesse von dem gleichen Programm instantiiert werden.

Jeder Prozeß erhält vom Kern eine eindeutige positive ganze Zahl, die auch Prozeßnummer (`pid`) genannt wird. Ein neuer Prozeß entsteht immer als Sohn des gerade laufenden Prozesses. Dieser Sohn-Prozeß erhält eine neue Prozeßnummer und erbt die meisten System-Attribute des Vaterprozesses. Hat beispielsweise der Vater noch irgendwelche Dateien geöffnet, so sind diese auch für den Sohn geöffnet.

Ein Prozeß registriert in seinen System-Daten neben der eigenen Prozeßnummer auch die des Vaters (`ppid`). Falls der Vaterprozeß zu einem gegebenen Zeitpunkt nicht mehr existiert, wird der ppid auf 1 gesetzt. Die Prozeßnummer 1 beschreibt *init*, den Urprozeß aller Prozesse in einem Unix-System. Dieser erbt also alle Waisen.

Für umfangreichere Anwendungen (z.B. Datenbank), bei denen die Arbeit auf mehrere Prozesse verteilt wird, können solche Prozesse zu einer Prozeßgruppe zusammengefaßt werden. Ein bestimmter Prozeß wird zum Prozeßgruppenleiter. Damit alle Prozesse wissen, wer dies ist, haben sie in ihrem System-Datensegment auch dessen Prozeßnummer gespeichert (`pgrpid`). Zusammengefaßt besitzt ein Prozeß also drei Prozeßnummern:

`pid`	Die Prozeßnummer des eigenen Prozesses. Dies ist, wie auch die folgenden Prozeßnummern, eine eindeutige positive Integerzahl.
`ppid`	Die Prozeßnummer des Vater-Prozesses. Ist diese gleich 1, so existiert der Vater-Prozeß nicht mehr.

`pgrpid` Die Prozeßnummer des Prozeßgruppenführers. Falls diese identisch mit der `pid` ist, ist dieser Prozeß der Gruppenführer.

Das folgende Programm `prozess.c` gibt diese drei Prozeßnummern aus. Zu beachten ist, daß `getpgrp()` einen Parameter erwartet (nur in 4.3BSD). Hat dieser Parameter den Wert 0, ist damit der eigene Prozeß gemeint. In diesem Fall lautet die Anfrage also: Wer ist mein Prozeßgruppenführer. Als Parameter kann auch jede andere Prozeßnummer übergeben werden. In diesem Fall lautet die Anfrage: Wer ist der Prozeßgruppenführer von <parameter>.

```
/* prozess.c    Ausgabe der Prozessnummern */

#include <stdio.h>
main()
{
  printf("Prozeßnummer pid = %d\n", getpid());
  printf("Prozeßnummer des Vaters ppid = %d\n", getppid());
  printf("Prozeßnummer des Gruppenfuehrers pgrp = %d\n", getpgrp(0));
}
```

Wir erhielten folgende Ausgabe:

```
Prozeßnummer pid = 5501
Prozeßnummer des Vaters ppid = 5368
Prozeßnummer des Gruppenfuehrers pgrp = 5501
```

Prozesse werden mit dem Systembefehl `fork()` erzeugt. `fork()` liefert einen Rückgabewert mit dem festgestellt werden kann ob es sich um den Vater- oder den neuen Sohnprozeß handelt.

```
int fork()

/* liefert Prozeßnummer und 0 bei Erfolg, sonst -1 */
```

Ist der Rückgabewert 0, so handelt es sich um den Sohnprozeß. Dieser Prozeß hat das Anweisungssegment, das Benutzer-Datensegment und das Systemsegment vom Vaterprozeß geerbt. Dabei gibt es allerdings ein paar Unterschiede. Die Prozeßnummer ist von der des Vaters verschieden und die gesammelten Ausführungszeiten des Sohnes sind auf Null zurückgesetzt worden, da dieser noch am Beginn des Lebens steht.
Ist der Rückgabewert eine positive Integerzahl, so handelt es sich bei dem Prozeß um den Vaterprozeß. Die positive Integerzahl ist die Prozeßnummer des Sohnes. In dem folgenden Beispiel wird der Rückgabewert von `fork()` nicht verwendet. Mit der Anweisung `fork()` wird der neue Prozeß erzeugt. Beide Prozesse laufen dann parallel nebeneinander und geben ihre jeweilige `pid` aus. Auf der Standardausgabe erscheinen also zwei Ausgaben.

```
/* fork.c   Erzeugen eines Sohn-Prozesses */

#include <stdio.h>

void main()
{
   printf("Start of test\n");                  /* Kurze Meldung */
   fork();                                     /* Prozess duplizieren */
   printf("Meine pid ist: %d\n", getpid());    /* Ausgabe (insgesamt zweimal!) */
}
```

Die Reihenfolge der Ausgaben hängt vom Zufall ab, da nicht im Voraus bestimmt werden kann, welchem Prozeß der Scheduler zuerst Rechenzeit zuweist. So sah unsere Ausgabe aus:

```
Start of test
Meine pid ist: 2023
Meine pid ist: 2024
```

Im nächsten Beispiel `fork1.c` wird der Rückgabewert von `fork()` verwendet, um zwischen Vater und Sohn unterscheiden zu können. Als erstes gibt der Prozeß seine `pid` aus. Im Grunde handelt es sich hier noch gar nicht um einen Vater-Prozeß, denn zu diesem Zeitpunkt besitzt er noch keinen Sohn. Bei `fork()` wird dann der Sohn-Prozeß erzeugt. Durch den Rückgabewert 0, den nur der Sohn erhält, kann dieser sich auch als Sohn identifizieren und die if-Anweisung passieren. Dort gibt der Sohn seine `pid` aus und beendet sich mit `exit()`. Der Vater umgeht die if-Anweisung, weil dessen Rückgabewert von `fork()` die Prozeßnummer des Sohnes und damit von 0 verschieden ist. Er gibt seine `pid` ebenfalls aus und endet.

```
/* fork1.c      Erzeugen eines Sohn-Prozesses */

#include <stdio.h>

/* Der Prozess splittet sich ab fork(). Der Sohn erhaelt den Returnwert 0 und kann so
   die if-Anweisung passieren. Der Sohn wird dann mit exit() beendet. Der Vater erhaelt
   von fork() die Prozeßnummer des Sohnes. Er uebergeht die if-Anweisung. */

void main()
{
   int result;

   printf("Dies ist der Vaterprozess, meine pid ist: %d\n", getpid());
   result = fork();
   if (result == 0) {        /* Hier nur Sohn */
    printf("Hier ist der Sohn-Prozess, meine pid ist: %d\n", getpid());
    exit(0);
   }

   /* Hier nur Vater */
   printf("Hier ist der Vaterprozess, meine pid ist: %d\n", getpid());
}
```

Auf der Standardausgabe erscheint der folgende Ausdruck:

```
Dies ist der Vaterprozess, meine pid ist: 189
Hier ist der Vaterprozess, meine pid ist: 189
Hier ist der Sohn-Prozess, meine pid ist: 190
```

Im letzten Beispiel `fork2.c` wird gezeigt, wie Filedeskriptoren und Variablen vererbt werden. Nach Erzeugung des Sohn-Prozesses können beide Prozesse auf ein- und dieselbe Datei zugreifen. Bei der Integervariablen `i` verhält es sich anders: Der Sohn erhält eine Kopie der Variablen, eine Änderung wirkt sich also nicht auf die Variable gleichen Namens des Vaters aus. Umgekehrt ist es ebenso.

```
/* fork2.c      Vater- und Sohnprozeß */

#include <stdio.h>
#include <fcntl.h>
#include "defs.h"

#define TEXT "1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define SLEEPTIME 1

void main()
{
   int result;
   int i = 0;
   int fd;
   char buf[32];
   for (i = 0; i < 32; i++)                                  /* buf mit '\0' vorbelegen */
    buf[i] = 0x0;

   printf("Meine pid ist: %d\n", getpid());
   if ((fd = open("testfile", O_RDWR | O_CREAT, 0666)) == -1)
    fatal("open");
   if (write(fd, TEXT, sizeof(TEXT)) == -1)
    fatal("write");
   if (lseek(fd, 0, 0) == -1)
    fatal("lseek");

   printf("Der Filedeskriptor fd ist: %d\n", fd);
   printf("Die Variable i hat den Wert: %d\n", i);
   printf("Jetzt wird der Sohn-Prozess erzeugt\n");

   result = fork();
   if (result == 0) {                                        /* Beginn Sohn */
    printf("Sohn: Meine pid ist: %d\n", getpid());
    printf("Sohn: Der vererbte Filedeskriptor fd ist: %d\n", fd);
    if (read(fd, buf, 5) == -1)
     syserr("Sohn read");
    printf("Sohn: Aus der Datei gelesen: %s\n", buf);
    i = 1;
    printf("Sohn: Die Variable i hat den Wert: %d\n", i);
    close(fd);
    exit(0);
   }                                                         /* Ende Sohn */

   /* Vater */

   sleep(SLEEPTIME);                                         /* Vater wartet etwas */
   printf("Vater: Meine pid ist: %d\n", getpid());
   if (read(fd, buf, 5) == -1)
     syserr("Vater read");
```

```
 printf("Vater: Aus der Datei gelesen: %s\n", buf);
   printf("Vater: Die Variable i hat den Wert: %d\n", i);
   close(fd);
}
```

Dies ist die Ausgabe, die das Programm erzeugt hat:

```
   Meine pid ist: 2923
   Der Filedeskriptor fd ist: 3
   Die Variable i hat den Wert: 0
   Jetzt wird der Sohn-Prozess erzeugt
   Sohn: Meine pid ist: 2924
   Sohn: Der vererbte Filedeskriptor fd ist: 3
   Sohn: Aus der Datei gelesen: 12345
   Sohn: Die Variable i hat den Wert: 1
   Vater: Meine pid ist: 2923
   Vater: Aus der Datei gelesen: 67890
   Vater: Die Variable i hat den Wert: 0
```

Der Prozeß gibt als erstes seine `pid` aus und erzeugt im aktuellen Verzeichnis eine Datei, in der eine Folge von Zahlen und Buchstaben abgelegt ist. Der Dateizeiger wird mittels `lseek()` auf das erste Zeichen zurückgestellt. Zur Information werden noch Filedeskriptor und Inhalt der Variablen `i` ausgegeben. Dann wird der Sohn-Prozeß erzeugt. Der Sohn gibt seine Prozeßnummer und den vererbten Filedeskriptor aus. Dann liest er 5 Zeichen aus der Datei und gibt diese ebenfalls aus. Um die Unabhängigkeit seiner Variablen `i` von der des Vaters zu demonstrieren wird diese auf den Wert 1 gesetzt und ausgegeben. Während dieser ganzen Zeit schläft der Vater-Prozeß. Nach `fork()` hat er die if-Anweisung übersprungen und wurde mittels `sleep()` für die Zeit `SLEEPTIME` schlafen gelegt. Nach dem Aufwachen gibt der Vater seine `pid` aus und liest ebenfalls 5 Zeichen aus der Datei. Dabei liest der Vater ab der Position, in der der Sohn den Dateizeiger hat stehen lassen. Er gibt die 5 Zeichen, sowie den Inhalt seiner Variablen `i` aus.

Die Sleep-Anweisung wurde nur eingeführt, um den Sohn in Ruhe seine Sachen erledigen zu lassen. `SLEEPTIME` kann auch auf den Wert 0 gesetzt werden. In diesem Fall laufen beide Prozesse parallel und die Ausgabe wird zerstückelt. Hier wird ersichtlich, wie wichtig die Prozeßkommunikation ist, um über gemeinsame Betriebsmittel (z.B. die Standardausgabe oder Dateien) zu verfügen ohne sich gegenseitig ins Gehege zu kommen.

3. Pipes

Pipes ermöglichen den Datentransport zwischen Prozessen nach einer Art Einbahnstraßenprinzip. Dabei werden Daten nach der *first-in-first-out* (FIFO) Methode an den anderen Partner am Ende der Pipe geschickt. Eine Pipe ist **keine** Datei, obwohl sie eine Inode besitzt. Sie hat weder einen Namen, noch existieren Verweise auf sie. Das Lesen und Schreiben einer Pipe geschieht trotzdem mit den für Dateien üblichen Befehlen `read()` und `write()`, jedoch mit fundamentalen Unterschieden. Wenn der Leser den Schreiber überholt, wartet der Leser auf weitere Daten. Wenn der Schreiber dem Leser zu weit vorauseilt und damit die vom System abhängige Maximalgröße einer Pipe überschreiten würde, wartet er bis die Daten zumindest teilweise gelesen wurden. Hierin besteht die eigentliche Synchronisation zwischen den beiden Prozessen. Jeder Prozeß am Ende einer Pipe 'merkt', ob der jeweilige Partner gelesen bzw. geschrieben hat. Einmal geschriebene Daten sind, nachdem sie vom anderen Partner gelesen wurden, verloren. Ein Vor- oder Rückspulen mittels `lseek()` ist nicht möglich und auch nicht erwünscht (FIFO-Prinzip).

Der Einsatz von Pipes beschränkt sich nur auf verwandte Prozesse. Üblicherweise sind dies Vater und Sohn. Dies ist auch nur dann möglich, wenn der Vater **vor** Erzeugung eines Sohn-Prozesses eine Pipe kreiert. Die erhaltenen Filedeskriptoren werden dann bei der Erzeugung des Sohnes automatisch vererbt. Nach Generierung eines Sohn-Prozesses ist der Aufbau einer Pipe zur Prozesskommunikation zwischen den beiden Prozessen nicht mehr machbar. Es besteht dann keine Möglichkeit mehr einen Filedeskriptor an den anderen Prozeß zu übergeben. Wer daran denkt, einen Filedeskriptor über eine bereits bestehende Pipe an den Partner-Prozeß zu übergeben wird enttäuscht: Der Filedeskriptor hat dort keine Gültigkeit. Eine Pipe muß also vor der Generierung eines Sohn-Prozesses erzeugt worden sein.

Eine Pipe wird durch den Systemaufruf `pipe()` erzeugt:

```
int pipe(fd)                                    /* Pipe erzeugen */
    int fd[2];

/* liefert 0 bei Erfolg oder -1 bei Fehler */
```

Dabei werden zwei Dateideskriptoren zurückgeliefert. `fd[0]` wird geöffnet zum Lesen, `fd[1]` zum Schreiben.

Wie schon erwähnt, werden zum Lesen und Schreiben einer Pipe die Ein/Ausgabe Systemaufrufe für gewöhnliche Dateien verwendet. Diese verhalten sich bei Pipes jedoch anders als bei normalen Dateien. Die Einzelheiten dazu folgen hier:

`write()` Die Daten werden nach dem FIFO-Prinzip hintereinander abgelegt. Ist die Pipe voll, blockiert `write()` solange, bis durch `read()` genug alte Daten gelesen wurden. Dieses Blockieren kann mit dem Systemaufruf `fcntl()` in Verbindung mit dem `ON_DELAY` - Flag unterbunden werden. In diesem Fall liefert write als Resultat den Wert 0. Die einzige Möglichkeit ein *end-of-file* in die Pipe zu schreiben besteht darin, den Filedeskriptor `fd[1]` zu schließen.

`read()` Die Daten werden in der Reihenfolge aus der Pipe gelesen, in der sie dort eintrafen. Ist die Pipe leer, blockiert `read()` solange, bis wieder Daten verfügbar sind oder der Schreibzugriff vom anderen Prozess geschlossen wurde. Auch dieses Blockieren ist mittels des `fcntl()`-Aufrufes mit dem `ON_DELAY`-Flag unterbindbar. In diesem Fall würde `read()` mit dem Resultat 0 zurückkehren, falls keine Daten verfügbar sind. Hier gibt es dann allerdings einen Haken: Man kann nicht unterscheiden, ob der Rückgabewert 0 besagt, daß nur keine Daten verfügbar sind oder ob gar der Schreibzugriff geschlossen wurde. Denn der Rückgabewert 0 bedeutet ebenfalls Dateiende.

