Hochschule Esslingen
University of Applied Sciences

# Mechanismen der Interprozesskommunikation

## Lerntext zur Verwendung im Modul Betriebssysteme

Prof. Dr. Peter Väterlein
Hochschule Esslingen
Stand: November 2013

# Inhaltsverzeichnis



 

# 1 Einführung

Im Lauf dieses Semesters haben Sie gelernt, was ein Prozess ist und wie Betriebssysteme wie Linux CPU-Zeit und Hauptspeicher unter den laufenden Prozessen aufteilen. Nun geht es darum, wie die Prozesse zusammenwirken, wie sie Daten austauschen und wie sie sich gegenseitig beeinflussen können.

Wie immer, wenn es ums Programmieren geht, ist es eine Sache, darüber zu lesen. Eine ganz andere Sache ist es, Programme selbst zu schreiben und zu testen. Deshalb die eindringliche Bitte: Probieren Sie die Beispiele in diesem Skript aus. Geben Sie sich nicht damit zufrieden, wenn Sie den Quellcode „abgemaust“ und vielleicht sogar zum Laufen gebracht haben.

Sie werden die Themen in diesem Skript nur durchdringen können, wenn Sie mit den Programmen spielen: probieren Sie Varianten aus, untersuchen Sie mit Hilfsmitteln wie `strace`, wie das Programm funktioniert.

Und nutzen Sie die Lernteams. Treffen Sie sich **mindestens** einmal untereinander, um über das zu diskutieren, was Sie gelesen und ausprobiert haben. Denn

> **Menschen behalten 20% von dem, was sie hören, 30% von dem, was sie lesen, aber 90% von dem, was sie tun**

# 2 Übersicht über die IPC-Mechanismen

Prinzipiell kann man solche Mechanismen der *Interprozesskommunikation (IPC)* unterscheiden, die der Übermittlung von Daten zwischen Prozessen dienen und solchen, die den Ablauf von Prozessen steuern bzw. Prozesse synchronisieren. In Abbildung 1 sind die gängigsten IPC-Mechanismen unter UNIX dargestellt.

## Signale

Ein Signal enthält keine Informationen

sind die einfachste Art, eine Nachricht zwischen zwei Prozessen auszutauschen. Ein Signal enthält außer der Tatsache, dass es geschickt wurde, keine weiteren

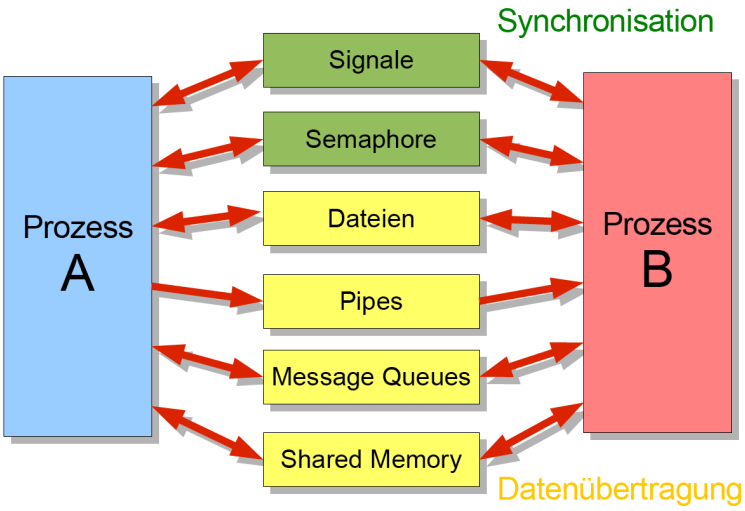


**Abbildung 1:** *Eine Übersicht über die gängigsten IPC-Mechanismen unter UNIX*

Informationen. Der Empfänger weiß nicht einmal, von welchem Prozess das Signal geschickt wurde.

Signale werden in der Regel dazu verwendet, um Prozesse „von außen“ zu steuern. Ihr größter Nachteil liegt darin, dass der Empfang eines Signals nicht quittiert wird. Das heißt, ein Absender weiß nie, ob der Empfänger das Signal auch richtig erhalten hat.

## Semaphore

Semaphore = Systemweit definierte Variable

sind eigentlich Variable, die nicht innerhalb eines Prozesses bzw. seines virtuellen Speichers, sondern systemweit definiert sind. Zusätzlich zu seinem Wert (einer Zahl) sind für ein Semaphor zwei Funktionen definiert (`up()` und `down()`, vgl. Abschnitt 4.2), In der ursprünglichen Form war nicht vorgesehen, auf den Inhalt eines Semaphors direkt zuzugreifen. Die aktuelle Linux-Implementierung lässt dies allerdings zu.

Semaphore werden in der Regel als Hilfsmittel zur Steuerung des Zugriffs auf unteilbare Ressourcen verwendet.

## Dateien

würde man auf den ersten Blick vielleicht gar nicht als Mittel der Kommunikation zwischen Prozessen einstufen. Dateien sind geeignet für die Übertragung großer Datenmengen bis zu Gigabytes. Ein Vorteil von Dateien ist die *persistente* Datenhaltung, d.h. die Daten sind auch noch da, wenn einer der beteiligten Prozesse oder sogar das ganze System abstürzt und neu gestartet werden muss.

Dateien können sehr große Datenmengen übertragen

Dateien gestatten einen wahlfreien Zugriff auf die Daten, man muss also nicht alle vorangegangenen Daten gelesen haben, um auf ein bestimmtes Byte in der Mitte der Datei zugreifen zu können. Und schließlich ist der Zugriff auf Dateien einfach zu programmieren.

Ein Nachteil von Dateien ist die vergleichsweise geringe Geschwindigkeit, da alle Transfers über die Festplatte erfolgen. Die Synchronisation der (schreibenden) Zugriffe von verschiedenen Prozessen müssen die Anwendungen selbst sicher stellen.

## Pipes

sind den Dateien sehr ähnlich. Allerdings sind Pipes auf die Übermittlung kleiner Datenmengen zwischen zwei Prozessen spezialisiert. Der Zugriff auf die Daten erfolgt nach dem FIFO-Prinzip (*First-In First-out*). Der Zugriff auf Pipes ist dem auf Dateien sehr ähnlich oder im Fall der Named Pipes sogar identisch.

Pipes haben eingebaute Synchronisationswerkzeuge

Ein großer Vorteil der Pipes ist die automatisch integrierte Synchronisation des Zugriffs

## Message Queues

Message Queues können mehr als zwei Endpunkte haben

sind eigentlich verfeinerte Pipes. Sie können kleine Datenvolumina nicht nur zwischen zwei Endpunkten, sondern auch zwischen einer größeren Zahl von Prozessen vermitteln. Der Zugriff erfolgt, wie bei den Pipes, prinzipiell nach dem FIFO Prinzip, das aber durch die Verwendung so genannter *Message Identifier* aufgeweicht werden kann.

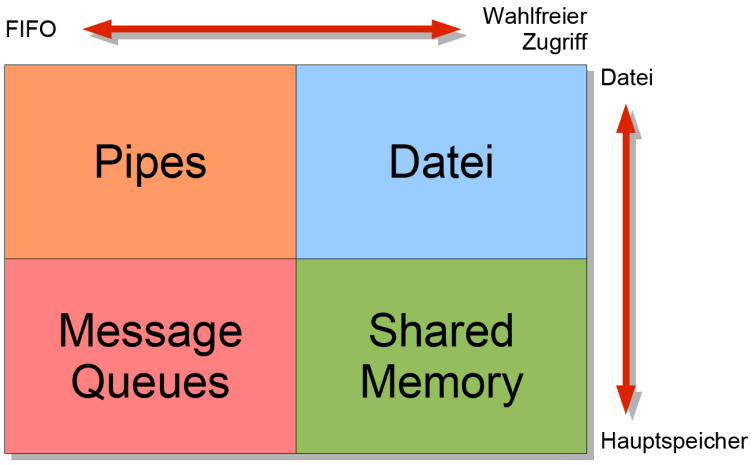


**Abbildung 2:** *Die vier gängigsten Methoden, Daten zwischen UNIX-Prozessen zu transferieren, sortiert nach Speicherort und Art des Zugriffs*

Wie die Pipes verfügen auch die Message Queues über eine automatische Synchronisation der Zugriffe.

### Shared Memory

bezeichnet einen Bereich im physikalischen Speicher eines Rechners, der gleichzeitig von mehreren Prozessen gelesen und beschrieben werden kann. Dazu wird dieser Bereich in den virtuellen Speicher der Prozesse eingeblendet und ist dann über eine Pointer-Variable zugänglich.

Shared Memory ist schnell!

Diese Form der Datenübermittung ist die weitaus schnellste aller genannten Möglichkeiten. Das übertragbare Datenvolumen ist allerdings durch die Größe des zur Verfügung stehenden Hauptspeichers begrenzt. Für die Synchronisation der Zugriffe müssen die beteiligten Anwendungen selbst sorgen.