`close()` Dieser Aufruf hat bei einer Pipe mehr Auswirkungen als bei einer normalen Datei. Wird der Filedeskriptor für den Schreibzugriff `fd[1]` geschlossen, erhält der Leseprozeß automatisch ein *end-of-file*. Wird der Filedeskriptor für den Lesezugriff `fd[0]` geschlossen, so verursacht ein `write()` auf der anderen Seite einen Fehler.

Anzumerken ist noch, daß `read()` und `write()` atomar arbeiten[1]. Das bedeutet, daß diese Aufrufe nicht durch Signale unterbrochen werden. Dies gilt nicht nur für Pipes, sondern auch für alle anderen Anwendungen, in denen `read()` und `write()` benutzt werden.

Im folgenden Beispiel `pipe.c` wird gezeigt, wie eine Pipe erzeugt wird. Dann werden zwei Daten in Form zweier Strings in die Pipe geschrieben und anschließend wieder ausgelesen. In Verbindung mit nur einem einzigen Prozess ist die Verwendung von Pipes nicht besonders sinnvoll. Hier soll auch nur der erste Kontakt mit Pipes hergestellt werden.

---

1 Dies ist zumindest auf den gängigen UNIX-System der Fall.

```
/* pipe.c      Schreiben in Pipe, Lesen aus Pipe */

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include "defs.h"

#define SATZ1 "Dies ist der erste Satz."
#define SATZ2 "Dies ist Datensatz 2."

void main()
{

   int fd[2];
   char buf[256]

   /* Pipe erzeugen */

   if (pipe(fd) == -1) fatal("Pipe Fehler");

   printf("Filedeskriptor 'Lesen fd[0]' ist: %d\n", fd[0]);
   printf("Filedeskriptor 'Schreiben fd[1]' ist: %d\n", fd[1]);

   /* In die Pipe schreiben */

   if (write(fd[1], SATZ1, sizeof(SATZ1)) == -1) syserr("write");
   if (write(fd[1], SATZ2, sizeof(SATZ2)) == -1) syserr("write");

   /* Aus der Pipe lesen */

   if (read(fd[0], buf, sizeof(SATZ1)) == -1) syserr("read");
   printf("Der erste Datensatz lautet: %s\n", buf);
   if (read(fd[0], buf, sizeof(SATZ2)) == -1) syserr("read");
   printf("Der zweite Datensatz lautet: %s\n", buf);


   close(fd[0]);
   close(fd[1]);
}
```

Wie üblich, muß bei `write()` die Anzahl der zu schreibenden Byte angegeben werden. Ebenso benötigt `read()` die Angabe der zu lesenden Byte. Es ist also nicht so, daß eine Pipe zwischen den unterschiedlich langen Daten heimliche Datentrenner einfügt und somit pro `read()` ein Datensatz, egal welcher Länge, gelesen würde. Die Wahl von Zeichenketten als Daten wurde hier der Einfachheit halber gewählt. Selbstverständlich kann jede Art von Daten durch eine Pipe vermittelt werden.

Im folgenden Beispiel wird gezeigt, wie der Vater-Prozeß dem Sohn-Prozeß durch eine Pipe eine Nachricht schickt.

```
/* pipe1.c      Nachricht ueber Pipe schicken */

#include <stdio.h>
#include "defs.h"

#define FIXSIZE 80

void main()
{
   int result;
   int fd[2];
   char buf[256];

   if (pipe(fd) == -1)                          /* Pipe erzeugen */
    fatal("Pipe Fehler");
```

```
   result = fork();                              /* Prozess duplizieren */

   if (result == 0) {
    printf("Hier ist der Sohn-Prozess, mein pid ist: %d\n", getpid());
    close(fd[1]);

    if (read(fd[0], buf, FIXSIZE) == -1)       /* Warten auf Nachricht vom Vater */
     syserr("read");
    printf("Sohn: Nachricht vom Vater war: %s\n", buf);
    exit(0);
   }

   close(fd[0]);
   printf("Hier ist der Vaterprozess, mein pid ist: %d\n", getpid());

   /* Schicken einer Nachricht an Sohn */

   if (write(fd[1], "Hallo!", FIXSIZE) == -1)
    syserr("write");
}
```

Der Vater erzeugt eine Pipe und ruft danach den Sohn ins Leben. Dieser gibt zur Information seine pid aus und schließt den Schreibzugriff der Pipe. Das wird gemacht, um Filedeskriptoren zu sparen. Der Sohn benutzt die Pipe in diesem Beispiel nur zum Lesen, der Vater benutzt die Pipe nur zum Schreiben. Es kann auch umgekehrt sein, jedoch können nicht beide Prozesse mit einer Pipe Lesen und Schreiben. Das ist technisch zwar möglich, aber logisch problematisch. Denn wenn ein Prozeß in die Pipe schreibt und kurz darauf liest, bekommt er seine eigenen Daten zurück. Dies wurde ja bewußt im Beispiel `pipe.c` gemacht. Es wäre ein ausgeklügeltes weiteres Management nötig, welches entscheidet, wann wer zu Lesen und zu Schreiben hat. Aus diesem Grunde werden Pipes immer nur in eine Richtung benutzt. Selbstverständlich kann man zwei Pipes benutzen um bidirektionalen Verkehr zu ermöglichen. Aber auch dann ist gutes Management erforderlich, damit es nicht zu einer Deadlock-Situation kommt. Eine Deadlock-Situation tritt z.B. dann ein, wenn beide Prozesse auf Daten des jeweilig anderen warten. Da jeder Prozeß wartet, kann er nicht schreiben. Diese Situation ist dann nur noch von außen durch Abbruch der Prozesse zu beenden. Während also der Sohn seine Arbeit macht, schließt auch der Vater seinen Lesezugriff auf die Pipe. Der Sohn ist mittlerweile in den `read()`-Aufruf geraten und wartet, da es sich um blockiertes Lesen handelt, auf eine Nachricht vom Vater. Der Vater schickt schließlich den String "Hallo!" an den Sohn. Dieser gibt den String aus und beendet sich durch `exit()`. Der Vater endet ebenfalls durch Passieren des `main()`-Blocks.

Die Ausgabe auf der Standardausgabe sah wie folgt aus:

```
Hier ist der Sohn-Prozess, meine pid ist: 3055
Hier ist der Vaterprozess, meine pid ist: 3054
Sohn: Nachricht vom Vater war: Hallo!
```

In dem letzten Listing soll ein Beispiel für nicht-blockierendes Lesen gezeigt werden. Der Grundaufbau entspricht dem Beispiel `pipe1.c`, jedoch wurde hier mittels `fcntl()` der `read()`-Aufruf so modifiziert, daß er nicht auf Daten des Vaters wartet, sondern gleich zurückkehrt. Bei 4.3BSD kehrt `read()` allerdings nicht mit dem Rückgabewert 0 (wie bei System V), sondern mit -1 zurück. Der Sohn gibt die Meldung "Ich warte.." aus und schläft eine Sekunde. Danach probiert er den Lesevorgang erneut. Der Vater wurde zwischenzeitlich für 3 Sekunden schlafen gelegt. Erst dann schickt er dem Sohn die Nachricht. Hier folgt das Listing von `pipe2.c`:

```
/* pipe2.c      Nicht-blockierendes Lesen */

#include <stdio.h>
#include <fcntl.h>
#include "defs.h"

#define FIXSIZE 80


void main()
{
   int result;
   int fd[2];
   char buf[256];


   if (pipe(fd) == -1)                          /* Pipe erzeugen */
    fatal("Pipe Fehler");

   result = fork();                             /* Prozess duplizieren */

   if (result == 0) {                           /* Sohn */
    int flags;

    printf("Hier ist der Sohn-Prozess, meine pid ist: %d\n", getpid());
    close(fd[1]);
    if ((flags = fcntl(fd[0], F_GETFL, 0)) == -1)          /* Flags holen */
     syserr("fnctl");
    if ((fcntl(fd[0], F_SETFL, flags | O_NDELAY)) == -1)   /* Flags setzen */
     syserr("fnctl");

    loop {
     switch (read(fd[0], buf, FIXSIZE)) {
           case -1 : printf("Sohn: Ich warte..\n");               /* 4.3BSD */
                     break;
           case  0 : printf("Sohn: Ich warte..\n");               /* System V */
                     break;
      case FIXSIZE : printf("Sohn: Nachricht war: %s\n", buf);
                     exit(0);
           default : fatal("Fehler read");
     }
     sleep(1);
    }
   }

   close(fd[0]);
   printf("Hier ist der Vaterprozess, meine pid ist: %d\n", getpid());
   sleep(3);
   /* Schicken einer Nachricht an Sohn */

   if (write(fd[1], "Hallo!", FIXSIZE) == -1)
    syserr("write");
}
```

Anzumerken ist noch, auf welche Weise der `read()`-Aufruf auf *non-blocking* geschaltet wurde. In dem ersten Aufruf von `fcntl()` werden die aktuellen Flaggen, die dem Filedeskriptor eigen sind, geholt. Diese werden mit dem `O_NDELAY` - Flag bitweise ODER-verknüpft und wieder neu gesetzt. Dies war die Ausgabe des Programmes:

```
Hier ist der Vaterprozess, meine pid ist: 5519
Hier ist der Sohn-Prozess, meine pid ist: 5520
Sohn: Ich warte..
Sohn: Ich warte..
Sohn: Ich warte..
Sohn: Nachricht war: Hallo!
```

4. FIFOs

FIFO ist die Abkürzung für *first-in-first-out*. Dabei wird das zuerst geschriebene Byte auch wieder als erstes ausgelesen. Eine Pipe arbeitet nach diesem Prinzip. Ein FIFO ist eine Pipe mit Namen. Der FIFO besitzt einen Katalogeintrag wie gewöhnliche Dateien. Folglich können auch nicht verwandte Prozesse über einen FIFO miteinander kommunizieren wenn sie dessen Namen kennen. Ist ein FIFO geöffnet, verhält er sich genau wie eine Pipe. Seine Kapazität ist ebenfalls identisch mit der einer Pipe.

Im Unterschied zu einer Pipe, muß ein FIFO nach Erzeugung erst geöffnet werden. Dies geschieht mittels `open()`. Wird ein FIFO zum Lesen geöffnet, so wartet `open()` bis er vom anderen Prozeß auch zum Schreiben geöffnet wurde. Ähnlich blockiert ein Öffnen zum Schreiben solange, bis der Leseprozeß den FIFO ebenfalls zum Lesen öffnet. Hierin besteht die Synchronisation der beiden Prozesse. Das blockierende Warten kann durch Setzen des `O_NDELAY`-Flags abgeschaltet werden.

Das folgende Beispiel hat die gleiche Funktion wie das Beispiel `pipe.c` aus Kapitel 3. Nur wird hier anstelle einer Pipe ein FIFO verwendet. Ein FIFO wird durch die Systemfunktion `mknod()` erzeugt wobei als Parameter das Flag `S_IFIFO` angegeben werden muß. Mittlerweile gibt es hierfür allerdings schon einen eigenen Systemaufruf `mkfifo()`.

```
/* fifo.c      Schreiben in FIFO, Lesen aus FIFO */

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include "defs.h"

#define SATZ1 "Dies ist der erste Satz."
#define SATZ2 "Dies ist Datensatz 2."
#define fifo "testfifo"


int makefifo(path)                              /* Erzeugen eines FIFOs, eigene Version */
 char* path;
{
  return(mknod(path, S_IFIFO | 0666, 0));
}


void main()
{

   int fd;
   char buf[256];

   /* FIFO erzeugen */

   if ((makefifo(fifo) == -1) && errno != EEXIST)
    fatal("makefifo");
   if ((fd = open(fifo, O_RDWR)) == -1) {
    fatal("open fifo");
   }
```

```
   printf("Filedeskriptor fd ist: %d\n", fd);

   /* In den FIFO schreiben */

   if (write(fd, SATZ1, sizeof(SATZ1)) == -1) syserr("write");
   if (write(fd, SATZ2, sizeof(SATZ2)) == -1) syserr("write");

   /* Aus dem FIFO lesen */

   if (read(fd, buf, sizeof(SATZ1)) == -1) syserr("read");
   printf("Der erste Datensatz lautet: %s\n", buf);
   if (read(fd, buf, sizeof(SATZ2)) == -1) syserr("read");
   printf("Der zweite Datensatz lautet: %s\n", buf);

   close(fd);
}
```

Einen FIFO im Filesystem erkennt man an dem Dateiattribut 'p' welches den Pipe-Charakter der Datei symbolisiert. Der von `fifo.c` erzeugte FIFO stellt sich im Dateisystem so dar:

```
prw-r--r--  1 freinat         0 Apr 22 13:02 testfifo
```

Die Ausgabe von `fifo.c` gleicht der von `pipe.c`:

```
Filedeskriptor fd ist: 3
Der erste Datensatz lautet: Dies ist der erste Satz.
Der zweite Datensatz lautet: Dies ist Datensatz 2.
```

Dabei wurden wieder zwei Datensätze, hier in Form von Strings, in den FIFO geschrieben und anschließend ausgelesen.

Jetzt sollen zwei unabhängige Prozesse durch einen FIFO miteinander verbunden werden. Ein Prozeß spielt den Empfänger, der andere den Sender. Als FIFO dient eine Datei gleichen Namens, die von einem der beiden Prozesse angelegt wird. Die Nachricht, die übermittelt wird, beschränkt sich auf den String "Hallo Receiver!". Hier folgen die Listings der beiden Programme, zuerst der Empfänger.

```
/* fifo1r.c   Der Empfaenger*/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include "defs.h"


int makefifo(path)                              /* FIFO erzeugen */
 char* path;
{
  return(mkfifo(path, S_IFIFO | 0666, 0));
}


void main()
{
  char* fifo = "FIFO";                          /* Name der FIFO-Datei */
  int fd;
  char buf[256];

  setbuf(stdout, NULL);                         /* ungepufferte Ausgabe */

  if ((makefifo(fifo) == -1) && errno != EEXIST)
```

```
   fatal("makefifo");
  if (errno == EEXIST) printf("Receiver: Aha, FIFO existiert schon.\n");
  else printf("Receiver: FIFO erzeugt!\n");

  printf("Receiver: Ich warte auf den Sender...\n");

  if ((fd = open(fifo, O_RDONLY)) == -1) {
   fatal("open FIFO");
  }

  printf("Receiver: FIFO ist zum Lesen geoeffnet.\n");
  read(fd, buf, 255);
  printf("Receiver: Folgende Nachricht ist gekommen: %s\n", buf);

}
```

Und hier der Sender:

```
/* fifo1s.c   Der Sender*/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include "defs.h"


#define NACHRICHT "Hallo Receiver!"            /* Die Nachricht */

int makefifo(path)                             /* FIFO erzeugen */
 char* path;
{
  return(mkfifo(path, S_IFIFO | 0666, 0));
}


void main()
{
  char* fifo = "FIFO";                         /* Name der FIFO-Datei */
  char buf[256];
  int fd;


  /* FIFO erzeugen, falls nicht schon existent */

  if ((makefifo(fifo) == -1) && errno != EEXIST)
   fatal("makefifo");
  if (errno == EEXIST) printf("Sender: Aha, FIFO existiert schon.\n");
  else printf("Sender: FIFO erzeugt!\n");

  if ((fd = open(fifo, O_WRONLY)) == -1) {
   fatal("open pipe");
  }

  printf("Sender: FIFO ist zum Schreiben geoeffnet.\n");
  write(fd, NACHRICHT, sizeof(NACHRICHT));
  printf("Sender: Daten geschrieben\n");
}
```

Der Empfänger wird als erstes gestartet, am besten durch Eingabe von '`fifo1r &`'. Damit läuft dieser im Hintergrund und der Sender kann auf der gleichen Konsole gestartet werden. So sah der Ablauf aus:

```
> fifo1r &
Receiver: FIFO erzeugt!
Receiver: Ich warte auf den Sender...
> fifo1s
Sender: Aha, FIFO existiert schon.
Receiver: FIFO ist zum Lesen geoeffnet.
Sender: FIFO ist zum Schreiben geoeffnet.
```

```
Sender: Daten geschrieben
Receiver: Folgende Nachricht ist gekommen: Hallo Receiver!
```

Der Empfänger `fifo1r` hat den FIFO angelegt und wartet in der `open()`-Anweisung bis auch der Sender `fifo1s` den FIFO zum Schreiben öffnet. Ist dies geschehen, gibt der Empfänger die Meldung 'FIFO zum Lesen geöffnet' aus und wartet wiederum in der `read()`-Anweisung auf Daten. Unterdessen hat auch der Sender festgestellt, daß der FIFO schon zum Lesen geöffnet wurde und gibt seinerseits die Meldung 'FIFO zum Schreiben geöffnet' aus. Dann wird die Nachricht mittels `write()` in den FIFO geschrieben. Der Empfänger liest die Nachricht und gibt sie auf der Standardausgabe aus.