# 3 Datenübermittlung zwischen Prozessen

## 3.1 Dateien

Die Behandlung von Dateien lernt man normalerweise relativ früh in der Programmierausbildung. Die Systemaufrufe, mit denen man eine Datei manipu-

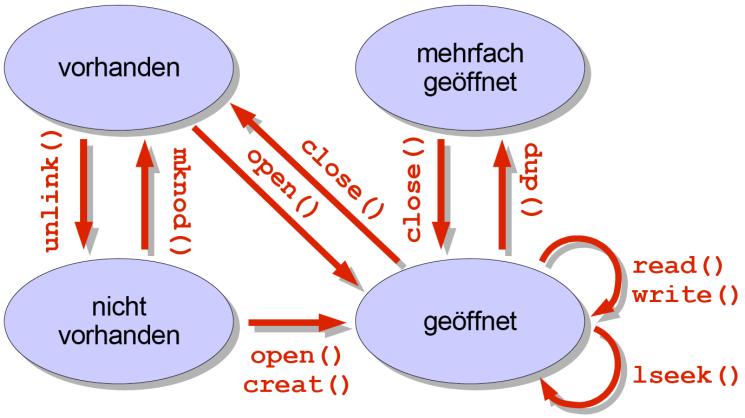


**Abbildung 3:** *Mögliche Zustände einer Datei und die Systemaufrufe, um von einem Zustand zum anderen zu kommen.*

liert, sind besonders wichtig, weil gemäß der UNIX-Grundphilosophie *„Alles ist Datei“* auch viele andere Objekte in einem UNIX System mit diesen elementaren Funktionen bedient werden können.

Abbildung 3 zeigt die Zustände einer Datei in einem UNIX System und die Systemaufrufe, mit denen man zwischen diesen Zuständen hin und her wechseln kann.

**`int mknod(const char *path, mode_t mode, dev_t dev)`**
: erzeugt eine Datei oder eine Spezialdatei (Named Pipe, Gerätedatei, ...) ohne sie zu öffnen. Weitere Details finden Sie in der Manpage zu dieser Funktion[1].

**`int creat( const char *path, mode_t mode )`** erzeugt eine bisher nicht existierende Datei und öffnet sie. Der Rückgabewert ist der Dateideskriptor, der die geöffnete Datei innerhalb des aufrufenden Prozesses beschreibt. Ganz wichtig ist es, wie bei **`open()`**, die Zugriffsrechte auf die neue Datei zu setzen, indem man dem Parameter **`mode`** auf eine Oktalzahl wie **`0600`** oder **`0644`** setzt.

**`int open(const char *path, int flags, mode_t mode)`**
: öffnet eine vorhandene Datei. Der Rückgabewert ist der Dateideskriptor, der die geöffnete Datei innerhalb des aktuellen Prozesses beschreibt.

[1] Da es sowohl ein Shellkommando mknod als auch einen Systemaufruf mknod gibt, ist es wichtig, die richtige Kapitelnummer in den Man-Pages anzugeben. In diesem Fall also „`man 2 mknod`" wie zu allen anderen in diesem Skript beschriebenen Funktionen.

**`int dup(int oldd)`** ordnet einer geöffneten Datei einen zweiten Dateideskriptor zu. Der bisherige Deskriptor wird als Argument übergeben, der Rückgabewert der Funktion ist der neue, zusätzliche Deskriptor. Diese Funktion ist beispielsweise nützlich, wenn man an zwei unterschiedlichen Stellen einer Datei gleichzeitig lesen möchte.

**`int close(int d)`** schließt die durch den Dateideskriptor **`d`** beschriebene Datei. Sind der Datei mehrere Deskriptoren zugeordnet, wird die Datei erst wirklich geschlossen, wenn der letzte Deskriptor geschlossen wurde. Beim Schließen werden auch Daten, die bisher noch im Speicher zwischengespeichert waren, auf den Datenträger geschrieben.

**`int lseek(int fildes, off_t offset, int whence)`**
positioniert den Schreib-/Lesezeiger innerhalb einer geöffneten Datei, die durch den Deskriptor **`filedes`** beschrieben wird. Der Parameter **`whence`**[2] gibt an, ob die im Parameter **`offset`** angegebene Zahl von Bytes vom Anfang der Datei (Wert von **`whence`** ist **`SEEK_SET`**, von der aktuellen Position (**`SEEK_CUR`**) oder vom Ende der Datei (**`SEEK_END`**) gezählt werden sollen.

**`ssize_t read(int d, void *buf, size_t nbytes)`** liest maximal **`nbytes`** Bytes aus der durch den Deskriptor **`d`** beschriebenen Datei und speichert sie in die Variable auf die der Pointer **`*buf`** zeigt. Da **`read()`** immer nur bis zum Zeilen- oder Dateiende liest, kann die Zahl der tatsächlich gelesenen Bytes kleiner sein. Diese Zahl wird als Rückgabewert der Funktion übergeben.

**`ssize_t write(int d, const void *buf, size_t nbytes)`**
schreibt **`nbytes`** Bytes aus der Variablen **`*buf`** in die Datei mit dem Deskriptor **`d`**. Der Rückgabewert der Funktion ist die Zahl der tatsächlich geschriebenen Bytes. Wenn der ungleich **`nbytes`** ist, kann man davon ausgehen, dass etwas schief gegangen ist.

Das folgende Beispielprogramm öffnet eine Datei, schreibt einen kurzen Text hinein und schließt die Datei wieder:

[2] „whence“ heißt eigentlich „woher“. Die Verwendung des Wortes ist an dieser Stelle sprachlich also nicht in Ordnung, wird aber aus historischen Gründen beibehalten.

```
/************************************************************
 * filetest.c:                                              *
 * Schreibt einen String in eine Datei                      *
 *                              Peter Vaeterlein, 2006-11-25 *
 ************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int fd, cnt;
char testtext[80];

main()
{
  fd = open ( "./testdatei", O_CREAT|O_RDWR, 0600 );
  sprintf ( testtext,
            "Dieser Text wird testhalber geschrieben\n" );
  cnt = write ( fd, testtext, strlen ( testtext ));
  printf ( "Es wurden %d Bytes geschrieben\n", cnt );
  close ( fd );
}
```

**Übung:** Probieren Sie dieses Programm auf Ihrem Rechner aus. Schreiben Sie ein entsprechendes Programm, das die Datei **testdatei** wieder ausliest. Beobachten Sie beide Programme auch mit dem Hilfsprogramm **strace**, um die tatsächlich ausgeführten Systemaufrufe zu studieren.

## 3.2 Pipes

Pipes sind genau das, was der Name sagt: Rohre für Daten, die zwischen zwei Prozessen ausgetauscht werden sollen. Am einen Ende werden die Daten in die Pipe eingefüllt, am anderen Ende kommen sie in genau derselben Reihenfolge wieder heraus. Der Zugriff erfolgt nach dem *„First-In First-Out (FIFO) Prinzip“*. Wie die meisten Wasserrohre sind Pipes Einbahnstrassen. Das heißt, ein Ende ist der Sender, das andere der Empfänger.

Es gibt unter UNIX verschiedene Arten von Pipes: *Unnamed Pipes*, *Named Pipes* und *High-Level Pipes*, die im folgenden genauer beschrieben werden.

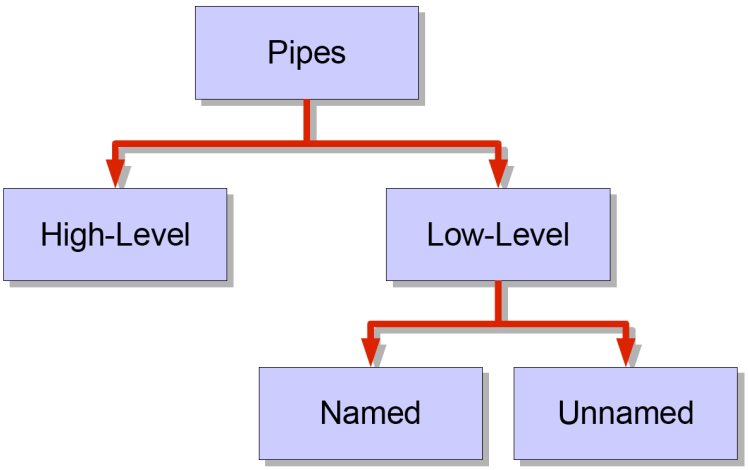


**Abbildung 4:** *Stammbaum der verschiedenen Formen von Pipes unter UNIX*

### 3.2.1 Unnamed Pipes

Unnamed P
nur Eltern-

Unnamed Pipes können nur zwischen einem Prozess und seinen Kindern aufgebaut werden. Sie sind im Dateisystem des Rechners nicht sichtbar sondern existieren nur im Hauptspeicher. Daher existieren Unnamed Pipes auch nur so lange wie die Prozesse, die sie erzeugt haben. Über eine Unnamed Pipe können maximal 4096 Bytes (4 kB) „atomar“ übertragen werden, das heißt, ohne dass ein Prozesswechsel die Übertragung unterbrechen könnte.