Durch das Blockieren von `open()` warten die Prozesse aufeinander. Aus diesem Grund kann die Reihenfolge der Programme auch vertauscht werden. Zuerst wird der Sender aufgerufen. Dieser wartet dann solange, bis auch der Empfänger den FIFO öffnet. Alles weitere geschieht wie oben beschrieben.

Mit der Anweisung `setbuf()` wird die Standardausgabe ungepuffert vorgenommen. Bei gepufferter Ausgabe, besonders bei kurzen Textstücken, kann es vorkommen, daß der Text erst zu einem späteren Zeitpunkt erscheint und zwar erst, wenn der Puffer voll genug ist.

5. Messages

Message-Queues, auch Nachrichtenwarteschlangen genannt, sind vom Systemkern unterstützte Datenkanäle zur Übertragung von Nachrichten (Messages) zwischen verschiedenen Prozessen. Ein solcher Datenkanal arbeitet ebenfalls nach dem FIFO-Prinzip. Eine zuerst in den Kanal geschickte Nachricht wird auch als erste Nachricht ausgelesen. Dies gilt allerdings nur für Nachrichten desselben Typs. Der Typ einer Nachricht ist eine vom Benutzer angegebene positive Zahl (long integer), die jeder Nachricht vorangestellt wird. Im Gegensatz zu Pipes oder FIFOs können aber auch Nachrichten unterschiedlichen Typs in den Kanal geschickt werden. Es liegt dann an dem Leseprozeß, ob er alle Nachrichten der Reihenfolge nacheinander liest oder ob er bestimmte Nachrichten, die sich von den anderen in ihrem Typ unterscheiden, vorzieht. Das FIFO-Prinzip gilt nach wie vor, aber eben bezogen auf Nachrichten desselben Typs.

Zum Kennenlernen von Messages dient ein einfaches Beispiel `msg.c`, bei dem eine Nachricht in den Kanal geschickt wird und anschließend vom selben Prozeß wieder ausgelesen wird. Wie auch in den vorangegangenen Kapiteln beschränkt sich die Nachricht wieder auf einen String. Selbstverständlich ist jede andere Art von Daten erlaubt. Wichtig ist nur, daß jede Nachricht mit einem long integer beginnt, die den oben besprochenen Nachrichtentyp repräsentiert. Aus diesem Grund ist eine Nachricht als Record (struct) definiert. Das erste Element ist der Nachrichtentyp. Was dann folgt, ist dem Programmierer überlassen. In unserem Fall sind die Daten, die übermittelt werden sollen, eine Zeichenkette mit einer maximalen Länge von 256 Byte.

Anzumerken ist noch, daß zur Erzeugung einer Nachrichtenwarteschlange und für den Zugriff auf diese einmalig ein einzigartiger Schlüssel erforderlich ist. Dieser Schlüssel übernimmt die Aufgabe eines Namens für eine Warteschlange. Anhand dieses Schlüssels liefert der Kern die message-queue-id, kurz *msqid*, über die der weitere Zugriff erfolgt. Der Bezeichner *msqid* ist vergleichbar einem Filedeskriptor *fd* für Dateien. 'Einmalig' bedeutet, daß man diesen Schlüssel nur einmal benötigt, um den *msqid* zu erhalten. Alle weiteren Zugriffe erfolgen dann über diesen Bezeichner. 'Einzigartig' bedeutet, daß dieser Schlüssel auf dem gesamten System nicht schon für eine andere Nachrichtenwarteschlange verwendet wurde. Das Bereitstellen eines solchen Schlüssels ist Aufgabe des Systems, es braucht aber dazu die Hilfe des

Anwenders. Die Systemfunktion, die diesen Schlüssel erzeugt und liefert, ist die Funktion `ftok()`. Ihre Syntax folgt hier:

```
key_t ftok(pathname, ch)                    /* Erzeugen eines einmaligen Schluessels */
 char *pathname;                            /* existierende Datei */
 char ch                                    /* beliebiger Buchstabe */

/* liefert Schluessel oder -1 bei Fehler */
```

Dabei ist *key_t* der Typ des Schlüssels, meist ein long-integer. *pathname* ist eine vom Anwender angegebene Datei, die bereits existieren muß. Der Inhalt der Datei oder deren Länge spielen keine Rolle. *ch* ist ein beliebiger Buchstabe. Das System benutzt den Inode-Wert der Datei *pathname*, der einzigartig ist, und den Character *ch* zur Generierung des Schlüssels. Aus diesem Grund ist der Inhalt der Datei unwesentlich, wesentlich ist die Inode der Datei im Dateisystem. Deshalb darf die Datei auch nicht gelöscht werden, wenn noch Prozesse den Schlüssel zur Erlangung des *msqid* benötigen werden. Auch ein Neuanlegen der Datei an alter Stelle ist untersagt, da der Datei mit großer Wahrscheinlichkeit ein neuer Inode-Wert zugewiesen wird. Der dann generierte Schlüssel wird sich von dem alten Schlüssel unterscheiden.

Diese Methode, über einen Schlüssel an einen Bezeichner zu gelangen, wird nicht nur für Messages benutzt, sondern auch für gemeinsam benutzte Speicherbereiche (shared memory) und Semaphore. Hier folgt das Listing von `msg.c`:

```
/* msg.c      Schreiben in Queue, Lesen aus Queue */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>
#include "defs.h"

struct msgform {                                    /* Eine Nachricht */
 long mtype;                                        /* Nachrichtentyp */
 char mtext[256];                                   /* Daten */
};

#define MSGSIZE sizeof(struct msgform)              /* Groesse der Nachricht */
#define NACHRICHT "Dies ist die Nachricht"          /* Inhalt */

void main()
{
   int msqid;
   key_t key;
   struct msgform msg;                              /* gibt Warning unter Linux */

   if ((key = ftok("KEY", 'A')) == -1) {            /* Falls Datei nicht exisiert */
    close(open("KEY", O_RDONLY | O_CREAT, 0666));   /* Datei anlegen */
    if ((key = ftok("KEY", 'A')) == -1)             /* Schluessel erzeugen */
     fatal("creating file");
   }


   printf("key = %ld\n", key);
   if ((msqid = msgget(key, 0666 | IPC_CREAT)) == -1)      /* Bezeichner besorgen */
```

```
    fatal("msgget");
   printf("Message-Queue hergestellt, msqid = %d\n", msqid);

   msg.mtype = 1;                                          /* Nachrichtentyp */
   strcpy(msg.mtext, NACHRICHT);                           /* Nachricht */

   if (msgsnd(msqid, &msg, MSGSIZE, 0) == -1)              /* Nachricht senden */
    fatal("msgsnd");

   if (msgrcv(msqid, &msg, MSGSIZE, 0, 0) == -1)           /* Nachricht empfangen */
    fatal("msgrcv");

   printf("Die Nachricht lautete: %s\n", msg.mtext);

   if (msgctl(msqid, IPC_RMID, NULL) == -1)                /* Message-Queue loeschen */
    syserr("msgctl");
}
```

Als Schlüsseldatei, die zur Generierung eines einzigartigen Schlüssels dient, wird eine Datei namens "KEY" im aktuellen Verzeichnis angelegt. Die Wahl des Namens ist dabei willkürlich, ebenso die Wahl des Buchstabens 'A' für die Erzeugung des Schlüssels in `ftok()`. Der Schlüssel wird zur Information ausgegeben. Mit `msgget()` wird nun eine Message-Queue erzeugt und der Bezeichner zurückgegeben. In unserem Beispiel hatte *msqid* den Wert 150. Dies wird ebenfalls ausgegeben. Die zu übermittelnde Nachricht wird mittels `strcpy()` in den Datenteil des Nachrichten-Records kopiert. Der Nachrichtentyp ist willkürlich zu 1 gewählt. Die Nachricht wird nun durch `msgsnd()` in die Nachrichtenwarteschlange geschickt und anschließend mit dem Pendant `msgrcv()` wieder aus der Queue gelesen. Nach der Ausgabe wird die Nachrichtenwarteschlange gelöscht. Dies erfolgt durch den Aufruf von `msgctl()`. Hier ist die Ausgabe von `msg.c`:

```
key = 1091031101
Message-Queue hergestellt, msqid = 150
Die Nachricht lautete: Dies ist die Nachricht
```

Das Löschen der Nachrichtenwarteschlange sollte immer erfolgen, wenn diese nicht mehr benötigt wird. Eine Nachrichtenwarteschlange besteht solange, bis sie entweder gelöscht oder das System neu gebootet wird. Bei nachlässigem Experimentieren mit Messages Queues sammeln sich Message-Queues an, die nie wieder benutzt werden, aber immer noch Platz im Systemkern belegen. Auch Nachrichten in den Warteschlangen werden solange behalten, bis sie endlich gelesen werden oder die Warteschlange gelöscht wird. Mit dem im Anhang A beschriebenen Programm `msgstat` kann das System nach vergessenen Message-Queues abgesucht werden.

Auf der nächsten Seite folgt eine Beschreibung der System-Aufrufe für Messages:

```
int msgget(key, flags)                          /* Bezeichner der Message-Queue holen */
    key_t key;                                  /* Schluessel der Queue */
    int flags;                                  /* Flaggen */

/* liefert Bezeichner (msqid) oder -1 bei Fehler */


int msgsnd(msqid, buf, nbytes, flags)           /* Nachricht in Warteschlange schicken */
    int msqid;                                  /* Bezeichner der Queue */
    struct msgbuf *buf;                         /* Zeiger auf Nachricht (struct) */
    int nbytes;                                 /* Groesse der Nachricht */
    int flags                                   /* Flaggen */

/* liefert 0 bei Erfolg oder -1 bei Fehler */


int msgrcv(msqid, buf, nbytes, mtype, flags)    /* Nachricht empfangen */
    int msqid;                                  /* Bezeichner der Queue */
    struct msgbuf *buf;                         /* Zeiger auf Nachrichtenpuffer */
    int nbytes;                                 /* Anzahl zu lesender Byte */
    int mtype;                                  /* Typ der gewünschten Nachricht */
    int flags;                                  /* Flaggen */


/* liefert Anzahl gelesener Byte oder -1 bei Fehler */


int msgctl(msqid, cmd, sbuf)                    /* Queue manipulieren */
    int msqid;                                  /* Bezeichner der Queue */
    int cmd;                                    /* Manipulationsbefehl */
    struct msqid_ds *sbuf                       /* Zeiger auf Statuspuffer */

/* liefert 0 bei Erfolg und -1 bei Fehler */
```

`msgget()` Erhalten eines Bezeichners für Zugriff auf eine Message-Queue. *key* ist ein einzigartiger Schlüssel, der vorher mittels `ftok()` ermittelt worden sein sollte. Hat *flags* den Wert `IPC_CREAT`, so wird für den Schlüssel *key* eine neue Message-Queue erzeugt, falls diese nicht schon existiert. Werden im Wort *flags* die Bits `IPC_CREAT` und `IPC_EXCL` gesetzt, so wird ein Fehler gemeldet, wenn zum übergebenen Schlüssel bereits eine Queue besteht.

`msgsnd()` Senden einer Nachricht in die Message-Queue. *msqid* ist der Bezeichner der gewünschten Queue. *buf* ist ein Zeiger auf eine beliebige Nachrichtenstruktur, die aber immer als erstes Element einen long-integer, den Message-Typ, enthalten muß. *nbytes* ist die Größe der Nachricht, wobei der Nachrichtentyp (long int) nicht mitgezählt wird. Das Argument *flags* kann entweder den Wert `IPC_NOWAIT` oder 0 annehmen. Bei `IPC_NOWAIT` kehrt `msgsnd()` sofort zurück, wenn die Nachricht wegen Überfüllung der Queue nicht mehr hinein paßt. Dann ist `errno` mit dem Wert `EAGAIN` gesetzt. Ansonsten wartet `msgsnd()` bis Platz in der Warteschlange frei geworden ist.

`msgrcv()` Empfangen einer Nachricht aus der Nachrichtenwarteschlange. *msqid*, *buf*, *nbytes* haben die gleiche Funktion wie bei `msgsnd()`. Wenn das Argument *flags* auf `MSG_NOERROR` gesetzt ist, so wird kein Fehler gemeldet, wenn die aktuelle Nachricht die Länge *nbytes* überschreitet. Die Nachricht wird einfach an der Stelle abgeschnitten.

Durch *mtype* wird festgelegt, welcher Nachrichtentyp aus der Schlange empfangen werden soll. Ist *mtype* gleich 0, wird nach dem FIFO-Prinzip die älteste Nachricht gelesen, egal welchen Typs diese ist. Ist *mtype* auf einen Wert größer 0 gesetzt, so wird die älteste Nachricht gelesen, deren Typ identisch dem Wert von *mtype* ist. Wenn *mtype* einen Wert kleiner 0 besitzt, so liest `msgrcv()` die älteste Nachricht, deren Typ gleich oder kleiner dem Absolutwert von *mtype* ist.

`msgctl()` Befragen oder Manipulieren einer Nachrichtenwarteschlange. *msqid* ist der Bezeichner der Queue. Durch *cmd* werden die Befehle: `IPC_RMID`, `IPC_STAT` und `IPC_SET` übergeben. *sbuf* ist ein Zeiger auf die in `<sys/types.h>` definierte Struktur `msqid_ds`, die Zugriffsrechte, Zugriffszeiten und andere aktuelle Informationen über die Queue enthält.

Mit *cmd* = `IPC_RMID` wird die Queue gelöscht. Durch Setzen von *cmd* auf `IPC_STAT` wird der Status der Queue abgefragt, der in der Struktur **sbuf* zurückgegeben wird. Umgekehrt setzt *cmd* = `IPC_SET` einen neuen Status, der in der Struktur **sbuf* übergeben wird.

Im folgenden Beispiel wird die Nachrichtenübermittlung zwischen zwei unabhängigen Prozessen gezeigt. Wie im Beispiel `fifo1r.c/fifo1s.c` aus Kapitel 4 gibt es auch hier einen Sender und einen Empfänger. Hier folgen die Listings:

```
/* msg1r.c      Der Empfaenger*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>
#include "defs.h"

struct msgform {                                /* eine Nachricht */
 long mtype;                                    /* Nachrichtentyp */
 char mtext[256];                               /* Daten */
};

#define MSGSIZE sizeof(struct msgform)          /* Groesse der Nachricht */

void main()
{
```

```
   int msqid,
       msgflag;
   char buf[256];
   key_t key;
   struct msgform msg;

   if ((key = ftok("KEY", 'B')) == -1) {                    /* existiert Datei? */
    close(open("KEY", O_RDONLY | O_CREAT, 0666));           /* Datei erzeugen */
    if ((key = ftok("KEY", 'A')) == -1)                     /* Schluessel holen */
     fatal("creating file");
   }

   if ((msqid = msgget(key, 0666 | IPC_CREAT)) == -1)       /* Bezeichner holen */
    fatal("msgget");
   printf("Receiver: Message-Queue hergestellt, msqid = %d. Ich warte..\n", msqid);

   if (msgrcv(msqid, &msg, MSGSIZE, 0, 0) == -1)            /* Warten auf Nachricht */
    fatal("msgrcv");
   printf("Receiver: Die Nachricht war: %s\n", msg.mtext);

   if (msgctl(msqid, IPC_RMID, NULL) == -1)                 /* Queue loeschen */
    syserr("msgctl");
}
```

Und hier der Sender:

```
/* msg1s.c  Der Sender */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>
#include "defs.h"

struct msgform {                                /* eine Nachricht */
 long mtype;                                    /* Nachrichtentyp */
 char mtext[256];                               /* Daten */
};

#define MSGSIZE sizeof(struct msgform)          /* Groesse der Nachricht */

void main()
{
   int msqid,
       msgflag;
   char buf[256];
   key_t key;
   struct msgform msg;

   if ((key = ftok("KEY", 'B')) == -1) {                    /* existiert Datei? */
    close(open("KEY", O_RDONLY | O_CREAT, 0666));           /* Datei erzeugen */
    if ((key = ftok("KEY", 'A')) == -1)                     /* Schluessel holen */
     fatal("creating file");
   }

   if ((msqid = msgget(key, 0666 | IPC_CREAT)) == -1)       /* Bezeichner holen */
    fatal("msgget");

   printf("Sender: Message-Queue hergestellt, msqid = %d\n", msqid);
   printf("Sender: Bitte Nachricht eingeben: ");
   gets(buf);
   msg.mtype = 1;                                           /* Nachrichtentyp = 1 */
   strcpy(msg.mtext, buf);                                  /* Daten kopieren */

   if (msgsnd(msqid, &msg, MSGSIZE, 0) == -1)               /* Nachricht senden */
    fatal("msgsnd");
}
```

Durch Eingabe von '`msg1r &`' wird zuerst der Empfänger gestartet. Dieser erzeugt die Schlüsseldatei "KEY", falls sie noch nicht existiert. Dann wird mit `ftok()` der Schlüssel ge-

neriert. Zu diesem Schlüssel erhält man durch `msgget()` den Bezeichner der Nachrichtenwarteschlange. Falls die Queue noch nicht existiert, wird sie vom Kern angelegt. In dem Aufruf `msgrcv()` wartet der Empfänger, bis die Nachricht kommt. Es handelt sich hier, wie im Beispiel `fifo1r.c`, wieder um blockierendes Lesen. Nachdem die Nachricht gelesen wurde, wird diese ausgegeben und die Warteschlange gelöscht.
Der Sender arbeitet äquivalent. Im Unterschied zu `fifo1s.c` kann hier ein beliebiger Text als Nachricht eingegeben werden (Funktion `gets()`). Beide Prozesse benutzen die Schlüsseldatei "KEY", sowie den willkürlich gewählten Buchstaben 'B' zur Generierung des Schlüssels. Dies ist die Ausgabe des Vorganges:

```
> msg1r&
Receiver: Message-Queue hergestellt, msqid = 500. Ich warte..
> msg1s
Sender: Message-Queue hergestellt, msqid = 500
Sender: Bitte Nachricht eingeben: Hallo Empfaenger!
Receiver: Die Nachricht war: Hallo Empfaenger!
```

Auch hier ist die Reihenfolge der Aufrufe von Empfänger und Sender umkehrbar. Man kann erst den Sender starten, der die Nachricht in die Queue absetzt ohne dabei auf den Empfänger zu warten. Dies ist der wesentliche Unterschied zu dem in Kapitel 4 behandelten Beispiel mit FIFOs. Dort warteten beide Prozesse gegenseitig aufeinander. Hier aber endet der Sender nachdem er Nachricht abgesetzt hat. Der Empfänger kann zu einem beliebigen Zeitpunkt gestartet werden, um die Nachricht zu lesen. Dabei wird deutlich, warum die Message-Queue Nachrichten*warte*schlange genannt wird. Die Nachrichten verbleiben solange, bis sie entweder gelesen werden oder die Queue gelöscht wird.

6. Shared-Memory

Die schnellste Art, Daten von einem Prozeß zu einem anderen zu übertragen, besteht darin, sie überhaupt nicht zu übertragen[1]. Dies ist die Grundlage für gemeinsam benutzte Speicherbereiche, sogenanntes *Shared-Memory*. Anstatt Daten zu übertragen, benutzen Prozesse den gleichen Speicherbereich. Wenn ein Prozeß dort Daten ablegt, sind sie unmittelbar für alle anderen Prozesse verfügbar. Allerdings muß auch hier darauf geachtet werden, daß Empfängerprozesse nicht zu früh lesen und damit nur unvollständige Daten erhalten.

Die Bereitstellung eines gemeinsamen Speicherbereiches, auch als *Shared-Memory Segment* bezeichnet, erfolgt durch das Betriebssystem. Es kann mehrere gemeinsame Segmente geben, die von unterschiedlichen Prozessen benutzt werden. Ein Shared-Memory Segment wird durch ein System-Aufruf in den virtuellen Adressraum eines Prozesses abgebildet. Der Zugriff erfolgt mit den üblichen Speichermanipulationsbefehlen. Ein Segment wird bei verschiedenen Prozessen an unterschiedliche virtuelle Adressen abgebildet. Diese Adresse wird vom System bestimmt und hängt davon ab, wo der nächste freie Platz im virtuellen Adressraum eines Prozesses ist.

Die Systemaufrufe für die Benutzung von Shared-Memory sind:

```
int shmget(key, nbytes, flags)                /* Bezeichner für Segment holen */
    key_t key;                                /* Schlüssel des Segments */
    int nbytes                                /* Groesse des Segments */
    int flags                                 /* Flaggen */

/* liefert Segmentbezeichner (shmid) oder -1 bei Fehler */


char *shmat(shmid, addr, flags)               /* Segment in virtuellen Adressraum abbilden */
      int shmid;                              /* Bezeichner des Segments */
      char *addr;                             /* gewuenschte Adresse im virt. Addressraum */
      int flags;                              /* Flaggen */

/* liefert Segmentadresse oder -1 bei Fehler */


int shmdt(addr)                               /* Segment aus virt. Adressraum entfernen */
    char *addr;                               /* Segmentadresse im virt. Adressraum */

/* liefert 0 bei Erfolg oder -1 bei Fehler */
```

[1] Zitat aus: Marc Rochkind, UNIX Programmierung für Fortgeschrittene.

```
int shmctl(shmid, cmd, sbuf)                    /* Segment manipulieren */
    int shmid;                                  /* Bezeichner des Segments */
    int cmd;                                    /* Manipulationsbefehl */
    struct shmid_ds *sbuf;                      /* Zeiger auf Status-Puffer */

/* liefert 0 bei Erfolg oder -1 bei Fehler */
```

`shmget()` Erhalten eines Bezeichners für Zugriff auf ein Shared-Memory Segment. *key* ist ein einzigartiger Schlüssel, der vorher mittels `ftok()` generiert worden sein sollte. Wenn in *flags* das `IPC_CREAT` Bit gesetzt ist, wird das Segment gegebenenfalls erzeugt. Ist zusätzlich das Bit `IPC_EXCL` gesetzt, so liefert `shmget()` einen Fehler, wenn zum übergebenen Schlüssel schon ein Segment existiert.

`shmat()` Abbilden eines Segments in den eigenen Adressraum. *shmid* ist der Bezeichner des gewünschten Segments, *addr* die gewünschte Adresse. Ist diese auf `NULL` gesetzt, sucht das System ein passende Adresse aus. Ist *addr* nicht `NULL`, dann kann mit *flags* = `SHM_RND` gefordert werden, daß die übergebene Adresse *addr* vom System um den festgelegten Wert `SHMLBA` abgerundet wird.

Als Default kann der aufrufende Prozeß das Segment lesen und beschreiben, durch Setzen des *flags*-Bit `SHM_RDONLY` kann der Zugriff auf 'nur Lesen' beschränkt werden. Der Rückgabewert ist ein Zeiger auf die Anfangsadresse des Segments im eigenen Adressraum.

`shmdt()` Ausblenden eines Segments aus dem eigenen Adressraum. Benötigt der Prozeß das gemeinsame Speichersegment nicht mehr, löst er mit `shmdt()` die Verbindung. Der Inhalt des Segments bleibt erhalten, es kann jederzeit wieder mit `shmat()` eingeblendet werden. Dabei kann die neue Adresse jedoch eine andere sein.

`shmctl()` Befragen oder Manipulieren eines Segments. Durch *cmd* werden die Befehle `IPC_RMID`, `IPC_STAT` und `IPC_SET` übergeben. Mit `IPC_RMID` wird das Shared-Memory Segment gelöscht. Mittels `IPC_STAT` kann der Status erfragt werden, der über *sbuf* geliefert wird. Mit `IPC_SET` können die Zugriffsrechte verändert werden.

Als einführendes Beispiel wird in `shm.c` eine Nachricht in Form eines Strings in ein Shared-Memory Segment geschrieben. Wie auch in den vorangegangenen Kapiteln beschränkt sich die Art der Nachricht nicht nur auf Zeichenketten, diese Form wurde wieder nur der Anschaulichkeit halber gewählt.

```
/* shm.c      Schreiben einer Nachricht in ein Shared-Memory Segment */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include "defs.h"


#define SHMSIZE 128*K

void main()
{
  key_t key;
  int shmid;
  char *addr;

  if ((key = ftok("KEY", 'C')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'C')) == -1)
    fatal("creating file");
  }

  if ((shmid = shmget(key, SHMSIZE, 0777 | IPC_CREAT)) == -1)      /* Bezeichner holen */
   fatal("shmget");
  printf("Segment bereitgestellt, shmid = %d\n", shmid);

  if ((addr = shmat(shmid, NULL, 0)) == (char*)-1)                 /* Segment einblenden */
   fatal("shmat");

  printf("Segment eingeblendet, addr = 0x%x\n", addr);
  printf("Bitte kurzen Text eingeben: ");
  gets(addr);                                          /* Nachricht im Segment speichern */

  printf("Text wurde ins Segment geschrieben\n");
  printf("Segment wird jetzt ausgeblendet..");

  if (shmdt(addr) == -1)                                           /* Segment ausblenden */
   syserr("shmdt");
  printf("..und wieder eingeblendet\n");

  if ((addr = shmat(shmid, NULL, 0)) == (char*)-1)                 /* Segment einblenden */
   fatal("shmat");

  printf("addr = 0x%x\n", addr);
  printf("Der gelesene Text lautet: %s\n", addr);
  if (shmctl(shmid, IPC_RMID, 0) == -1)                            /* Segment loeschen */
   fatal("shmctl");
}
```

Die Methode der Benutzung eines Schlüssels zur Identifizierung eines Shared-Memory Segments wird auf die gleiche Weise wie bei der Verwendung von Messages angewandt. Dabei weist die Funktion `ftok()` der Schlüsselvariablen `key` einen Schlüssel zu. Dessen Berechnung erfordert die Datei "KEY", die nötigenfalls angelegt wird. Mit Hilfe des Schlüssels erhält man durch `shmget()` den Bezeichner `shmid` des Shared Memory Segments. Falls zu dem übergebenen Schlüssel noch kein Segment existiert, wird dieses angelegt. Die Größe des gewünschten Segments ist in diesem Beispiel durch die Konstante `SHMSIZE` festgelegt worden und beträgt hier 128 KByte.

Der von `shmget()` gelieferte Bezeichner wird zur Information auf der Standardausgabe ausgegeben, in diesem Fall beträgt dessen Wert 400. Das Shared-Memory Segment existiert nun

zwar in dem System und ist über den Bezeichner `shmid` auch anwählbar, jedoch können noch keine Daten gelesen oder geschrieben werden. Dazu muß des Segment erst in den eigenen Adressraum abgebildet werden. Dies erfolgt durch `shmat()`, wobei wir das System eine geeignete Adresse aussuchen lassen. Der Rückgabewert von `shmat()` ist ein Zeiger auf den Anfang des Segments. Über diesen Zeiger kann man nun direkt in das Segment schreiben. Nachdem die Adresse in hexadezimaler Darstellung ausgegeben ist, kann ein kurzer Text eingegeben werden. Dieser wird direkt im Shared-Memory Segment gespeichert.
Zur weiteren Demonstration wird das Segment nun aus dem Adressraum ausgeblendet. Danach würde ein Ansprechen des Speicherbereiches über `addr` einen Fehler verursachen. Das Ausblenden erfolgt durch `shmdt()`. Um zu zeigen, daß sich der eingelesene String noch immer in dem Shared-Memory Segment befindet, wird dieses wieder mittels `shmat()` eingeblendet. In unserem Beispiel bestimmt das System wieder die gleiche Adresse. Diese wird nocheinmal zur Information ausgegeben, dann folgt die Ausgabe des Strings. Zuletzt wird mittels `shmctl()` und dem Parameter `IPC_RMID` das Segment gelöscht. Hier folgt die Ausgabe von `shm.c` :

```
Segment bereitgestellt, shmid = 400
Segment eingeblendet, addr = 0xf76f0000
Bitte kurzen Text eingeben: Dies ist ein Test.
Text wurde ins Segment geschrieben
Segment wird jetzt ausgeblendet....und wieder eingeblendet
addr = 0xf76f0000
Der gelesene Text lautet: Dies ist ein Test.
```

Wie auch bei Messages sollte immer darauf geachtet werden, daß nicht mehr benötigte Shared-Memory Segmente gelöscht werden. Sie belegen Platz im System bis die Maschine neu gebootet wird. Zum Auffinden vergessener Shared-Memory Segmente dient das im Anhang A vorgestellte Programm `shmstat`.
Im nächsten Beispiel wird das klassische Sender/Empfänger-Konzept anhand von Shared-Memory als Übertragungsmedium gezeigt. Dabei fungiert `shm1r.c` als Empfänger und `shm1s.c` als Sender.

```
/* shm1r.c      Der Empfaenger */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <malloc.h>
#include "defs.h"
```

```
void main()
{
  key_t key;
  int shmid;
  char *addr1;
  extern char* shmat();

  setbuf(stdout, NULL);

  if ((key = ftok("KEY", 'D')) == -1)
   fatal("Datei KEY muss existieren!");

  if ((shmid = shmget(key, 1*K, 0777 | IPC_CREAT)) == -1)
   fatal("shmget");
  if ((addr1 = shmat(shmid, NULL, 0)) == (char*)-1)
   fatal("shmat");

  printf("Receiver: Warte auf Nachricht..\n");

  while (*addr1 == (char)0);                        /* Warten, bis Flag gesetzt ist */

  printf("Receiver: Nachricht empfangen: %s\n", addr1+1);
  if (shmctl(shmid, IPC_RMID, 0) == -1)
   fatal("shmctl");
}
```

Hier folgt der Sender:

```
/* shm1s.c Der Sender */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <malloc.h>
#include "defs.h"


void main()
{
  key_t key;
  int shmid;
  char *addr1;
  extern char* shmat();


  if ((key = ftok("KEY", 'D')) == -1)
   fatal("Datei KEY muss existieren!");

  if ((shmid = shmget(key, 1*K, 0777 | IPC_CREAT)) == -1)
   fatal("shmget");
  printf("Sender: Segment bereitgestellt, shmid = %d\n", shmid);
  if ((addr1 = shmat(shmid, NULL, 0)) == (char*)-1)
   fatal("shmat");

/* Flag setzen, dass noch nicht gelesen werden darf! */

  *addr1 = (char)0;

  printf("Sender: Bitte kurzen Text eingeben: ");
  gets(addr1+1);
  *addr1 = (char)1;                              /* Jetzt darf gelesen werden */
}
```

Wir beginnen bei der Beschreibung des Senders. Der Schlüssel wird auf die gleiche Weise wie im Beispiel `shm.c` erzeugt. Auf ein Anlegen der notwendigen Schlüsseldatei "KEY"

wurde hier verzichtet, da diese schon durch Ausprobieren von `shm.c` existieren sollte[2]. Das Shared-Memory Segment hat eine Größe von 1KByte, über `shmget()` erhält man den Bezeichner `shmid`. Mit `shmat()` wird das Segment dann in den Adressraum abgebildet. Bevor jedoch ein String eingegeben werden kann, wird das erste Byte im Shared-Memory Segment auf den Wert 0 gesetzt. Es fungiert als Flag und signalisiert dem schon wartenden Empfänger daß die Daten noch nicht vorliegen und er deshalb nicht schon lesen darf. Jetzt kann eine Zeichenkette eingegeben werden, die hinter dem Flag im Shared-Memory Segment abgelegt wird. Das Argument von `gets()` zeigt deshalb auf das zweite Byte im Segment. Zuletzt wird das Flag auf den Wert 1 gesetzt. Damit hat der Sender seine Arbeit erledigt.
Der Empfänger erzeugt ebenfalls einen Schlüssel, der wegen des Schlüsseldatei "KEY" und dem Verschlüsselungsbuchstaben 'C' genau mit dem des Senders übereinstimmt. Zu diesem Schlüssel erhält man durch `shmat()` einen Bezeichner für das Shared-Memory Segment. Die Größe des Segments ist ebenfalls 1KByte. Dann wird das Segment in den eigenen Adressraum abgebildet, um über den von shmat() gelieferten Zeiger darauf zugreifen zu können. Jetzt fragt der Empfänger kontinuierlich das vereinbarte Flag ab und wartet bis dieses den Wert 1 angenommen hat. Dies ist eine sehr schlechte Methode weil es sich um sogenanntes *busy-wait* handelt, d.h. der Prozessor vergeudet beim Warten wertvolle Rechenzeit. Ist das Flag endlich vom Sender auf den Wert 1 gesetzt worden, kann der Empfänger die Daten lesen. Er gibt die Zeichenkette auf der Standardausgabe aus und löscht anschließend das Shared-Memory Segment.
Der Empfänger wird als erstes gestartet, wie üblich mit einem Ampersand '&' in der Befehlszeile. Dann wird der Sender aufgerufen. So sah der Vorgang aus:

```
> shm1r &
Receiver: Warte auf Nachricht..
> shm1s
Sender: Segment bereitgestellt, shmid = 301
Sender: Bitte kurzen Text eingeben: Hallo Empfaenger!
Receiver: Nachricht empfangen: Hallo Empfaenger!
```

Die Aufrufreihenfolge ist auch hier umkehrbar. Als erstes kann der Sender gestartet werden, der das Segment erzeugt und seine Daten dort ablegt. Der Empfänger kann zu einem späteren Zeitpunkt aufgerufen werden, er liest die Daten und löscht das Segment.