Zur Verwendung von Unnamed Pipes ist nur ein einziger zusätzlicher Systemaufruf notwendig:

`int pipe(int *fildes)` richtet eine Unnamed Pipe ein. Das Argument ist ein Pointer auf ein Array, das zwei int-Zahlen enthält. Das erste Element (`filedes[0]`) ist der Dateideskriptor, der zum Lesen verwendet wird. Das zweite Element (`filedes[1]`) ist der Dateideskriptor, der zum Schreiben verwendet wird.

Das normale Szenario zum Einsatz von Unnamed Pipes sieht wie folgt aus:

- Die Unnamed Pipe wird mit dem Systemaufruf `pipe()` eingerichtet
- Mit dem Systemaufruf `fork()` wird eine identische Kopie des aufrufenden Prozesses erzeugt

- In einem der beiden Prozesse (dem späteren Sender) wird der Dateideskriptor zum Lesen geschlossen
- Im anderen Prozess (dem späteren Empfänger) wird der Dateideskriptor zum Schreiben geschlossen
- Nun kann, wie von den Dateien her gewohnt, mit **`write()`** in die Pipe geschrieben und mit **`read()`** daraus gelesen werden
- Zum Abschluss müssen die noch geöffneten Dateideskriptoren mit **`close()`** geschlossen werden

Die eingebaute Synchronisation funktioniert bei allen Pipes so, dass der lesende Prozess (Systemaufruf **`read()`**) blockiert, wenn die Pipe leer ist und erst weiterläuft, wenn der schreibende Prozess die Pipe mit dem Systemaufruf **`write()`** Daten in die Pipe „einfüllt“.

Das folgende Beispielprogramm zeigt die prinzipielle Vorgehensweise:

```
/************************************************************
 * upipe.c:                                                 *
 * Demoprogramm fuer Unnamed Pipes                          *
 * Peter Vaeterlein, 2006-11-25                             *
 ************************************************************/
#include<stdio.h>
#include<string.h>

int pdes[2];
int num;
int status;
char ptr[256];

main ()
{
  pipe(pdes); /* Erzeuge Pipe
                 pdes[0]: Dateideskriptor Lesen
                 pdes[1]: Dateideskriptor Schreiben
              */

  if ( fork() == 0 ){   /* Kind = Lesen */
    close(pdes[1]);     /* wird nicht mehr gebraucht */
    read( pdes[0], ptr, 4096);
                        /* Ausgabe auf Konsole */
    printf ( "[%d]: Aus Pipe gelesen: %s\n",
             getpid(), ptr );
  } else {              /* Elter = Schreiben */
    close(pdes[0]);     /* wird nicht mehr gebraucht */
    printf ( "<%d>: Was soll in die Pipe : ",
             getpid() );
    scanf ( "%s", ptr );
    write( pdes[1], ptr, strlen ( ptr ));
    wait ( &status );
```

```
    }
}
```

**Übung:** Probieren Sie dieses Programm auf Ihrem Rechner aus und beobachten Sie, was das Programm tut, mit Hilfe von strace. Damit Sie auch die Aktionen des Kindprozesses mitbekommen, verwenden Sie strace mit den Optionen -o und -ff.
Können Sie sich denken, warum die Dateideskriptoren „3“ und „4“ sind?

### 3.2.2 Named Pipes

sind mit gewöhnlichen Dateien noch enger verwandt, als die Unnamed Pipes. Sie haben einen i-Node und mindestens einen Verzeichniseintrag, über den sie im Dateisystem sichtbar sind. Daher bleiben sie auch, genau wie Dateien, so lange präsent, bis sie explizit gelöscht werden.

Durch die Sichtbarkeit im Dateisystem können prinzipiell alle Prozesse eine Named Pipe nutzen, unabhängig von ihrem „Verwandschaftsgrad“ zu dem erzeugenden Prozess. Wie bei Dateien müssen natürlich die entsprechenden Zugriffsrechte gesetzt sein.

Zur Erzeugung einer Named Pipe wird der auf Seite 9 beschriebene Systemaufruf **`mknod()`** verwendet. Dem Parameter **`mode`** muss neben den Zugriffsrechten (als dreistellige Oktalzahl[3]) die vordefinierte Konstante **`S_IFIFO`** enthalten. Beide Größen werden bitweise verodert. Ein möglicher Wert für den Parameter **`mode`** wäre also **`S_IFIFO | 0644`**.

Nachdem die Named Pipe erzeugt wurde, muss sie wie eine Datei zum Lesen oder Schreiben geöffnet werden. Danach ist Lesen und Schreiben wie in eine Datei möglich. Allerdings funktioniert das nur in eine Richtung. Wenn man eine bidirektionale Verbindung zwischen zwei Prozessen benötigt, muss man zwei Pipes einrichten.

[3] zum Beispiel 0600 oder 0644 (Die führende Null sagt dem Betriebssystem, dass es sich bei dieser Zahl um eine Oktalzahl handelt, so wie das Prefix „0x“ eine Hexadezimalzahl kennzeichnet).

Die folgenden beiden Programme demonstrieren den Einsatz von Named Pipes. Das erste Programm ist der Empfänger, der eine Named Pipe einrichtet, sie öffnet und dann daraus liest:

```
/************************************************************
 * npipe_recv.c                                             *
 * Empfaenger fuer Kommunikation ueber eine Named Pipe      *
 *                              Peter Vaeterlein, 2006-11-25 *
 ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MODE 0644
#define MAXLEN 255

int fd;
char text[MAXLEN];

main() {
  mknod ( "./testpipe", S_IFIFO | MODE , 0 );
  fd = open ( "./testpipe", O_RDONLY, 0644 );
  for ( ;; ) {
    memset ( text, '\0', sizeof ( text ));
    read ( fd, text, MAXLEN );
    if ( text [0] == '.' ) {
      unlink ( "./testpipe" );
      exit (0);
    }
    printf ( "Aus der Pipe: %s", text );
  }
}
```

Das zweite Programm ist der Sender, der in die Pipe schreibt. Dieses Programm setzt voraus, dass die Pipe bereits existiert.

```
/************************************************************
 * npipe_send.c                                             *
 * Sender fuer Kommunikation  ueber eine Named Pipe         *
 * Peter Vaeterlein, 2006-11-25                             *
 ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

#define MAXLEN 255
```

```
int fd;
char text [MAXLEN];

main()
{
  fd = open ( "./testpipe", O_WRONLY, 0644 );
  do {
    printf ( "Was soll in die Pipe hinein : " );
    fgets ( text, MAXLEN, stdin );
    write ( fd, text, strlen ( text ));
  } while ( text [0] != '.' );
}
```

**Übung:** Probieren Sie die beiden Programme aus. Starten Sie jedes Programm in einem eigenen Terminalfenster. Kontrollieren Sie auch hier das Verhalten der Programme, indem Sie `strace` verwenden.

### 3.2.3 High-Level Pipes

sind eine Variante der Unnamed Pipes[4]. Zur Einrichtung einer High-Level Pipe dient der folgende Systemaufruf:

**`FILE *popen(const char *command, const char *type)`**
: richtet zunächst eine Unnamed Pipe ein. Dann wird mit **`fork()`** ein Kindprozess erzeugt. In diesem Kindprozess wird per **`exec()`** das als **`command`** übergebene Kommando geladen. Wenn der Parameter **`type`** den Wert **`r`** hat, wird die Pipe mit dem stdout-Kanal des ausgeführten Kommandos verbunden. Hat der Parameter **`type`** den Wert **`w`**, wird der stdin-Kanal des aufgerufenen Programms mit der Pipe verbunden. Das jeweils andere Ende der Pipe wird als Rückgabewert an den aufrufenden Prozess zurück gegeben. Achtung: Dieser Rückgabewert ist ein Pointer auf den Typ **`FILE`**.

Die High-Level Pipe ist eine Art „Nabelschnur“ zu einem neu gestarteten Programm. Wie es für Pipes charakteristisch ist, funktioniert diese Nabelschnur nur

[4] Wenn Sie später das Verhalten der High-Level Pipe mit strace untersuchen, werden Sie feststellen, dass die Pipe intern mit der Funktion `pipe()` eingerichtet wird.

in eine Richtung. Welche das ist, muss bei der Einrichtung der Pipe angegeben werden.

Der Rückgabewert des Systemaufrufs **popen()** ist ein Pointer auf den Typ **FILE**. Deshalb kann man eine High-Level Pipe nicht mit **read()** oder **write()** bedienen, sondern muss die Bibliotheksfunktionen **fprintf()** bzw. **fscanf()** benutzen.

Geschlossen wird eine High-Level Pipe mit dem Systemaufruf

**int pclose(FILE *stream)** der nicht nur die Pipe schließt, sondern zudem auch auf das Ende des Kindprozesses wartet. Als Argument wird der Funktion der von **popen()** gelieferte Deskriptor bergeben. Der Rückgabewert ist der Exit-Code der **wait()**-Funktion, die auf das Ende des Kindprozesses gewartet hat.