---

[2] Oder aus den Beispielen der vorangegangen Kapitel.

Besonders wichtig bei der Verwendung von Shared-Memory durch mehrere Prozesse ist das Management über den Zugriff. Ohne die Verwendung des simplen Flags als Synchronisationsmittel wäre eine korrekte Übermittlung nicht möglich gewesen. Gerade Shared-Memory, das auch als Systemresource betrachtet werden kann, erfordert Synchronisationsmittel, die nicht auf *busy-wait* basieren und damit Rechenzeit vergeuden. Ein solches Mittel, die Semaphore, wird im kommenden Kapitel behandelt.

Nun soll noch gezeigt werden, wie man ein Shared-Memory Segment an eine feste Adresse einblenden kann. Dazu wird `shmat()` als zweiter Parameter nicht `NULL`, sondern die gewünschte Adresse übergeben. Hier ist allerdings Vorsicht geboten: liegt die Adresse zu tief, besteht die Gefahr, daß Speicherallozierungsaufrufe wie `malloc()` Zeiger zurückliefern, die in das eingeblendete Segment zeigen. Ein Zugriff über diese Zeiger zerstört dann den Inhalt des Segments. Das liegt daran, daß die Allokationsfunktionen ein eingeblendetes Segment nicht erkennen können. Aus diesem Grund liegen die vom System ausgesuchten Adressen auch am Ende des virtuellen Adressraums, der auf unserer Maschine 4 immerhin GByte beträgt. Es ist unwahrscheinlich, daß ein Prozeß 4 GByte Speicher alloziert und damit die eingeblendeten Segmente wieder überschreiben würde.

```
/* shm2.c      Festlegen eines Shared-Memory Segments an feste Adresse */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include "defs.h"

#define SHMSIZE 1*K

void main()
{
  key_t key;
  int shmid;
  char *shmaddr;
  char *shmaddr1;
  extern char* shmat();

  if ((key = ftok("KEY", 'C')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'C')) == -1)
    fatal("creating file");
  }

  if ((shmid = shmget(key, SHMSIZE, 0777 | IPC_CREAT)) == -1)
   fatal("shmget");
  printf("Segment bereitgestellt, shmid = %d\n", shmid);

  shmaddr1 = (char*) 0x50000000;

  if ((shmaddr = shmat(shmid, shmaddr1, 0)) == (char*)-1)
   fatal("shmat");
  printf("Segment attached, address = 0x%x\t (dez=%ld)\n", shmaddr, shmaddr);

  /* testen */
```

```
  strcpy(shmaddr1, "Shared-Memory ist schreib- lesebereit");
  printf("%s\n", shmaddr1);
  if (shmctl(shmid, IPC_RMID, 0) == -1)
   fatal("shmctl");
}
```

In diesem Beispiel wird die Adresse für das Shared-Memory Segment explizit auf die Adresse 0x5000000 gelegt (= 1,25 GByte), was vom System auch anstandslos akzeptiert wird.
Zum Abschluß muß noch gesagt werden, daß sich Shared-Memory nur für statische Datenstrukturen eignet, weil ein Segment in unterschiedlichen Prozessen auch an unterschiedlichen Adressen eingeblendet wird. Ist diese Adresse bekannt, kann über bekannte Offsets auf die verschiedenen Daten zugegriffen werden. Für dynamische Datenstrukturen, die über Zeiger miteinander verbunden sind, ist die Verwendung von Shared-Memory ungeeignet. In einem solchen Fall müssten die Zeiger nicht die absolute Adresse, sondern relative Adressen, gemessen vom Segmentanfang, beinhalten. Das erfordert eine eigene Speicherverwaltung. Im Übrigen ist die Größe von Shared-Memory Segmenten begrenzt, auf unserem System beträgt die maximale Größe 1 MByte.

7. Semaphore

Semaphore sind vom System unterstützte spezielle Flaggen, die zur Prozeßsynchronisation und zum geregelten Zugriff auf Systemresourcen dienen. Sie werden nicht zum Austausch von Daten benutzt, sondern helfen diesen Austausch zu regeln. Die Funktion einer solchen Flagge ist denkbar einfach: Ist sie gesetzt, darf kein anderer Prozeß auf ein bestimmtes Betriebsmittel zugreifen weil das Betriebsmittel schon benutzt wird. Ist sie hingegen frei, kann ein Prozeß die Flagge setzen und auf das Betriebsmittel zugreifen. In diesem Fall hat die Flagge binären Charakter: *Gesetzt* bedeutet für die Prozesse, daß sie warten müssen, *nicht gesetzt* bedeutet, daß sich ein Prozeß der Flagge bemächtigen darf und dann auf das Betriebsmittel zugreifen kann. Die Flagge hat also zwei Zustände.

Ohne die Benutzung von Semaphore muß der Anwender seine eigenen Flaggen programmieren. In einem einfachen gedanklichen Beispiel sollen parallel laufende Prozesse auf die Standardausgabe schreiben. Damit die Ausgabe nicht zerstückelt und verwürfelt erscheint, dürfen die Prozesse nicht wild durcheinander auf die Standardausgabe schreiben, sondern geordnet nacheinander. Dazu muß eine Flagge vereinbart werden. Ein Prozeß, der schreiben möchte, wartet bis die Flagge frei ist. Dann setzt er die Flagge und schreibt. Anschließend muß er die Flagge wieder zurücksetzen, damit der nächste Prozeß genauso verfahren kann.

Das erste Problem besteht nun darin, eine gemeinsame Flagge, auf die alle Prozesse zugreifen können, festzulegen. Eine Lösung wäre die Verwendung einer Datei in einem vorab abgesprochenen Verzeichnis. Existiert die Datei, wird dies als gesetzte Flagge interpretiert, daß heißt, die Standardausgabe wird schon von einem Prozeß benutzt. Existiert die Datei nicht, so muß der Prozeß, der die Standardausgabe benutzen möchte, die Datei anlegen und kann dann schreiben. Aber auch hier gibt es Probleme. Angenommen, ein Prozeß hat gerade festgestellt daß die Datei nicht existiert. Damit ist also die Standardausgabe unbenutzt. Er legt die Datei an und beginnt mit der Ausgabe. In der Zwischenzeit hat aber auch ein anderer Prozeß festgestellt, daß die Datei noch nicht existierte. Beide Prozesse legen die Datei an. Wenn sie beim Anlegen der Datei nicht erneut testen ob diese zu genau diesem Zeitpunkt existiert, sind beide in dem Glauben das alleinige Schreibrecht zu besitzen. Damit ist die Exklusivität aber nicht mehr gegeben.

Durch ausgeklügelte Programmierung lassen sich Dateien in der Tat als Flaggen verwenden. Das funktioniert aber nur wenn die Funktionen `open()` oder `creat()` atomar arbeiten, d.h.

nicht durch andere Prozesse unterbrochen werden. Abgesehen davon sind Dateioperationen im Vergleich zur Rechengeschwindigkeit langsam. Man stelle sich vor, daß ein Prozeß in unserem Beispiel eine Datei erzeugen und später wieder löschen muß, nur um in Ruhe ein paar Zeichen auf die Standardausgabe zu schreiben. Dazu kommt, daß sich wartende Prozesse in einem *busy-wait* befinden, also während des Wartens kostbare Rechenzeit verbrauchen. Die Forderungen an eine Flagge sind also:

- Die Flagge muß für alle Prozesse erreichbar sein. Das geht nur, wenn der Betriebssystemkern die Flaggen verwaltet und die Prozesse über Systemaufrufe darauf zugreifen können.

- Die Operationen 'Testen der Flagge' und ggf. 'Setzen der Flagge' müssen unmittelbar hintereinander ausführbar und atomar sein. Sie dürfen nicht von anderen Prozessen unterbrochen werden. Das kann nur der Betriebssystemkern veranlassen.

- Prozesse, die auf den Erwerb der Flagge warten, sollen nicht wertvolle Rechenzeit verbrauchen, sondern geweckt werden, wenn die Flagge verfügbar ist. Dies kann ebenfalls nur der Betriebssystemkern veranlassen.

- Die Flaggenoperationen sollten schnell ausführbar sein.

Diese Forderungen werden durch Semaphore erfüllt. Semaphore sind Flaggen, die vom Systemkern verwaltet werden. Der Zugriff erfolgt durch Systemfunktionen. Semaphore sind jedoch mehr als nur einfache Flaggen. Im Besonderen handelt es sich nicht nur um binäre Flaggen, sondern ein Semaphor kann eine große Anzahl von Werten annehmen. Damit kann z.B. festgelegt werden, wieviel Prozesse ein Betriebsmittel gleichzeitig benutzen, falls ein solches Betriebsmittel eine Mehrfachbenutzung zuläßt. Das Wort Betriebsmittel ist hier im weiteren Sinn zu verstehen, es bedeutet hier nicht nur Hardware sondern es kann sich auch um Software handeln. Beispielsweise kann ein Semaphor regeln, wieviel Prozesse gleichzeitig mit einem anderen Prozeß kommunizieren. Im den meisten Fällen werden Semaphore jedoch nur binär benutzt, um Exklusivität zu erreichen. Dies wird auch in den kommenden Beispielen gemacht.
Das System stellt Semaphore nicht nur im Einzelstück, sondern als Satz zur Verfügung. Ein solcher Satz, hier auch als Menge bezeichnet, kann mehrere Semaphore beinhalten. Hier folgen die zur Benutzung von Semaphore notwendigen Systemaufrufe:

```
int semget(key, nsems, flags)                  /* Bezeichner fuer Semaphor-Satz holen */
    key_t key;                                 /* Schluessel */
    int nsems;                                 /* Anzahl Semaphore pro Satz */
    int flags                                  /* Optionen */

/* liefert Bezeichner des Semaphor-Satzes oder -1 bei Fehler */


int semop(semid, ops, nops)                    /* Operationen auf Semaphor-Satz durchfuehren
*/
    int semid;                                 /* Bezeichner des Semaphor-Satzes */
    struct sembuf (*ops[])                     /* Vektor mit Operationen */
    int nops;                                  /* Anzahl Operationen */

/* liefert 0 oder -1 bei Fehler */


int semctl(semid, snum, cmd, arg)              /* Manipulation des Semaphor-Satzes */
    int semid;                                 /* Bezeichner des Semaphor-Satzes */
    int snum;                                  /* Semaphor in diesem Satz */
    int cmd;                                   /* Kommando */
    union semun arg                            /* Variante zur Eingabe/Rückgabe */

/* liefert vom Kommando abhaengigen Wert oder -1 bei Fehler */
```

`semget()` Erhalten eines Bezeichners für Zugriff auf einen Semaphore-Satz. *key* ist ein einzigartiger Schlüssel, der vorher mittels `ftok()` generiert worden sein sollte. Wenn in *flags* das `IPC_CREAT` Bit gesetzt ist, wird der Semaphore-Satz gegebenenfalls erzeugt. Dabei bestimmt *nsems* die gewünschte Anzahl der Semaphoren des Satzes. Ist zusätzlich das Bit `IPC_EXCL` gesetzt, so liefert `semget()` einen Fehler, wenn zum übergebenen Schlüssel schon ein Semaphore-Satz existiert. Ist die Anzahl der Semaphore pro Menge erst einmal festgelegt, so läßt sich dieser Wert später nicht mehr ändern.

`semop()` Operationen auf Semaphore-Satz durchführen. *semid* ist der Bezeichner der gewünschten Semaphore-Menge. *ops* ist ein Zeiger auf einen Vektor von Operationsbefehlen wobei *nops* die Anzahl von Befehlen angibt. Jeder Befehl in diesem Vektor hat folgende Struktur:

```
struct sembuf {
  ushort sem_num;          /* Nummer des Semaphors */
  short  sem_op;           /* Operation */
  short  sem_flag;         /* Flaggen fuer Operation */
}
```

Dabei gibt *sem_num* das Semaphor des Satzes an, auf das die Operation *sem_op* ausgeführt werden soll. *sem_op* bezeichnet die Operation wie folgt:

1. Wenn *sem_op* positiv ist, so wird dieser Wert zum aktuellen Wert des Semaphors *sem_num* addiert.

2. Ist *sem_op* gleich 0, bedeutet dies: Warten bis Semaphor *sem_num* den Wert 0 annimmt. Der Prozeß schläft in dieser Zeit.

3. Wird *sem_op* als negativer Wert angegeben, müssen zwei Fälle unterschieden werden: Ist der absolute Wert von *sem_op* kleiner oder gleich dem Wert des Semaphors, so wird *sem_op* zum Wert des Semaphors addiert. Damit wird der Semaphor-Wert runtergezählt. Ist der absolute Wert von *sem_op* größer als der Wert des Semaphors, so bedeutet dies: Warten bis Semaphor *sem_num* einen Wert, der gleich oder größer dem Absolutwert von *sem_op* ist, angenommen hat. Der Prozeß schläft in dieser Zeit.

sem_flag ermöglicht Optionen für die Operation *sem_op*. Beispielsweise kann das Flag `IPC_NOWAIT` gesetzt werden. Damit wird das Betriebssystem angewiesen, nicht zu warten, wenn die gewünschte Operation nicht vollständig ausgeführt werden kann. *sem_flag* kann auch den Wert `SEM_UNDO` annehmen, dadurch wird zu jeder Operation intern eine Gegenoperation gespeichert. Bei einem Notausstieg des Prozesses kann das System die Semaphoren wieder in den Ursprungszustand zurückversetzen. Darauf wird noch näher eingegangen.

`semctl()` Befragen oder Manipulieren einer Semaphoren-Menge: *semid* ist der Bezeichner des Semaphoren-Satzes, *snum* gibt das gewünschte Semaphor in diesem Satz an. Die Befehle werden mittels *cmd* übergeben. *arg* ist eine Variante, die wie folgt definiert ist:

```
union semun {
  int val;
  struct semid_ds *buff;
  ushort *array
}
```

Abhängig von *cmd* werden die verschiedenen Elemente von *arg* benutzt.

Um einen Semaphore-Satz zu löschen, muß für *cmd* der Wert `IPC_RMID` benutzt werden. Die Werte `GETVAL` und `SETVAL` werden für *cmd* eingesetzt um ein Semaphor zu setzen oder dessen Wert zu erfragen. Im Falle von `SETVAL` wird der Wert in *arg.val* übergeben. Bei `GETVAL` wird der Wert des Semaphors als Funktionswert von `semctl()` zurückgegeben. Mit *cmd* = `IPC_STAT` wird der

Status der Semaphore-Menge abgefragt. Die Informationen werden dann in **arg.buff* zurückgeliefert.