Das folgende Beispielprogramm öffnet eine High-Level Pipe, an deren anderem Ende ein **ls** Prozess gestartet wird. Die Ausgabe dieses Datei-Listings wird durch die Pipe geleitet und formatiert ausgegeben.

```
/*************************************************************
 * hlpipe.c                                                  *
 * Demoprogramm fuer die Benutzung von High-Level Pipes      *
 *                              Peter Vaeterlein, 2006-11-25 *
 *************************************************************/

#include <stdio.h>
#include <string.h>

#define MAXLEN 255

FILE *fd;
char text[MAXLEN];

main()
{
  fd = popen ( "ls -l", "r" );

  while ( fgets ( text, MAXLEN, fd ) > 0 ) {
    printf ( "Aus der Pipe: %s", text );
    text [0] = '\0';
  }

  pclose ( fd );
}
```

**Übung:** Probieren Sie das Programm aus und untersuchen Sie das Verhalten mit strace. Achten Sie dabei besonders auf die Verwendung der grundlegenden Systemaufrufe `pipe()` und `fork()`.

## 3.3 Message Queues

Vor allem *Named* Pipes sind potenziell langsam, weil sie auf der Festplatte zwischengespeichert werden[5]. Deshalb wurden die *Message Queues* als schnelles Kommunikationsmittel zwischen beliebigen Prozesse entwickelt, das nach dem FIFO-Prinzip und nur im Hauptspeicher arbeitet.

Aber Message Queues können noch mehr. Sie können mehr als nur zwei Prozesse verbinden. Jeder teilnehmende Prozess kann als Sender und Empfänger fungieren. Welcher Prozess eine Nachricht empfängt, entscheidet sich an ihrem „Typ“, einer positiven ganzen Zahl, die jedem versandten Paket mitgegeben wird. Jeder Empfänger hat ebenfalls einen Typ. Dabei gibt es verschiedene Möglichkeiten

`n > 0` : Ist der Typ eine positive Zahl, wird die nächste Nachricht von diesem Typ gelesen

`n = 0` : In diesem Fall wird die nächste Nachricht, unabhängig von ihrem Typ, gelesen

`n < 0` : In diesem Fall wird die nächste Nachricht gelesen, deren Typ kleiner oder gleich dem Betrag von `n` ist

Gibt es mehrere potenzielle Empfänger für einen bestimmten Nachrichtentyp, werden diese abwechselnd bedient.

In Abbildung 5 ist die Struktur einer Message Queue mit zwei beteiligten Prozessen schematisch dargestellt. Man erkennt, dass die Message Queue ein Objekt ist, das sich außerhalb aller Prozesse befindet. Das Betriebssystem erkennt

[5] Heute ist das allerdings meist kein Problem mehr, da die meisten Computer die Daten, die auf Festplatte gespeichert werden sollen, im Hauptspeicher zwischenspeichern („cachen“), so dass gute Chancen bestehen, dass es die Daten in der Pipe nie wirklich auf die Festplatte schaffen.

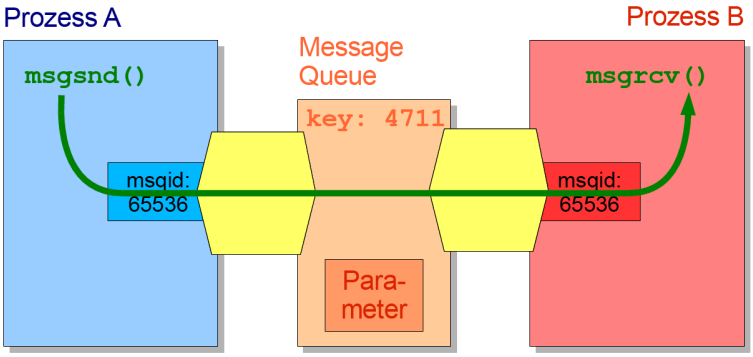


**Abbildung 5:** *Prinzipielle Struktur einer Message Queue*

eine bestimmte Message Queue an Ihrem *Key*, einer systemweit eindeutigen Kennung. Will ein Prozess eine Message Queue benutzen, muss er sie, genau wie eine Datei, erst öffnen. Genau wie eine geöffnete Datei kann der Prozess die Message Queue über einen Deskriptor (**msqid**) ansprechen.

Zur Verwendung von Message Queues gibt es eine Reihe spezieller Systemaufrufe, die im Folgenden erklärt werden.

### Einrichten und Ankoppeln einer Message Queue

Mit dem Systemaufruf

```
int msgget(key_t key, int msgflg);
```

wird eine Message Queue bei Bedarf eingerichtet und mit dem aktuellen Prozess verbunden. Jede Message Queue wird durch einen systemweit eindeutigen **key** beschrieben. Der häufig als **msqid** bezeichnete Rückgabewert der Funktion **msgget()** ist ein Deskriptor, vergleichbar dem von **open()** zurückgegebenen Dateideskriptor. Mit dem Parameter **msgflg** lässt sich das Verhalten der Message Queue steuern. Es gibt eine Reihe von vordefinierten Konstanten, die sich durch bitweises verodern (Operator „|“) verknüpfen lassen:

Merke: Der key kennzeichnet eine Message Queue systemweit, die msqid kennzeichnet die Message Queue in einem bestimmten Prozess.

**IPC_CREAT** erzeugt eine Message Queue, wenn zum angegebenen Key noch keine existiert

**IPC_EXCL** führt zu einem Fehler, wenn gleichzeitig das **IPC_CREAT** Flag gesetzt ist und eine Message Queue mit dem angegebenen Key bereits existiert.

Der Parameter **msgflg** enthält darüber hinaus die Zugriffsrechte auf die Message Queue, die identisch mit den Zugriffsrechten auf eine Datei sind. Setzt man diese Rechte bei der Einrichtung einer Message Queue nicht, so kann man später nicht auf die Message Queue zugreifen. Meist hilft dann nur noch ein Reboot, um die Queue wieder los zu werden. Die Rechte werden als dreistellige Oktalzahl angegeben, die mit den anderen Konstanten bitweise verodert wird. Ein möglicher Wert für **msgflg** wäre also **IPC_CREAT|0600**.

Der Key kann eine beliebige Zahl sein. Setzt man als Key die vordefinierte Konstante **IPC_PRIVATE** ein, kann die Message Queue nur vom aufrufenden Prozess und seinen Kindern, an die die **msqid** vererbt wird, verwendet werden.

### Eine Nachricht in die Message Queue schicken

Um Nachrichten über eine Message Queue zu verschicken, kann man ausnahmsweise nicht die Standardaufrufe **read()** und **write()** verwenden. Zum Verschicken einer Nachricht wird der Systemaufruf

```
int msgsnd(int msqid, struct msgbuf *msgp,
           size_t msgsz, int msgflg)
```

verwendet. Die Parameter dieser Funktion haben folgende Bedeutung:

**msqid** : Gibt an, in welche Message Queue die Nachricht geschickt werden soll

**msgp** : Ein Pointer auf die Nachricht die verschickt werden soll. Dieser Zeiger zeigt auf einen speziellen Strukturtyp, der neben der eigentlichen Nachricht noch den Typ der Nachricht enthält. Dieser Strukturtyp wird vom Anwendungsentwickler selbst definiert. Ein Besipiel wäre

```
struct msgbuf {
                long mtype;
                char mtext [255];
              }
```

Anstelle des Strings kann aber auch eine andere Struktur oder Union stehen

**msgsz** : Die Länge der Nachricht, d.h. im Beispiel oben die Länge des Strings **mtext** (255)

**msgflag** : Flags, die das Verhalten der Funktion steuern. Mögliche Werte sind

**IPC_NOWAIT** : Normalerweise blockiert die Funktion **msgsnd** den Prozess, wenn die Nachricht nicht sofort abgesetzt werden kann, weil beispielsweise die Message Queue voll ist. Ist aber dieses Flag gesetzt, kehrt die Funktion sofort mit einer entsprechenden Fehlermeldung zurück. Die Nachricht wird in diesem Fall nicht verschickt.

**MSG_NOERROR** : Wenn die Nachricht länger ist, als der von der Message Queue vorgesehene Platz, bricht die Funktion normalerweise mit einem Fehler ab. Ist das Flag **MSG_NOERROR** gesetzt, wird die Nachricht abgeschnitten und die Funktion fortgesetzt.

Die verschiedenen Flags können, wie gewohnt, bitweise verodert werden.

Der Rückgabewert der Funktion **msgsnd** ist 0 im Erfolgsfall und -1 im Fehlerfall.

### Lesen aus einer Message Queue

Zum Lesen aus einer Message Queue dient der Systemaufruf

```
ssize_t msgrcv(int msqid, struct msgbuf *msgp,
      size_t msgsz, long msg-typ, int msgflg)
```

Die Parameter dieser Funktion haben folgende Bedeutung

**msqid** : Kennzeichnet die Message Queue, aus der gelesen werden soll.

**msgp** : Pointer auf die Variable, in der das gelesene Datenpaket gespeichert werden soll. Der Datentyp muss dem entsprechen, der zur Verwendung mit **msgsnd** definiert wurde.

**msgsz** : Die Länge der eigentlichen Daten (vgl. **msgsnd**

**msg-typ** : Der Typ der zu lesenden Nachricht. Ist **msg-typ** gleich Null, wird die nächste Nachricht, unabhängig von ihrem Typ, gelesen. Ist der Wert größer als Null, wird die nächste Nachricht von diesem Typ gelesen. Ist gleichzeitg das Flag **MSG_EXCEPT** gesetzt, wird die nächste Nachricht gelesen, deren Typ *nicht* **msg-typ** ist. Ist **msg-typ** kleiner als Null, so wird die erste Nachricht gelesen, deren Typ kleiner oder gleich dem Betrag dieser Zahl ist.

**msgflg** : Die Flags, die das Verhalten der Funktion steuern

**IPC_NOWAIT** : Normalerweise blockiert die Funktion **msgrcv**, wenn die Message Queue keine geeigneten Nachrichten enthlt. Ist aber dieses Flag gesetzt, kehrt die Funktion sofort mit einer entsprechenden Fehlermeldung zurück.

**MSG_EXCEPT** : Wenn **msg-typ** größer als Null ist, wird die erste Nachricht gelesen, deren Typ *nicht* gleich **msg-typ** ist.

Der Rückgabewert der Funktion ist die Länge der tatsächlich gelesenen Nachricht (ohne die Bytes für den Typ).

### Verändern der Eigenschaften einer Message Queue

Die Message Queue ist, wie vorher erwähnt, ein Objekt, das nicht Teil eines Prozesses ist. Deshalb ist auch eine direkte Manipulation der Eigenschaften der Message Queue aus einem Prozess heraus nicht möglich. Um trotzdem die Message Queue aus einem Prozess heraus steuern zu knnen, ist es möglich, ihre Parameter in eine Datenstruktur innerhalb des Prozesses zu kopieren, dort wo möglich und notwendig zu modifizieren und schließlich wieder in die Message Queue zurück zu transferieren. Abbildung 6 skizziert dieses Verfahren.

Die Datenstruktur, in der die Eigenschaften der Message Queue abgebildet werden, heißt **msqid_ds**. Diese Struktur hat folgende Gestalt:

```
struct msqid_ds {
     struct ipc_perm msg_perm;     /* Ownership and permissions */
     time_t          msg_stime;    /* Time of last msgsnd() */
     time_t          msg_rtime;    /* Time of last msgrcv() */
     time_t          msg_ctime;    /* Time of last change */
     unsigned long   __msg_cbytes; /* Current number of bytes in
                                      queue (non-standard) */
     msgqnum_t       msg_qnum;     /* Current number of messages
                                      in queue */
     msglen_t        msg_qbytes;   /* Maximum number of bytes
                                      allowed in queue */
     pid_t           msg_lspid;    /* PID of last msgsnd() */
     pid_t           msg_lrpid;    /* PID of last msgrcv() */
};
```

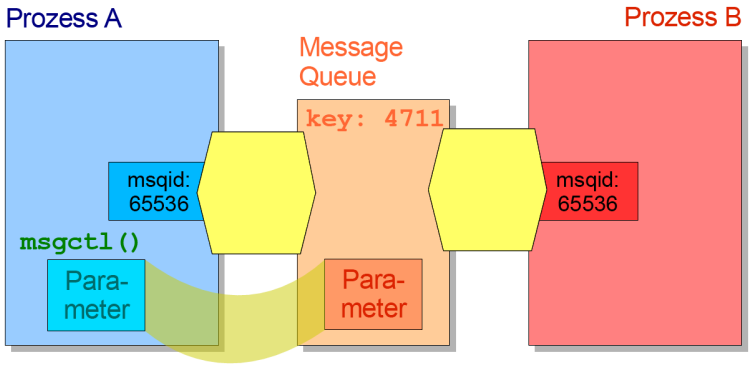


**Abbildung 6:** *Prinzip des Zugriffs auf die Parameter einer Message Queue*

Die meisten Felder dieser Struktur können nicht verändert werden[6]. Einige, wie die Zugriffsrechte lassen sich verändern.

Die Zugriffsrechte sind in einer weiteren Struktur namens **ipc_perm** verborgen. Deren Definition lautet

```
struct ipc_perm {
     key_t key;            /* Key supplied to msgget() */
     uid_t uid;            /* Effective UID of owner */
     gid_t gid;            /* Effective GID of owner */
     uid_t cuid;           /* Effective UID of creator */
     gid_t cgid;           /* Effective GID of creator */
     unsigned short mode;  /* Permissions */
     unsigned short seq;   /* Sequence number */
};
```

[6] Man kann sie natürlich verändern, aber diese Änderungen werden ignoriert, wenn die Daten später in die Message Queue zurück übertragen werden

Um die Datenstruktur **msqid_ds** zwischen der Message Queue und einem der mit ihr verbundenen Prozesse hin und her zu transferieren, dient der Systemaufruf

```
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Die Parameter dieser Funktion haben folgende Bedeutung

**msqid** : Kennzeichnet die Message Queue, deren Eigenschaften gelesen bzw. modifiziert werden sollen.

**cmd** : Was eigentlich gemacht werden soll. Die wichtigsten möglichen Werte für **cmd** sind

**IPC_STAT** : Kopiert die Eigenschaften der Message Queue in die Struktur, auf die der Pointer ***buf** zeigt.

**IPC_SET** : Schreibt die veränderbaren Elemente von **msqid_ds** in die Message Queue zurück. Dies sind **msg_qbyte**, **msg_perm.uid**, **msg_perm.gid** und die Zugriffsrechte, d.h. die neun niederwertigsten Bit von **msg_perm.mode**. Die Änderungen werden allerdings nur akzeptiert, wenn die effektive User ID des aufrufenden Prozesses gleich der des Eigentümers oder des Erzeugers der Message Queue ist

**IPC_RMID** : Entfernt die Message Queue aus dem System, sofern die effektive User ID des aufrufenden Prozesses gleich der des Eigentümers oder des Erzeugers der Message Queue ist. Alle Prozesse, die zu diesem Zeitpunkt auf die Message Queue warten, werden geweckt und die entsprechenden Systemaufrufe kehren mit einem Fehler zurück.

Es gibt noch eine Reihe weiterer, Linux-spezifischer Kommandos für **msgctl()**, die bei Bedarf in der Manpage machgeschlagen werden können.

**buf** : Zeigt auf eine **msqid_ds**-Struktur, die die Daten aus der Message Queue aufnehmen bzw. die zu schreibenden Daten enthält. Wenn das Kommando **IPC_RMID** ist. reicht hier ein NULL-Pointer.

Der Rückgabewert der Funktion **msgctl()** ist 0 im Erfolgsfall und -1 im Fehlerfall.

### Beispielprogramme

Im Folgenden sind die Listings von zwei Programmen aufgefhrt, die eine Message Queue zwischen einem Sender und zwei Empfängern aufbauen.

**msgsend.c** Richtet eine Message Queue ein und schickt Nachrichten hinein

**msgrecv.c** Liest Botschaften aus der Message Queue aus, die **msgsend** eingerichtet hat. Der Typ von Nachrichten, die der Empfänger lesen soll, wird als Kommandozeilenparameter übergeben.

```
/*************************************************************
 * msgsend.c:                                                *
 * DEMO Programm zur Verwendung von Message Queues           *
 * (Senderteil)                                              *
 *                           Peter Vaeterlein, 2004-11-20    *
 *                                                           *
 *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>

#define IPCKEY 4711
#define MYMODE 0644
#define MAXLEN 255

int msqid;
int rval;
char dummy;

struct {
         long msg_type;
         char msg_text [MAXLEN];
       } msg;

int main()
{
  printf( "\nMessage Queue Sender:\n\n" );
  printf( "Beenden durch Eingabe von Empfaenger = 0 ODER
      Nachricht = '.'\n" );

  // --- Message Queue einrichten, Abbruch, wenn die Queue
      bereits existiert

  if (( msqid = msgget( IPCKEY, IPC_CREAT|IPC_EXCL|MYMODE )) < 0
      ) {
    perror( "Fehler bei der Einrichtung der Message Queue" );
    exit( 1 );
  }

  while ( strcmp( msg.msg_text, ".\n" )) {
```

```
                                                    // --- Festlegung
                                                        des Empfaengers
    printf( "\n- Empfaenger: " );
    scanf( "%ld", &msg.msg_type );

    // --- Wenn entweder als Empfaenger 0 oder ein Punkt als
    //     Nachricht eingegeben wurde ...

    if (( msg.msg_type == 0 ) || ( strcmp( ".\n", msg.msg_text )
        == 0 )) {
    // --- Abbruch-Befehl an Empfaenger 1 schicken
      msg.msg_type = 1;
      strcpy (msg.msg_text, ".\n" );
      if ( rval = msgsnd( msqid,
                  &msg, (size_t) strlen( msg.msg_text ),
                  NULL )) {
        perror( "Fehler beim Fuellen der Message Queue 1 (Ende)"
            );
        exit( 2 );
      }