Das folgende Beispiel `sem.c` zeigt, wie man einen Semaphor-Satz erzeugt und wieder aus dem System entfernt. Das Programm hat keine besondere Funktion weil der Semaphor-Satz noch nicht benutzt wird. Es soll nur die Verwendung von `semget()` und `semctl()` vorgestellt werden.

```
/* sem.c      Erzeugen und Loeschen eines Semaphor-Satzes */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"


#define NSEMS 1                                 /* Anzahl Semaphore pro Satz */

void main()
{
  key_t key;
  int semid;

  if ((key = ftok("KEY", 'D')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'D')) == -1)
    fatal("creating file");
  }

  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)
   fatal("semget");

  printf("Semaphor-Satz bereitgestellt, semid = %d\n", semid);

  if (semctl(semid, 0, IPC_RMID, 0) == -1)
   fatal("semctl");
}
```

Wie schon in den Kapiteln 5 und 6 vorgestellt, wird auch hier eine Schlüsseldatei zur Generierung eines eindeutigen Schlüssels benötigt. Die Datei hat den Namen "KEY" und wird nötigenfalls angelegt. Die Funktion `ftok()` liefert den Schlüssel, der dann in `semget()` zum Erwerb des Satz-Bezeichners *semid* benötigt wird. Mit `NSEMS` wird die Anzahl der gewünschten Semaphore des Satzes angegeben. In unserem Beispiel wollen wir nur ein einziges Semaphor erzeugen. Der Parameter *flags* ist hier auf `0777 | IPC_CREAT` gesetzt. Damit geben wir Zugriffsrechte für jedermann und bitten `semget()` den Semaphore-Satz anzulegen, falls dieser noch nicht existiert. `NSEMS` hat nur Sinn, wenn ein neuer Satz angelegt wird. Später kann die Anzahl Semaphore pro Satz nicht mehr geändert werden. `NSEMS` wird dann ignoriert. Mehrere Semaphore pro Satz sind dann interessant, wenn man verschiedene Semaphore benutzen will und doch nur einen Bezeichner *semid*.

Nach der Ausgabe des Bezeichners wird der Semaphor-Satz wieder aus dem System entfernt. Dies geschieht mit `semctl()` und dem *cmd*-Parameter `IPC_RMID`. *arg* muß in diesem Fall als *arg.val* = 0 übergeben werden. Dies stieß aber beim Kompilieren auf Probleme, der Parameter 0 anstelle von *arg* wurde jedoch nicht beanstandet.
Jetzt soll gezeigt werden, wie Semaphore auf bestimmte Initialisierungswerte gesetzt werden und wie Operationen auf Semaphore durchgeführt werden. Dazu erzeugen wir in `sem1.c` einen Semaphore-Satz mit 4 Semaphore. Hier folgt das Listing von `sem1.c` :

```
/* sem1.c      Semaphore initialisieren, Operationen ausfuehren */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"


#define NSEMS 4


void main()
{
  key_t key;
  int semid;
  struct sembuf op[2];
  union semun arg;

  if ((key = ftok("KEY", 'D')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'D')) == -1)
    fatal("creating file");
  }

  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)       /* Satz erzeugen */
   fatal("semget");
  printf("sem1: semid = %d\n", semid);

  /* Setzen des ersten Semaphors auf den Wert 7 */

  arg.val = 7;
  if (semctl(semid, 0, SETVAL, arg) == -1) syserr("semctl SETVAL");

  /* Setzen des vierten Semaphors auf den Wert 2 */

  arg.val = 2;
  if (semctl(semid, 3, SETVAL, arg) == -1) syserr("semctl SETVAL");


  op[0].sem_num = 1;                          /* Semaphor: Das zweite */
  op[0].sem_op = 1;                           /* Operation: Erhoehen um 1 */
  op[0].sem_flg = 0;                          /* Optionen: keine */
  op[1].sem_num = 2;                          /* Semaphor: Das dritte */
  op[1].sem_op = 2;                           /* Operation: Erhoehen um 2 */
  op[1].sem_flg = 0;                          /* Optionen : keine */

  if (semop(semid, op, 2) == -1) syserr("semop");
  }
```

Das Erzeugen eines Semaphore-Satzes geschieht wie in `sem.c`. Alle Semaphore sind dabei standardmäßig auf den Wert 0 gesetzt. Als erste Tat wollen wir das erste Semaphor des Satzes auf den Wert 7 setzen. Dazu benutzen wir `semctl()`. *semid* ist der Bezeichner der Sema-

phor-Menge, in unserem Fall war *semid* gleich 220. Der zweite Parameter von `semctl()`, *snum*, hat hier den Wert 0 und bezeichnet damit das erste Semaphor der Menge. Der dritte Parameter, *cmd*, beschreibt das Kommando, das ausgeführt werden soll. Hier ist *cmd* gleich `SETVAL`, weil ein Semaphor gesetzt werden soll. *arg.val* erhält den Wert 7, dies ist der Wert, auf den das Semaphor gesetzt wird. Bei fehlerfreier Ausführung liefert `semctl()` den Wert 0 zurück.

Das Setzen des vierten Semaphors in der Menge auf den Wert 2 geschieht in analoger Weise. Danach werden zwei Operationen auf die Semaphore 1 und 2 angewandt. Beide Semaphore haben zu diesem Zeitpunkt noch immer den Wert 0. Die Operationen werden in einer Art 'Auftragsarray' an `semop()` übergeben. Der Grund liegt darin, daß diese Operationen vom Kern atomar, also ungestört, durchgeführt werden. Aus diesem Grund dürfen nicht zwei nacheinanderfolgende Aufrufe benutzt werden, weil der Prozeß zwischen den beiden Aufrufen unterbrochen werden kann. Kann der Kern aus einem bestimmten Grund die Operationen nicht durchführen, kehrt `semop()` mit einer Fehlermeldung zurück. Er stellt die Semaphore aber wieder so her, wie sie vor dem Aufruf waren. Dies ist der Grundsatz bei Semaphor-Operationen: entweder werden sie ganz (und ungestört) durchgeführt, oder gar nicht.

Bevor also `semop()` aufgerufen wird, muß das Array entsprechend den Operationen initialisiert werden. Jede Operation wird durch die Struktur *sembuf* beschrieben. In unserem Fall wollen wir zwei Operationen ausführen, deshalb besitzt das Array `op` zwei Strukturen, nämlich `op[0]` und `op[1]`. Zuerst wird `op[0]` gesetzt. Die Strukturvariable *sem_num* gibt an, welches Semaphor von der Operation betroffen ist. In unserem Fall ist dies Semaphor 2, folglich wird *sem_num* auf den Wert 1 gesetzt. *sem_op* beschreibt die Operation. Hier wird *sem_op* auf den Wert 1 gesetzt, was bedeutet: Addiere 1 zum aktuellen Wert des Semaphors. Die Optionsvariable *sem_flg* enthält mögliche Optionen. Wir verzichten vorerst darauf. Damit ist die erste Operation gesetzt, sie erhöht einfach den Wert des zweiten Semaphors um den Betrag 1. In der zweiten Operation soll das dritte Semaphor der Menge um den Betrag 2 erhöht werden. Dies geschieht analog.

Nun wird `semop()` aufgerufen. *semid* bezeichnet wieder die Semaphor-Menge, die wir manipulieren wollen. Als zweiter Parameter *ops* wird das Array `op` übergeben, in dem die Operationen beschrieben sind. Der dritte Parameter *nops* gibt die Anzahl von Operationen an,

in diesem Fall sind dies 2 Operationen. Bei fehlerfreier Ausführung liefert semop() den Rückgabewert 0.

Da in `sem1.c` der Semaphor-Satz nicht gelöscht wurde, verbleibt er im System. Mit dem im Anhang A beschriebenen Statusprogramm `semstat` kann das System nach vorhandenen Semaphore-Sätzen abgesucht werden. Wir benutzen `semstat`, um das Ergebnis der Manipulation zu betrachten. Dies ist die Ausgabe:

```
semstat: Status report of Semaphor-Sets. All Sets with an id up to semid 5000
will be looked for.

                    Current Semaphor-Set ID : 220
             owner's user id (sem_perm.uid) : 550
            owner's group id (sem_perm.gid) : 500
          creator's user id (sem_perm.cuid) : 550
        creators's group id (sem_perm.cgid) : 500
               access modes (sem_perm.mode) : 0100777
        slot sequence number (sem_perm.seq) : 22
                  queue key (sem_perm.key) : 1141362749
                       access permissions : 0777
current # of Semaphores in set (sem_nsems) : 4
             time of last semop (sem_otime) : Mon May  2 15:16:31 1994
            time of last change (msg_ctime) : Mon May  2 15:16:31 1994

              Value of semaphor #0 (semval) : 7
   last process modified semaphor (sempid) : 4314
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
              Value of semaphor #1 (semval) : 1
   last process modified semaphor (sempid) : 4314
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
              Value of semaphor #2 (semval) : 2
   last process modified semaphor (sempid) : 4314
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
              Value of semaphor #3 (semval) : 2
   last process modified semaphor (sempid) : 4314
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
```

Nach der allgemeinen Information über den Satz folgen die Daten der vier Semaphore. Wie zu erwarten, ist Semaphor 0 auf den Wert 7 gesetzt. Der Prozeß, der dies getan hat, hatte die `pid` 4314. Offensichtlich hatte `sem1` also die `pid` 4314. Weiter ist zu entnehmen, daß keine Prozesse auf Erhöhung dieses Semaphors warten. Ebenso warten keine Prozesse darauf, daß dies Semaphor den Wert 0 annimmt.

Semaphor 1, das zweite in der Menge, wurde wie gewünscht um den Betrag 1 erhöht. Da es vorher den Wert 0 hatte beträgt sein Wert nun 1. Semaphor 2 wurde ebenfalls durch die Operation `op[1]` korrekt um den Betrag 2 erhöht. Das letzte Semaphor wurde mittels `semctl()` direkt auf den Wert 2 gesetzt.

Der Semaphor-Satz sollte nun gelöscht werden. Dies geschieht mit dem Programm `semctl` aus Anhang A.

Jetzt soll ein Semaphor endlich dazu benutzt werden, zwei Prozesse zu synchronisieren. Wir nehmen an, daß die beiden Prozesse auf ein fiktives Betriebsmittel zugreifen möchten. Aus Platzgründen werden die Prozesse in unserem Beispiel nicht wirklich auf dies Betriebsmittel zugreifen, sie setzen bzw. warten lediglich auf Freigabe des Semaphors und tun sonst nichts. Ein Prozeß soll das Semaphor setzen und dann einfach 10 Sekunden warten. Dann gibt er das Semaphor wieder frei. Während dieser Wartezeit soll ein anderer Prozeß gestartet werden, der auf die Freigabe wartet.

Es stellt sich nun die Frage, wie 'Setzen' und 'Freigeben' in Semaphore umgesetzt werden sollen. Bedeutet Semaphorwert 0 Freigabe oder Besetzt? Wenn der Wert 0 eine Besetzung des Betriebsmittels anzeigt, wie ist dann ein Semaphorwert von 7 zu interpretieren? In der Tat herrscht hier einige Verwirrung. Semaphore können in vielfältiger Weise benutzt werden, was sie leicht unüberschaubar macht.

Für das folgende Beispiel legen wir fest, daß der Semaphorwert 0 freien Zugriff auf ein Betriebsmittel signalisiert. Der Wert 1 zeigt an, daß das Betriebsmittel besetzt ist. Es handelt sich also um ein binäres Semaphor. Das Programm `sem_lock` blockiert das fiktive Betriebsmittel für 10 Sekunden. In der Zwischenzeit muß `sem_wait` gestartet werden, das auf die Freigabe wartet. Hier folgt das Listing `sem_lock.c`:

```
/* sem_lock.c     Blockieren eines fiktiven Betriebsmittels fuer 10 Sekunden */
/*                Semaphorwert: 0 = Betriebsmittel frei    1 = Betriebsmittel besetzt */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"


#define NSEMS 1                                 /* Anzahl Semaphore pro Satz */

void main()
{
  key_t key;
  int semid;
  struct sembuf op[1];

  if ((key = ftok("KEY", 'E')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'E')) == -1)
    fatal("creating file");
  }
  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)
   fatal("semget");
  printf("sem_lock: Semaphor-Satz bereitgestellt, semid = %d\n", semid);

  /* Semaphor Setzen */

  op[0].sem_num = 0;                            /* Semaphor: Das erste im Satz */
  op[0].sem_op = 1;                             /* Operation: Erhoehen um den Wert 1 */
  op[0].sem_flg = 0;                            /* Optionen: keine */
```

```
  if (semop(semid, op, 1) == -1)                /* Setzen */
   syserr("semop");
  sleep(10);                                    /* In dieser Zeit 'sem_wait' starten */

  /* Semaphor freigeben */

  op[0].sem_num = 0;                            /* Semaphor: Das erste im Satz */
  op[0].sem_op = -1;                            /* Operation: Verringern um den Wert 1 */
  op[0].sem_flg = 0;                            /* Optionen: keine */

  if (semop(semid, o, 1) == -1)                 /* Freigeben */
   syserr("semop");
  printf("sem_lock: Semaphor freigegeben.\n");
}
```

Durch den Aufruf von `semop()` wird `op[0]` ausgeführt: Das Semaphor wird um den Betrag 1 erhöht. Damit gilt das Betriebsmittel als besetzt. Nach 10 Sekunden Schlaf wird das Semaphor um den Betrag 1 verringert. Damit hat es wieder den Wert 0 und das Betriebsmittel gilt als frei. Das Löschen des Semaphor-Satzes soll `sem_wait` übernehmen.

```
/* sem_wait.c     Warten auf Freigabe eines fiktiven Betriebsmittels */
/*                Semaphorwert: 0 = Betriebsmittel frei    1 = Betriebsmittel besetzt */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"


#define NSEMS 1                                 /* Anzahl Semaphore pro Satz */

void main()
{
  key_t key;
  int semid;
  struct sembuf op[1];

  if ((key = ftok("KEY", 'E')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'E')) == -1)
    fatal("creating file");
  }

  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)
   fatal("semget");
  printf("sem_wait: Semaphor-Satz bereitgestellt, semid = %d\n", semid);

  /* Warten bis Semaphor freigegeben */

  op[0].sem_num = 0;                            /* Semaphor: Das erste im Satz */
  op[0].sem_op = 0;                             /* Operation: Warten bis Semaphor == 0 */
  op[0].sem_flg = 0;                            /* Optionen: keine */

  printf("sem_wait: Warte bis Semaphor auf 0 gesetzt wird..\n");
  if (semop(semid, op, 1) == -1)
   syserr("semop");
  else printf("sem_wait: Semaphor wurde auf 0 gesetzt!\n");

  semctl(semid, IPC_RMID, 0);                   /* Semaphor-Satz loeschen */
}
```

Die Operation `op[0]` weist den Betriebssystemkern an, diesen Prozeß erst zu wecken wenn das Semaphor den Wert 0 erreicht hat. In dieser Zeit wartet der Prozeß also auf die Freigabe des Betriebsmittels ohne Rechenzeit zu verbrauchen. Ist das Semaphor auf 0 gesetzt worden,

gibt es eine entsprechende Meldung. Dann wird der Semaphore-Satz gelöscht. Der Ablauf sah wie folgt aus:

```
> sem_lock &
sem_lock: Semaphor-Satz bereitgestellt, semid = 100
> sem_wait
sem_wait: Semaphor-Satz bereitgestellt, semid = 100
sem_wait: Warte bis Semaphor auf 0 gesetzt wird..
sem_lock: Semaphor freigegeben.
sem_wait: Semaphor wurde auf 0 gesetzt!
```

In dem nun folgenden Beispiel wollen wir die Bedeutung des Semaphors umdrehen. Jetzt soll der Semaphor-Wert 0 bedeuten, daß das Betriebsmittel besetzt ist. Ein Semaphor-Wert größer 0 bedeutet, daß das Betriebsmittel benutzbar ist. Wir wollen im folgenden Beispiel wieder nur ein binäres Semaphor implementieren, d.h. wir benutzen nur den Semaphorwert 1 als Freigabezeichen. Da die Semaphore bei Erzeugung mit dem Wert 0 initialisiert sind, müssen die beiden Beispielprogramme etwas anders aufgebaut werden als im vorangegangen Beispiel. Das Programm `sem_wt.c` erzeugt ein Semaphor. Da das Semaphor dann den Wert 0 besitzt, bedeutet dies eine Besetzung des fiktiven Betriebsmittels. `sem_wt.c` muß also warten, bis ein anderer Prozeß das Semaphor auf den Wert 1 setzt. Genau dies erledigt `sem_free.c`. Hier folgt das Listing von `sem_wt.c` :

```
/* sem_wt.c       Warten auf Freigabe eines Betriebsmittels */
/*                Semaphorwert: 0 = Betriebsmittel besetzt    1 = Betriebsmittel frei */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"

#define NSEMS 1


void main()
{
  key_t key;
  int semid;
  struct sembuf op[1];

  if ((key = ftok("KEY", 'F')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'F')) == -1)
    fatal("creating file");
  }

  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)
   fatal("semget");

  printf("sem_wait: Semaphor-Satz bereitgestellt, semid = %d\n", semid);
  /* Warten auf Freigabe */

  op[0].sem_num = 0;                          /* Semaphor: Das erste im Satz */
  op[0].sem_op = -1;                          /* Operation: Warten bis Semaphor == 1 */
  op[0].sem_flg = 0;                          /* Optionen: keine */

  printf("sem_wt: Warte bis Semaphor auf 1 gesetzt wird..\n");
```

```
  if (semop(semid, op, 1) == -1)              /* warten */
   syserr("semop");
  else printf("sem_wt: Semaphor wurde auf 1 gesetzt!\n");

  semctl(semid, 0, IPC_RMID, 0);              /* Semaphor-Satz loeschen */
}
```

Der Semaphor-Satz wird wie in den vorangegangenen Beispielen erzeugt. Die Strukturvariable *sem_op* in `op[0]` wird auf -1 gesetzt. Dadurch will der Prozeß warten, bis das Semaphor einen Wert gleich oder größer dem Absolutwert von *sem_op* angenommen hat, also bis das Semaphor einen Wert gleich oder größer 1 erhält. Wenn dies geschehen ist, wird eine entsprechende Meldung ausgegeben und der Semaphor-Satz gelöscht. Wie schon erwähnt, soll `sem_free.c` das Semaphor setzen:

```
/* sem_free.c     Freigeben des Betriebsmittels */
/*                Semaphorwert: 0 = Betriebsmittel besetzt    1 = Betriebsmittel frei */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "defs.h"

#define NSEMS 1

void main()
{
  key_t key;
  int semid;
  struct sembuf op[1];


  if ((key = ftok("KEY", 'F')) == -1) {
   close(open("KEY", O_RDONLY | O_CREAT, 0666));
   if ((key = ftok("KEY", 'F')) == -1)
    fatal("creating file");
  }

  if ((semid = semget(key, NSEMS, 0777 | IPC_CREAT)) == -1)
   fatal("semget");

  printf("sem_free: Semaphor-Satz bereitgestellt, semid = %d\n", semid);

  /* Semaphor setzen */

  op[0].sem_num = 0;                          /* Semaphor: Das erste im Satz */
  op[0].sem_op = 1;                           /* Operation: Semaphor setzen */
  op[0].sem_flg = 0;                          /* Optionen: keine */

  if (semop(semid, op, 1) == -1)              /* Setzen */
   syserr("semop");

  sleep(1);
  semctl(semid, 0, IPC_RMID, 0);              /* Semaphor-Satz loeschen */
}
```

Hier wird die Strukturvariable *sem_op* in `op[0]` auf den Wert 1 gesetzt. Damit wird der Wert des Semaphors von 0 auf 1 geändert. Zur Sicherheit wird nach einer Sekunde der Semaphor-Satz gelöscht. Dies ist nur dann wichtig, wenn `sem_free` irrtümlich gestartet wurde, ohne daß `sem_wt` aktiv ist. Wir erhielten folgende Ausgabe:

```
> sem_wt &
sem_wt: Semaphor-Satz bereitgestellt, semid = 370
sem_wt: Warte bis Semaphore auf 1 gesetzt wird..
> sem_free
sem_free: Semaphoren-Satz bereitgestellt, semid = 370
sem_wt: Semaphor wurde auf 1 gesetzt!
```

Das Interessante an diesem Beispiel ist die Tatsache, daß alle Werte größer 0 eine Freigabe bedeuten. Wenn in `sem_free.c` die Strukturvariable *sem_op* beispielsweise auf den Wert 5 anstatt 1 gesetzt werden würde, wäre der Ablauf der gleiche. Man kann diese Art der Benutzung von Semaphore auch so auffassen: Das Semaphor gibt an, wieviel Prozesse auf das Betriebsmittel gleichzeitig zugreifen dürfen. Jeder Prozeß, der auf das Betriebsmittel zugreift, verringert den Semaphor-Wert um 1. Hat das Semaphor den Wert 0 erreicht, bedeutet dies: Das Betriebsmittel ist voll belegt und kann zur Zeit keinen Prozeß mehr zulassen. Das Semaphor fungiert dabei als Zähler und erlaubt nur einer bestimmten Anzahl von Prozessen die gleichzeitige Benutzung.

In welcher Art Semaphore zu interpretieren und zu benutzen sind, hängt ganz von der Problemstellung ab. Kann das Betriebsmittel mehrere Prozesse, aber davon nur eine begrenzte Anzahl, zulassen, eignet sich die gerade gezeigte Methode. Wird hingegen ein einfaches binäres Semaphor benötigt, empfiehlt sich die Interpretation wie in `sem_lock/sem_wait` gezeigt.

Nun soll noch gezeigt werden, wozu das SEM_UNDO - Flag in der Struktur *sembuf* dienen kann. Folgendes Problem tritt in der Praxis auf: Ein Prozeß, der gerade ein Betriebsmittel besetzt hat und dies auch durch Setzen eines Semaphors kundtut, wird plötzlich abrupt abgebrochen. Dies kann beispielsweise durch das UNIX-Kommando kill -9 *pid* geschehen[1]. Dadurch ist es dem Prozeß nicht mehr möglich, Aufräumarbeiten, z.B. das Rücksetzen des Semaphors, zu erledigen. Andere Prozesse, die auf Freigabe des Semaphors warten, glauben das Betriebsmittel besetzt und warten deshalb ewig. Damit kann eine ganze Prozeßgruppe lahmgelegt werden. Nun wäre es schön, wenn sich der Systemkern jede Änderung eines Prozesses an einem Semaphor merken würde, so wie sich moderne Editoren getätigte Editierungen merken und diese mittels der Undo-Funktion rückgängig gemacht werden können. Genau dazu dient das `SEM_UNDO`-Flag. Alle Operationen, die mit dem Flag `SEM_UNDO` durchgeführt werden, merkt sich der Systemkern. Bei Verlassen des Prozesses

werden diese Operationen wieder rückgängig gemacht. Da zu jeder Operation das Flag gesetzt werden kann, aber nicht muß, ist der Prozeß in der Lage, Semaphoränderungen durchzuführen, die nicht wieder restauriert werden.
Zur Verdeutlichung wird das Beispiel `sem1.c` modifiziert. Die Strukturvariable *sem_flg* erhält nun den Wert `SEM_UNDO`:

```
op[0].sem_num = 1;                      /* Semaphore: Die zweite */
op[0].sem_op = 1;                       /* Operation: Erhoehen um 1 */
op[0].sem_flg = SEM_UNDO;               /* Optionen: UNDO-Flag */
op[1].sem_num = 2;                      /* Semaphore: Die zweite */
op[1].sem_op = 2;                       /* Operation: Erhoehen um 2 */
op[1].sem_flg = SEM_UNDO;               /* Optionen : UNDO-Flag */
```

Mit dieser Modifikation wird `sem1.c` noch einmal gestartet. Im Unterschied zum ersten Beispiel erhalten wir mit `semstat` folgenden Status des Semaphor-Satzes:

```
semstat: Status report of Semaphore-Sets. All Sets with an id up to semid
300 will be looked for.

                  Current Semaphor-Set ID : 280
            owner's user id (sem_perm.uid) : 550
           owner's group id (sem_perm.gid) : 500
         creator's user id (sem_perm.cuid) : 550
       creators's group id (sem_perm.cgid) : 500
              access modes (sem_perm.mode) : 0100777
       slot sequence number (sem_perm.seq) : 28
                  queue key (sem_perm.key) : 1141362749
                        access permissions : 0777
  current # of Semaphor in set (sem_nsems) : 4
            time of last semop (sem_otime) : Tue May  3 13:34:48 1994
           time of last change (msg_ctime) : Tue May  3 13:34:48 1994

             Value of semaphor #0 (semval) : 7
   last process modified semaphor (sempid) : 5543
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
             Value of semaphor #1 (semval) : 0
   last process modified semaphor (sempid) : 5543
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
             Value of semaphor #2 (semval) : 0
   last process modified semaphor (sempid) : 5543
 # of proc. waiting sem. increase (semcnt) : 0
   # of proc. wait semaphor zero (semzcnt) : 0
             Value of semaphor #3 (semval) : 2
   last process modified semaphor (sempid) : 5543
 # of proc. waiting sem. increase (semcnt) : 0
  # of proc. wait semaphor zero (semzcnt) : 0
```

Das erste Semaphor hat nach wie vor den Wert 7, weil es explizit gesetzt wurde. Dieses Setzen ist nicht rückgängig zu machen. Das gleiche gilt für das vierte Semaphor. Die beiden anderen Semaphore wurden jedoch durch die Operationen `op[0]` und `op[1]` mit gesetztem `SEM_UNDO` Flag bearbeitet. Bei Beendigung des Prozesses hat das Betriebssystem die Sema-

---

1 Es ist eine weit verbreitete Unart, jeden hängenden Prozeß sofort mit `kill -9` *`pid`* zu entsorgen. Zuerst sollte versucht werden, diesen mit *normalem* kill, also `kill` *`pid`* zu beenden.

phore wieder so hergestellt, wie sie vor der Bearbeitung waren. Da in unserem Fall der Semaphor-Satz vor dem Aufruf von `sem1` noch nicht existierte, haben Semaphor 1 und 2 wieder den Initialisierungswert 0 angenommen.

Das `SEM_UNDO`-Flag kann zwar verhindern, daß ein Semaphor bei Abbruch eines Prozesses besetzt bleibt, es kann aber nicht verhindern, daß ungenutzte Semaphor-Sätze im System zurückbleiben. Es muß also bei der Implementation darauf geachtet werden, daß ein Semaphor-Satz endgültig gelöscht wird, wenn er von keinem Prozeß mehr benötigt wird. Dies gilt auch für die anderen Mechanismen, insbesondere für Message-Queues und Shared-Memory.

Anhang A: Die Include-Datei `defs.h`

In dieser Datei sind zwei Routinen enthalten, die in allen Beispielen benötigt werden. Die Routinen `syserr()` und `fatal()` geben im Fehlerfall eine kurze Fehlermeldung aus, sowie den Wert von `errno` und die dazu gehörende Fehlerbeschreibung. Während `syserr()` danach zum Aufrufpunkt zurückkehrt, beendet `fatal()` das Programm.
Das Listing folgt hier:

```
/* DEFS.H   Include Datei fuer allgemein benoetigte Routinen */

#define K 1024                                  /* 1K = 1024 */


/* syserr() gibt Fehlermeldung aus und kehrt zurueck */

void syserr(msg)
 char* msg;
{
  extern int errno, sys_nerr;
  extern char* sys_errlist[];

  fprintf(stderr, "SYSERR: %s (%d", msg, errno);
  if (errno > 0 && errno < sys_nerr)
   fprintf(stderr, "; %s)\n", sys_errlist[errno]);
  else
   fprintf(stderr, ")\n");
 }



/* fatal() gibt Fehlermeldung aus und beendet das Programm */

void fatal(msg)
 char* msg;
{
  extern int errno, sys_nerr;
  extern char* sys_errlist[];

  fprintf(stderr, "FATAL: %s (%d", msg, errno);
  if (errno > 0 && errno < sys_nerr)
   fprintf(stderr, "; %s)\n", sys_errlist[errno]);
  else
   fprintf(stderr, ")\n");
  exit(0);
}
```

Anhang B: Hilfsprogramme

Beim Experimentieren mit Messages, Shared-Memory und Semaphore kann es passieren, daß diese im System verbleiben ohne je wieder genutzt zu werden. Um solche 'Leichen' aufzuspüren, dienen die Programme `msgstat`, `shmstat` und `semstat`, die im Quellcode auf den nachfolgenden Seiten abgedruckt sind.

`msgstat`	Aufspüren von Message-Queues
`shmstat`	Aufspüren von Shared-Memory Segmenten
`semstat`	Aufspüren von Semaphor-Sätzen

Standardmäßig suchen die Programme alle Bezeichner bis zum Wert 5000 ab. Ein anderer Grenzwert kann als Parameter angegeben werden. Mit den Programmen `msgctl`, `shmctl` und `semctl` können die jeweiligen Leichen entfernt werden.

`msgctl`	Information über Message-Queue oder Löschen einer Message-Queue.
`shmctl`	Information über Shared-Memory Segment oder Löschen eines Shared-Memory Segmentes.
`semctl`	Information über Semaphor-Satz oder Löschen eines Semaphor-Satzes.

Diese drei Programme entstanden aus gleichnamigen Beispielen des Sun-OS Manuals.