      // --- Abbruch-Befehl an Empfaenger 2 schicken
      msg.msg_type = 2;
      if ( rval = msgsnd( msqid, &msg,
                          (size_t) strlen( msg.msg_text ),
                          NULL )) {
        perror( "Fehler beim Fuellen der Message Queue 2 (Ende)"
            );
        exit( 2 );
      }

    // --- anderenfalls ( Empfaenger != 0 und Nachricht != '.' )
        ...
    } else {
      printf ( "- Nachricht : " );
      dummy = fgetc ( stdin ); // --- uebriggebliebenes \n
          entfernen
      fgets ( msg.msg_text, MAXLEN, stdin );
                                // --- Nachricht verschicken
      if ( rval = msgsnd( msqid, &msg,
                          (size_t) strlen( msg.msg_text ),
                          NULL )) {
        perror( "Fehler beim Fuellen der Message Queue 1/2" );
        exit( 2 );
      }
    }
  }

  // --- Message Queue loeschen
  sleep ( 1 );
  if ( rval = msgctl ( msqid, IPC_RMID, NULL ) ) {
    perror( "Fehler bei der Aufloesung der Message Queue" );
    exit( 3 );
  }
}
```

```
/*************************************************************
 * msgrecv.c                                                 *
 * DEMO Programm zur Verwendung von Message Queues           *
 * (Empfaengerteil)                                          *
 *                          Peter Vaeterlein, 2004-11-20     *
 *                                                           *
 *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>

#define IPCKEY 4711
#define MAXLEN 255

int msqid;
int myid;
int rval;

struct {
         long msg_type;
         char msg_text [MAXLEN];
       } msg;

int main( int argc, char **argv )
{
  // --- Fehlerbehandlung, wenn kein Empfaenger als
      Kommandozeilenargument
  // angegeben wurde
  if ( argc != 2 ) {
    printf( "\nBitte die Nummer des Empfaengers (1 oder 2) als
        Argument\
auf der\n" );
    printf( "Kommandozeile angeben!\n" );
    exit( 4 );
  }

  myid = atoi ( argv[1] );
  printf( "\nMessage Queue Receiver %d:\n\n", myid );

  // --- Message Queue einrichten (muss vorher existieren)
  if (( msqid = msgget( IPCKEY, 0 )) < 0 ){
    perror( "Fehler bei der Einrichtung der Message Queue" );
    exit( 1 );
  }

  // --- Endlosschleife zum Empfang von Nachrichten
  while ( 1 ){
    // --- Eventuell noch vorhandene Nachrichten loeschen
    memset( msg.msg_text, '\0', MAXLEN );

    // --- Nachrichten auslesen, die an diesen Empfaenger
        gerichtet sind
    if ( msgrcv( msqid, &msg, MAXLEN, (long) myid, NULL ) == -1
        ) {
      perror( "Fehler beim Auslesen der Message Queue" );
      exit( 2 );
    }
```

```
    // --- Schleife abbrechen, wenn Nachricht = '.' ist
    if ( strcmp( ".\n", msg.msg_text ) == 0 ) {
      break;
    }

    // --- Nachricht ausgeben
    printf( " - Nachricht: %s", msg.msg_text );
  }
}
```

**Übung:** Übersetzen Sie die beiden Programme. Starten Sie in einem Terminalfenster den Sender **msgsend** und in zwei weiteren Terminalfenstern zwei Empfänger **msgrecv**. Sie müssen die beiden natürlich mit dem jeweiligen Typ (also 1 oder 2) starten. Nun können Sie Nachrichten vom Sender an einen der beiden Empfänger schicken. Untersuchen Sie die Funktion der Programme auch mit *strace*.
Sie können das Starten der Programme komfortabler gestalten, wenn Sie ein kleines Shellskript schreiben, das die drei Terminalfenster in drei Ecken des Bildschirms und die jeweiligen Programme in den Fenstern startet.

**Exkurs: Erkundung von IPC-Objekten**
Die wichtigsten Eigenschaften der auf einem Rechner aktiven Message Queues (wie auch die der weiter unten besprochenen Shared Memory Segmente und Semaphorenstapel) lassen sich mit dem Kommando **ipcs** ansehen.
Mit dem Kommando **ipcrm** lassen sich überflssig gewordene IPC-Objekte löschen (zum Beispiel, wenn die Anwendung, die die Objekte erzeugt hat, abgebrochen wurde, bevor die Objekte wieder ordnungsgemäß entfernt wurden). Die Objekte lassen sich entweder durch den zugehörigen Key oder die entsprechenden Deskriptoren (**msqid**, **shmid** bzw. **semid**) identifizieren. Die genaue Syntax kann der Manpage von von **ipcrm** entnommen werden.

## 3.4 Shared Memory

Das Prinzip von *Shared Memory* ist denkbar einfach: Ein und dieselbe Seite des physikalischen Speichers, ein so genanntes *Shared Memory Segment* wird

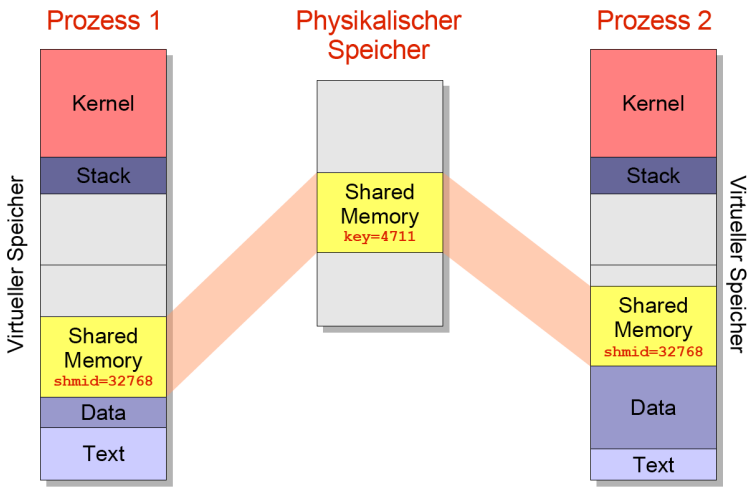


**Abbildung 7:** *Schematische Darstellung der Funktionsweise des Shared Memory Mechanismus*

in den virtuellen Speicher mehrerer Prozesse eingeblendet. Ein Prozess kann auf diese Speicherseite schreiben, während ein anderer Prozess von derselben Speicherseite lesen kann. Für die Datenbertragung sind also keine weiteren Kopiervorgänge nötig. Deshalb ist *Shared Memory* die schnellste Form der Interprozesskommunikation. Allerdings ist die transferierbare Datenmenge durch die Größe des zur Verfügung stehenden Speichers begrenzt.

Die größte Schwäche des *Shared Memory* Mechanismus ist, dass der Zugriff unterschiedlicher Prozesse auf das Shared Memory Segment nicht synchronisiert ist. Es kann daher zu Inkonsistenzen zwischen den geschriebenen und den gelesenen Daten kommen. Das zu verhindern ist Sache der Anwendungsprogrammierer(innen).

Wie die Message Queues sind auch die Shared Memory Segmente Objekte, die sich außerhalb von allen Prozessen eines Rechners befinden. Um Sie nutzen zu können, sind daher zwei Schritte notwendig:

- Das Shared Memory Segment muss im physikalischen Speicher eingerichtet werden (Systemaufruf **`shmget()`**)
- Das Shared Memory Segment muss in den virtuellen Speicher der beteiligten Prozesse eingeblendet werden. (**`shmat()`**)

Danach kann der gemeinsam genutzte Speicherbereich über eine Pointervaria-

ble genutzt werden. Soll die gemeinsame Nutzung beendet werden, muss

- Die Ankopplung des Shared Memory Segmentes an den Prozess gelöst werden (**`shmdt()`**)
- Schließlich muss das Shared Memory Segment aus dem physikalischen Speicher entfernt werden (**`shmctl()`**)

### Einrichten von Shared Memory im physikalischen Speicher

Der Systemaufruf

```
int shmget(key_t key,int size,int shmflg);
```

richtet bei Bedarf ein Shared Memory Segment ein, das wie schon die Message Queues durch einen systemweit eindeutigen **`key`** gekennzeichnet wird. Dieser Key kann entweder eine beliebige positive Zahl sein oder mit der vordefinierten Konstante **`IPC_PRIVATE`** belegt sein. In diesem Fall ist das Shared Memory Segment nur für den aufrufenden Prozess und seine Kinder zugänglich. Der zweite Parameter (**`size`**) der Funktion gibt die Größe des zu reservierenden Speicherbereichs an. In der Regel verwendet man eine Struktur oder ein Array zur Übertragung von Daten über Shared Memory. In diesem Fall würde man hier die Größe der verwendeten Datenstruktur angeben. Der dritte Parameter (**`shmflags`**) enthält schließlich Flags, die das Verhalten der **`shmget()`**-Funktion steuern. Mögliche Werte für diese Flags sind

**`IPC_CREAT`** : Erzeuge das Shared Memory Segment zu dem angegebenen Key, wenn es nicht bereits existiert. Sonst mache das existierende Segment für den aktuellen Prozess zugänglich.

**`IPC_EXCL`** : Wenn dieses Flag gemeinsam mit **`IPC_CREAT`** gesetzt ist, kehrt die Funktion mit einem Fehler zurück, wenn ein Shared Memory Segment mit dem angegebenen Key bereits existiert

Ebenfalls zu den **`shmflags`** gehören die Zugriffsrechte auf den gemeinsam genutzten Speicher. Wie bei UNIX üblich werden diese Rechte als dreistellige

Oktalzahl angegeben, die mit den anderen Flags bei Bedarf bitweise verodert werden kann.

Der Rückgabewert der Funktion **shmget()** ist ein Deskriptor (**shmid**), mit dem der Prozess Bezug auf ein bestimmtes Shared Memory Segment nehmen kann.

### Ankoppeln des SHM Segments an den Prozess

Mit dem Systemaufruf

```
void *shmat(int shmid, const void *shmaddr,
            int shmflg);
```

wird ein Shared Memory Segment mit einer Pointervariablen verknüpft und so für den aufrufenden Prozess zugänglich gemacht. Der erste Parameter (**shmid**) ist der Deskriptor des zu verknüpfenden Shared Memory Segmentes, so wie er von **shmget()** geliefert wurde. Der zweite Parameter (**shmaddr**) gibt die Möglichkeit, die Lage der neuen Adresse im virtuellen Speicher des aufrufenden Prozesses zu beeinflussen. Meist steht hier einfach **NULL**, was dem Betriebssystem freie Hand lässt, den Speicherbereich zu positionieren. Der dritte Parameter enthält wieder Flags, die das Verhalten der **shmat()**-Funktion steuern. Das wichtigste Flag ist **SHM_RDONLY**, das den Speicherbereich für Schreibzugriffe sperrt. Weitere Flags[7] sind in der Manpage von **shmat()** dokumentiert.

Der Rückgabewert von **shmat()** ist die Anfangsadresse des Shared Memory Segmentes im virtuellen Speicher.

### Nutzen des Shared Memory

Die Nutzung von Shared Memory ist so einfach wie die Nutzung einer dynamisch allokierten Variablen. Zunächst wird eine Pointervariable deklariert, die

[7] Diese Flags sind eigentlich nur wichtig, wenn man selbst den Ort des gemeinsam genutzten Speicherbereichs im virtuellen Speicher wählt (wovon in der Regel nur abgeraten werden kann).

dann mit dem Rückgabewert von **shmat()** gefüllt wird. Weist man dieser Variablen (nicht der Adresse sondern dem Inhalt) Daten zu, so werden diese automatisch im Shared Memory gespeichert.

Ein anderer oder derselbe Prozess kann auf die Daten genau so einfach zugreifen, indem er auf den Inhalt der Zeigervariablen zugreift, die auf das Shared Memory Segment zeigt. Die Anwendungen müssen lediglich dafür sorgen, dass der Zugriff der verschiedenen Prozesse synchronisiert wird, Das bedeutet, dass die zu übertragenden Daten auch tatsächlich im Shared Memory liegen, bevor der lesende Prozess darauf zugreift. Weiter unten wird an Beispielen gezeigt, wie eine solche Synchronisation realisiert werden könnte.

### Abkoppeln des Shared Memory

Wird ein Shared Memory Segment nicht mehr benötigt, muss man es zunächst von allen damit verbundenen Prozessen abkoppeln. Dazu dient der Systemaufruf

```
int shmdt(const void *shmaddr);
```

Der Parameter **shmaddr** ist die Adresse, die **shmat()** zurückgegeben hatte. Der Rückgabewert ist 0 bei Erfolg und -1 im Fehlerfall.

Die Funktion **shmdt()** ist dem Systemaufruf **free()** vergleichbar, mit dem man Speicher, den man zuvor mit **malloc()** einer Variablen zugewiesen hatte, wieder frei gibt.

### Kontrollieren des Shared Memory

Wie schon bei den Message Queues ist eine unmittelbare Einflussnahme auf Shared Memory Segmente deshalb nicht möglich, weil die nicht Teil der Prozessumgebung sind. Hier wie dort wird das Problem dadurch gelöst, dass man die Parameter des Shared Memory Segmentes in eine Struktur kopiert, die dann im virtuellen Speicher des aufrufenden Prozesses gelesen und, wo möglich, verändert werden kann. Diese Datensruktur mit der Bezeichnung **shmid_ds** ist der oben diskutierten Struktur **msqid_ds** sehr ähnlich:

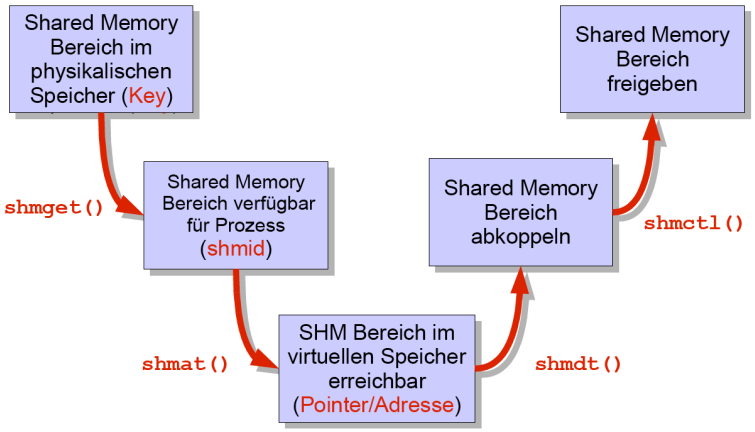


**Abbildung 8:** *Schematische Darstellung des Lebenszyklus eines Shared Memory Segmentes*

```
struct shmid_ds {
     struct ipc_perm shm_perm;    /* Ownership and permissions */
     size_t          shm_segsz;   /* Size of segment (bytes) */
     time_t          shm_atime;   /* Last attach time */
     time_t          shm_dtime;   /* Last detach time */
     time_t          shm_ctime;   /* Last change time */
     pid_t           shm_cpid;    /* PID of creator */
     pid_t           shm_lpid;    /* PID of last shmat()/shmdt()
         */
     shmatt_t        shm_nattch;  /* No. of current attaches */
};
```

Die einzigen neuen Elemente dieser Struktur sind **shm_dtime** (der Zeitpunkt der letzten Abkopplung von einem Prozess) und **shm_nattach** (die Zahl der Kopplungen an Prozesse, eine Art Linkcount). Die Stuktur **ipc_perm** ist dieselbe wie bei der Struktur **msqid_ds**.

Um die Struktur **shmid_ds** mit den Eigenschaften einer Shared Memory Segmentes zu füllen, oder um diese Eigenschaften zu modifizieren, dient der Systemaufruf

```
int shmctl(int shmid, int cmd,
                      struct shmid_ds *buf);
```

Die Parameter sind prinzipiell dieselben wie bei **msgctl()**, nur dass **shmid** jetzt den Deskriptor eines Shared Memory Segmentes bezeichnet und **buf** auf

eine Struktur des Typs **shmid_ds** zeigt. Die möglichen Werte von **cmd** und deren Bedeutung sind dieselben wie bei **msgctl()** (s. Seite 24).

### Beispielprogramm

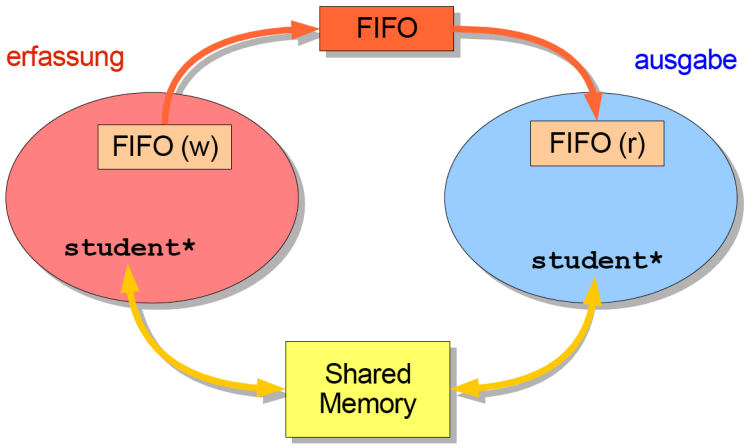


**Abbildung 9:** *Schematische Darstellung der Funktionsweise der Studierendenverwaltung via Shared Memory*

Die Funktionsweise von Shared Memory wird im Folgenden anhand eines einfachen Beispiels illustriert. Zwei Programme stellen eine stark vereinfachte Version einer Studierendenverwaltung dar. Das eine Programm (**erfassung**) dient der Eingabe von Name, Vorname und Matrikelnummer von Studierenden. Diese Daten werden in einem Shared Memory Segment gespeichert und dort von dem zweiten Programm (**ausgabe**) gelesen und der Einfachheit halber einfach auf der Konsole ausgegeben.

Die Synchronisation zwischen beiden Programmen erfolgt über eine Named Pipe mit dem Namen **FIFO**. Das Programm **erfassung** schreibt ein **D** in die Pipe, wenn neue Daten anliegen und ein **E**, wenn das Programm beendet werden soll. Das Programm **ausgabe** blockiert so lange bei dem Versuch, aus der Pipe etwas zu lesen, bis dort ein Zeichen ankommt. Handelt es sich um ein **D**, so liest das Programm das Shared Memory aus, ist es ein **E**, wird das Programm beendet.

Da die Named Pipe vom Programm **ausgabe** eingerichtet wird, muss dieses Programm immer zuerst gestartet werden.

Da die Struktur, die im Shared Memory abgelegt werden soll, in beiden Programmen gleich sein muss, bietet es sich an, diese Struktur und gemeinsam genutzte Konstanten in einer Headerdatei zu definieren.