Messages-Queues: `msgstat.c, msgctl.c`

```
/* msgstat.c    Status aller Message-Queues. Simple Version S.F. Feb. 1994 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>
#include "defs.h"
#include <errno.h>

#define MAXQID 5000

extern int errno;


void Info(msqid, buf)
 int msqid;
 struct msqid_ds buf;

{
  printf("                    Current message queue : %d\n", msqid);
  printf("           owner's user id (msg_perm.uid) : %d\n", buf.msg_perm.uid);
  printf("          owner's group id (msg_perm.gid) : %d\n", buf.msg_perm.gid);
  printf("        creator's user id (msg_perm.cuid) : %d\n", buf.msg_perm.cuid);
  printf("      creators's group id (msg_perm.cgid) : %d\n", buf.msg_perm.cgid);
  printf("             access modes (msg_perm.mode) : %#o\n", buf.msg_perm.mode);
  printf("      slot sequence number (msg_perm.seq) : %d\n", buf.msg_perm.seq);
  printf("                  queue key (msg_perm.key) : %ld\n", buf.msg_perm.key);
  printf("                       access permissions : %#o\n", buf.msg_perm.mode & 0777);
  printf(" current # of bytes in queue (msg_cbytes) : %d\n", buf.msg_cbytes);
  printf("               size of queue (msg_qbytes) : %d\n", buf.msg_qbytes);
  printf("           pid of last msgsnd (msg_lspid) : %d\n", buf.msg_lspid);
  printf("           pid of last msgrvc (msg_lrpid) : %d\n", buf.msg_lrpid);
  printf("             last msgsnd time (msg_stime) : %s", buf.msg_stime ?
          ctime(&buf.msg_stime) : "Not set\n");
  printf("             last msgrcv time (msg_rtime) : %s", buf.msg_rtime ?
          ctime(&buf.msg_rtime) : "Not set\n");
  printf("       last queue change time (msg_ctime) : %s\n\n", ctime(&buf.msg_ctime));
}


void main(argc, argv)
 int argc;
 char *argv[];

{
  struct msqid_ds buf;
  int cmd,
      msqid,
      result;
  unsigned maxqid = MAXQID;

 if (argc == 2) {
  if ((maxqid = atoi(argv[1])) == 0) maxqid = MAXQID;
 }

 printf("msgstat: Status report of message queues. All queues with an id up");
 printf(" to msqid\n%d will be looked for.\n\n", maxqid);
 for (msqid = 0; msqid <= maxqid; msqid++) {
  if ((result = msgctl(msqid, IPC_STAT, &buf)) == -1) {
   if (errno == EINVAL) continue;
   if (errno == EIDRM) continue;
   printf("msqid = %d : %s\n", msqid, sys_errlist[errno]);
  } else Info(msqid, buf);
 }
}
```

```
/* msgctl.c  Modifiziertes Beispielprogramm aus SUN Manual: Programmers
             Overview Utilities & Libraries. Seite 61 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>



static void do_msgctl(msqid, cmd, buf)
 int msqid,
     cmd;
 struct msqid_ds* buf;

{
  register int rtrn;
  printf("\nmsgctl: calling msgctl (%d, %d, %s)\n",
          msqid, cmd, buf ? "&buf" : "(struct msqid_ds*) NULL");
  rtrn = msgctl(msqid, cmd, buf);
  if (rtrn == -1) {
   perror("msgctl: msgctl failed");
   exit(1);
  } else {
   printf("msgctl: msgctl returned %d\n", rtrn);
  }
}


void main()
{
  struct msqid_ds buf;
  int cmd,
      msqid;

  printf("All numeric input is expected to follow C conventions:\n");
  printf("\t0x... is interpreted as hexadezimal,\n");
  printf("\t0... is interpreted as octal,\n");
  printf("\totherwise decimal.\n");
  printf("Please enter arguments for msgctl() call as requested.\n");
  printf("Enter the desired msqid: ");
  scanf("%i", &msqid);
  printf("Valid msgctl commands are:\n");
  printf("\tIPC_RMID = %d\n", IPC_RMID);
  printf("\tIPC_STAT = %d\n", IPC_STAT);
  printf("Enter the value for the desired command: ");
  scanf("%i", &cmd);

  switch (cmd) {
   case IPC_STAT : do_msgctl(msqid, IPC_STAT, &buf);
                   printf("msg_perm.uid = %d\n", buf.msg_perm.uid);
                   printf("msg_perm.gid = %d\n", buf.msg_perm.gid);
                   printf("msg_perm.cuid = %d\n", buf.msg_perm.cuid);
                   printf("msg_perm.cgid = %d\n", buf.msg_perm.cgid);
                   printf("msg_perm.mode = %#o\n", buf.msg_perm.mode);
                   printf("access perms = %#o\n", buf.msg_perm.mode & 0777);
                   printf("msg_cbytes = %d\n", buf.msg_cbytes);
                   printf("msg_qbytes = %d\n", buf.msg_qbytes);
                   printf("msg_lspid = %d\n", buf.msg_lspid);
                   printf("msg_lrpid = %d\n", buf.msg_lrpid);
                   printf("msg_stime = %s", buf.msg_stime ?
                           ctime(&buf.msg_stime) : "Not set\n");
                   printf("msg_rtime = %s", buf.msg_rtime ?
                           ctime(&buf.msg_rtime) : "Not set\n");
                   printf("msg_ctime = %s", ctime(&buf.msg_ctime));
                   break;
   case IPC_RMID : do_msgctl(msqid, cmd, (struct msqid_ds*) NULL);
                   printf("Message queue %d removed\n", msqid);
                   break;
         default : printf("Invalid selection\n");
                   exit(0);
                   break;
  }
}
```

Shared-Memory: `shmstat.c`, `shmctl.c`

```
/* shmstat.c    Status aller Segment-IDs von Shared Memory. Simple Version S.F. Feb. 1994 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
#include "defs.h"
#include <errno.h>

#define MAXSHMID 5000

extern int errno;


void Info(shmid, buf)
 int shmid;
 struct shmid_ds buf;

{
  printf(" Current segment/shared memory id (shmid) : %d\n", shmid);
  printf("           owner's user id (shm_perm.uid) : %d\n", buf.shm_perm.uid);
  printf("          owner's group id (shm_perm.gid) : %d\n", buf.shm_perm.gid);
  printf("        creator's user id (shm_perm.cuid) : %d\n", buf.shm_perm.cuid);
  printf("      creators's group id (shm_perm.cgid) : %d\n", buf.shm_perm.cgid);
  printf("             access modes (shm_perm.mode) : %#o\n", buf.shm_perm.mode);
  printf("      slot sequence number (shm_perm.seq) : %d\n", buf.shm_perm.seq);
  printf("               segment key (shm_perm.key) : %ld\n", buf.shm_perm.key);
  printf("                        access permissions : %#o\n", buf.shm_perm.mode & 0777);

  printf("        segment size in bytes (shm_segsz) : %d\n", buf.shm_segsz);
  printf("  pid of last oper. on segment (shm_lpid) : %d\n", buf.shm_lpid);
  printf("     pid of creator of segment (shm_cpid) : %d\n", buf.shm_cpid);
  printf("  number of current attaches (shm_nattch) : %d\n", buf.shm_nattch);
  printf("              last attach time (shm_atime) : %s", buf.shm_atime ?
          ctime(&buf.shm_atime) : "Not set\n");
  printf("              last detach time (shm_dtime) : %s", buf.shm_dtime ?
          ctime(&buf.shm_dtime) : "Not set\n");
  printf("     last segment change time (shm_ctime) : %s\n\n", ctime(&buf.shm_ctime));
}




void main(argc, argv)
  int argc;
  char *argv[];

{
  struct shmid_ds buf;
  int cmd,
      shmid,
      result;
  unsigned maxshmid = MAXSHMID;


 if (argc == 2) {
  if ((maxshmid = atoi(argv[1])) == 0) maxshmid = MAXSHMID;
 }


 printf("shmstat: Status report of shared memory segments. All segments with");
 printf(" an ID up\nto shmid %d will be looked for.\n\n", maxshmid);
 for (shmid = 0; shmid <= maxshmid; shmid++) {
  if ((result = shmctl(shmid, IPC_STAT, &buf)) == -1) {
   if (errno == EINVAL) continue;
   printf("shmid = %d : %s\n", shmid, sys_errlist[errno]);
  } else Info(shmid, buf);
 }
}
```

```
/* shmctl.c  Modifiziertes Beispielprogramm aus SUN Manual: Programmers
             Overview Utilities & Libraries. Seite 85 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>


static void do_shmctl(shmid, cmd, buf)
 int shmid,
     cmd;
 struct shmid_ds* buf;

{
  register int rtrn;
  printf("\nshmctl: calling shmctl (%d, %d, %s)\n",
          shmid, cmd, buf ? "&buf" : "(struct shmid_ds*) NULL");
  rtrn = shmctl(shmid, cmd, buf);
  if (rtrn == -1) {
   perror("shmctl: shmctl failed");
   exit(1);
  } else {
   printf("shmctl: shmctl returned %d\n", rtrn);
  }
}


void main()
{
  struct shmid_ds buf;
  int cmd,
      shmid;

  printf("All numeric input is expected to follow C conventions:\n");
  printf("\t0x... is interpreted as hexadezimal,\n");
  printf("\t0... is interpreted as octal,\n");
  printf("\totherwise decimal.\n");
  printf("Please enter arguments for shmctl() call as requested.\n");
  printf("Enter the desired shmid: ");
  scanf("%i", &shmid);
  printf("Valid shmctl commands are:\n");
  printf("\tIPC_RMID = %d\n", IPC_RMID);
  printf("\tIPC_STAT = %d\n", IPC_STAT);
  printf("Enter the value for the desired command: ");
  scanf("%i", &cmd);

  switch (cmd) {
   case IPC_STAT : do_shmctl(shmid, IPC_STAT, &buf);
                   printf("shmid = %d\n", shmid);
                   printf("shm_perm.uid = %d\n", buf.shm_perm.uid);
                   printf("shm_perm.gid = %d\n", buf.shm_perm.gid);
                   printf("shm_perm.cuid = %d\n", buf.shm_perm.cuid);
                   printf("shm_perm.cgid = %d\n", buf.shm_perm.cgid);
                   printf("shm_perm.mode = %#o\n", buf.shm_perm.mode);
                   printf("shm_perm.seq) = %d\n", buf.shm_perm.seq);
                   printf("shm_perm.key = %ld\n", buf.shm_perm.key);
                   printf("access perms = %#o\n", buf.shm_perm.mode & 0777);

                   printf("shm_segsz = %d\n", buf.shm_segsz);
                   printf("shm_lpid = %d\n", buf.shm_lpid);
                   printf("shm_cpid = %d\n", buf.shm_cpid);
                   printf("shm_nattch = %d\n", buf.shm_nattch);
                   printf("shm_atime = %s", buf.shm_atime ?
                           ctime(&buf.shm_atime) : "Not set\n");
                   printf("shm_dtime = %s", buf.shm_dtime ?
                           ctime(&buf.shm_dtime) : "Not set\n");
                   printf("shm_ctime = %s\n\n", ctime(&buf.shm_ctime));
                   break;
```

```
   case IPC_RMID : shmctl(shmid, IPC_STAT, &buf);
                   if (buf.shm_nattch) {
                    char s[100];
                    printf("There are %d attached programs/processes to this segment!", buf.shm_nattch);
                    printf(" If you remove the\nsegment now, those processes could fail and hang.\n");
                    printf("You really want to remove the segment now (y/n)? ");
                    scanf("%s", s);
                    if (s[0] != 'y') break;
                   }
                   do_shmctl(shmid, cmd, (struct shmid_ds*) NULL);
                   printf("Segment %d finally removed\n", shmid);
                   break;
         default : printf("Invalid selection\n");
                   exit(0);
                   break;
  }

}
```

Semaphore: `semstat.c`, `semctl.c`

```
/* semstat.c    Status aller Semaphore. Simple Version S.F. Mai 1994 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <time.h>
#include "defs.h"
#include <errno.h>

#define MAXSEMID 5000

extern int errno;


void Info(semid, buf)
 int semid;
 struct semid_ds buf;

{
  union semun arg;
  int i, result;

  printf("                  Current Semaphore-Set ID : %d\n", semid);
  printf("            owner's user id (sem_perm.uid) : %d\n", buf.sem_perm.uid);
  printf("           owner's group id (sem_perm.gid) : %d\n", buf.sem_perm.gid);
  printf("         creator's user id (sem_perm.cuid) : %d\n", buf.sem_perm.cuid);
  printf("       creators's group id (sem_perm.cgid) : %d\n", buf.sem_perm.cgid);
  printf("              access modes (sem_perm.mode) : %#o\n", buf.sem_perm.mode);
  printf("       slot sequence number (sem_perm.seq) : %d\n", buf.sem_perm.seq);
  printf("                  queue key (sem_perm.key) : %ld\n", buf.sem_perm.key);
  printf("                            access permissions : %#o\n", buf.sem_perm.mode & 0777);
  printf("current # of Semaphores in set (sem_nsems) : %d\n", buf.sem_nsems);
  printf("            time of last semop (sem_otime) : %s", buf.sem_otime ?
          ctime(&buf.sem_otime) : "Not set\n");
  printf("           time of last change (msg_ctime) : %s\n", ctime(&buf.sem_ctime));

  for (i=0; i< buf.sem_nsems; i++) {
   if ((result = semctl(semid, i, GETVAL, NULL)) == -1) syserr("semctl GETVAL");
   printf("            Value of semaphore #%d (semval) : %d\n", i, result);
   if ((result = semctl(semid, i, GETPID, NULL)) == -1) syserr("semctl GETPID");
   printf("  last process modified semaphore (sempid) : %d\n", result);
   if ((result = semctl(semid, i, GETNCNT, NULL)) == -1) syserr("semctl GETNCNT");
   printf(" # of proc. waiting sem. increase (semcnt) : %d\n", result);
   if ((result = semctl(semid, i, GETZCNT, NULL)) == -1) syserr("semctl GETZCNT");
   printf("  # of proc. wait semaphore zero (semzcnt) : %d\n", result);
  }
  printf("\n\n");
}



void main(argc, argv)
 int argc;
 char *argv[];

{
  struct semid_ds buf;
  union semun arg;
  int cmd,
      semid,
      result;
  unsigned maxsemid = MAXSEMID;


 if (argc == 2) {
  if ((maxsemid = atoi(argv[1])) == 0) maxsemid = MAXSEMID;
 }

 arg.buf = &buf;

 printf("semstat: Status report of Semaphore-Sets. All Sets with an id up");
 printf(" to semid\n%d will be looked for.\n\n", maxsemid);
```

```
 for (semid = 0; semid <= maxsemid; semid++) {
  if ((result = semctl(semid, 0, IPC_STAT, arg)) == -1) {
   if (errno == EINVAL) continue;
   printf("semid = %d : %s\n", errmsg(errno));
  } else Info(semid, buf);
 }
}



/* shmctl.c  Modifiziertes Beispielprogramm aus SUN Manual: Programmers
             Overview Utilities & Libraries. Seite 85 */



#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>



static void do_shmctl(shmid, cmd, buf)
 int shmid,
     cmd;
 struct shmid_ds* buf;

{
  register int rtrn;
  printf("\nshmctl: calling shmctl (%d, %d, %s)\n",
          shmid, cmd, buf ? "&buf" : "(struct shmid_ds*) NULL");
  rtrn = shmctl(shmid, cmd, buf);
  if (rtrn == -1) {
   perror("shmctl: shmctl failed");
   exit(1);
  } else {
   printf("shmctl: shmctl returned %d\n", rtrn);
  }
}



void main()
{
  struct shmid_ds buf;
  int cmd,
      shmid;

  printf("All numeric input is expected to follow C conventions:\n");
  printf("\t0x... is interpreted as hexadezimal,\n");
  printf("\t0... is interpreted as octal,\n");
  printf("\totherwise decimal.\n");
  printf("Please enter arguments for shmctl() call as requested.\n");
  printf("Enter the desired shmid: ");
  scanf("%i", &shmid);
  printf("Valid shmctl commands are:\n");
  printf("\tIPC_RMID = %d\n", IPC_RMID);
  printf("\tIPC_STAT = %d\n", IPC_STAT);
  printf("Enter the value for the desired command: ");
  scanf("%i", &cmd);

  switch (cmd) {
   case IPC_STAT : do_shmctl(shmid, IPC_STAT, &buf);
                   printf("shmid = %d\n", shmid);
                   printf("shm_perm.uid = %d\n", buf.shm_perm.uid);
                   printf("shm_perm.gid = %d\n", buf.shm_perm.gid);
                   printf("shm_perm.cuid = %d\n", buf.shm_perm.cuid);
                   printf("shm_perm.cgid = %d\n", buf.shm_perm.cgid);
                   printf("shm_perm.mode = %#o\n", buf.shm_perm.mode);
                   printf("shm_perm.seq) = %d\n", buf.shm_perm.seq);
                   printf("shm_perm.key = %ld\n", buf.shm_perm.key);
                   printf("access perms = %#o\n", buf.shm_perm.mode & 0777);

                   printf("shm_segsz = %d\n", buf.shm_segsz);
```

```
                    printf("shm_lpid = %d\n", buf.shm_lpid);
                    printf("shm_cpid = %d\n", buf.shm_cpid);
                    printf("shm_nattch = %d\n", buf.shm_nattch);
                    printf("shm_atime = %s", buf.shm_atime ?
                            ctime(&buf.shm_atime) : "Not set\n");
                    printf("shm_dtime = %s", buf.shm_dtime ?
                            ctime(&buf.shm_dtime) : "Not set\n");
                    printf("shm_ctime = %s\n\n", ctime(&buf.shm_ctime));
                    break;
   case IPC_RMID : shmctl(shmid, IPC_STAT, &buf);
                    if (buf.shm_nattch) {
                     char s[100];
                     printf("There are %d attached programs/processes to this segment!", buf.shm_nattch);
                     printf(" If you remove the\nsegment now, those processes could fail and hang.\n");
                     printf("You really want to remove the segment now (y/n)? ");
                     scanf("%s", s);
                     if (s[0] != 'y') break;
                    }
                    do_shmctl(shmid, cmd, (struct shmid_ds*) NULL);
                    printf("Segment %d finally removed\n", shmid);
                    break;
         default : printf("Invalid selection\n");
                    exit(0);
                    break;
  }

}
```

Literatur

Bach, Maurice: *Wie funktioniert das Unix Betriebssystem?* Hanser Verlag, ISBN 3-446-15693-3 (1991)

Rochkind, Marc: *Fortgeschrittene Unix-Programmmierung*, Hanser Verlag, ISBN 3-446-15202-4 (1988)

Stevens, Richard: *Programmierung von Unix-Netzen*, Hanser Verlag.

Stevens, Richard: *Programmierung in der Unix-Umgebung*, Addison-Wesley, ISBN 3-89319-814-8 (1995)