```
/*************************************************************
 * studenten.h                                               *
 *************************************************************/

#define SHMKEY 4711
#define PIPMODE 0600

typedef struct {
                  char name [20];
                  char vorname [20];
                  char matnr [6];
                } datensatz;
```

Das Eingabeprogramm **erfassung**

- setzt voraus, dass das Programm **ausgabe** bereits läuft
- fordert das Shared Memory Segment an und koppelt es an den Pointer ***student**
- öffnet die Pipe **FIFO** zum Schreiben
- fordert zur Eingabe eines neuen Datensatzes auf der Konsole auf
- schickt, wenn alle Felder ausgefüllt wurden, ein **D** in die Pipe, was das Programm **ausgabe** dazu veranlasst, die Daten im SHM-Segment zu lesen und zu verarbeiten
- schickt, wenn das Programm beendet werden soll, ein **E** in die Pipe, was das Programm **ausgabe** dazu veranlasst, sich ebenfalls zu beenden.

```
/*************************************************************
 * erfassung.c                                               *
 * Sender fuer eine Kommunikation via Shared Memory          *
 *                              Peter Vaeterlein, 2006-11-25 *
 *************************************************************/

#include <sys/types.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
```

```c
#include "studenten.h"

main()
{
  datensatz *student; /* Pointer, der spaeter auf das SHM zeigt
      */
  int gelesen;
  int fd;
  int shmid;

  /* FIFO zum Schreiben oeffnen */
  if (( fd = open ( "FIFO", O_WRONLY, 0 )) < 0 ) {
    perror ( "Oeffnen des FIFO" );
    exit (1);
  }

  /* SHM anfordern */
  if (( shmid = shmget ( SHMKEY, sizeof ( datensatz ), 0 )) < 0
      ) {
    perror ( "Anfordern von SHM" );
    exit (2);
  }

  /* SHM ankoppeln */
  if (( student = ( datensatz * ) shmat ( shmid, NULL, 0 )) <
    (datensatz *) NULL ) {
    perror ( "Ankoppeln des SHM" );
    exit (3);
  }

  /* Endlosschleife zur Dateneingabe */
  while ( 1 ) {

    /* Neuen Datensatz abfragen und in das SHM eintragen */
    printf ( "\nDatensatz fuer neue(n) Student(in):\n" );
    printf ( "Name            : " );
    scanf ( "%s", student->name );
    if ( student->name [0] == '.' )
      break;
    printf ( "Vorname         : " );
    scanf ( "%s", student->vorname );
    printf ( "Matrikelnummer : " );
    scanf ( "%s", student->matnr );

    /* In FIFO schreiben, um neue Daten anzukuendigen */
    write ( fd, "D", sizeof ( "D" ) );
  }

  /* SHM abkoppeln */
  shmdt ( student );

  /* Ende-Signal in die Pipe schreiben */
  write ( fd, "E", sizeof ( "E" ));

  /* FIFO schliessen */
  close ( fd );
}
```

Das Ausgabeprogramm **ausgabe**

- erzeugt Named Pipe und Shared Memory Segment
- koppelt das Shared Memory Segment an den Pointer ***student**
- öffnet die Named Pipe zum Lesen
- versucht, aus der Pipe zu lesen und blockiert so lange, bis ein Zeichen ankommt
  - Handelt es sich um ein **D**, wird das Shared Memory gelesen und die Daten ausgewertet
  - Handelt es sich um ein **E**, wird das Programm beendet und Shared Memory Segment und Named Pipe werden gelöscht

```
/*************************************************************
 * ausgabe.c                                                 *
 * Empfaenger fuer eine Kommunikation via Shared Memory      *
 *                              Peter Vaeterlein, 2006-11-25 *
 *************************************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include "studenten.h"

main()
{
  char msg [16]; /* Puffer fuer FIFO */
  int gelesen;
  datensatz *student; /* Pointer, der spaeter auf das SHM zeigt
      */
  int fd;
  int shmid;

  /* FIFO erzeugen */
  if (( mknod ( "FIFO", S_IFIFO | PIPMODE, 0 )) < 0 ) {
    perror ( "Erzeugen des FIFO" );
    exit (1);
  }

  /* FIFO zum Lesen oeffnen */
  if (( fd = open ( "FIFO", O_RDONLY, 0 )) < 0 ) {
    perror ( "Oeffnen des FIFO zum Lesen" );
    exit (2);
  }

  /* SHM anfordern */
```

```
  if (( shmid = shmget ( SHMKEY, sizeof ( datensatz ),
                         IPC_CREAT | PIPMODE )) < 0 ) {
    perror ( "Anfordern von SHM" );
    exit (3);
  }

  /* SHM ankoppeln (im virtuellen Speicher verankern */
  if (( student =
        ( datensatz * ) shmat ( shmid, NULL, 0 )) < (datensatz
            *) NULL ) {
    perror ( "Ankoppeln von SHM");
    exit (4);
  }

  /* Endlosschleife zum Lesen von Daten */
  while ( 1 ) {
    /* Auf Zeichen aus dem FIFO warten (Zeichen, dass neue Daten
        vorhanden
       sind */
    gelesen = read ( fd, msg, 256 );

    if ( msg [0] == 'D' ) {
      /* Wenn das erste Zeichen in der Pipe ein 'D'
         ist, liegen neue Daten an ... */
      printf ( "Neuer Student: %s %s (Matr.Nr.: %s )\n",
               student->vorname,
               student->name,
               student->matnr );
    } else { /* Sonst verabschiedet sich der Erfassungs-
                Prozess */
      printf ( "Keine weiteren Datensaetze zu erwarten\n" );
      break;
    }
  }

  shmdt ( student ); /* SHM abkoppeln */
  close ( fd ); /* FIFO schliessen */
  unlink ( "FIFO" ); /* FIFO loeschen */
}
```

# 4 Synchronisation von Prozessen

## 4.1 Signale

Wurden bereits in der Vorlesung behandelt.

## 4.2 Semaphore

Semaphore sind Variable, die nicht innerhalb eines Prozesses sondern systemweit gültig sind. Insofern sind sie konzeptionell eng verwandt mit dem Shared Memory. Zusätzlich gehören zu einem Semaphor aber immer noch zwei Funktionen: **up()** und **down()**.

Unter Linux werden nicht einzelne Semaphore verwaltet sondern ganze Stapel (oder Arrays) von Semaphoren. Wie schon bei Message Queues und Shared Memory Segmenten muss man die Eigenschaften dieser Objekte in eine Struktur kopieren, um von einem UNIX-Prozess darauf zugreifen zu können. Diese Struktur **semid_ds** ist eng verwandt mit **msqid_ds** und **shmid_ds**:

```
struct semid_ds {
     struct ipc_perm sem_perm;  /* Ownership and permissions
     time_t          sem_otime; /* Last semop time */
     time_t          sem_ctime; /* Last change time */
     unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

### Semaphorensatz anfordern

Der Systemaufruf

```
int semget(key_t key, int nsems, int semflg);
```

hat große Ähnlichkeit mit den analogen Funktionen **msgget()** und **shmget**. Der erste Parameter ist wieder ein systemweit eindeutiger **key**, der den Semaphorensatz kennzeichnet. Wie bei den anderen IPC Mechanismen kann man

die vordefinierte Konstante **IPC_PRIVATE** als Key angeben. In diesem Fall ist der Semaphorensatz nur dem aufrufenden Prozess und seinen Kindern zugänglich. Der zweite Parameter (**nsems**) gibt die Zahl der Semaphoren im Semaphorensatz an. Der dritte Parameter (**semflg**) enthält wieder die Flags zur Steuerung des Verhaltens des Semaphorensatzes. Die wichtigsten Flags sind schon von Message Queues und Shared Memory her bekannt: **IPC_CREAT** und **IPC_EXCL**. Und wieder gehören die Zugriffsrechte zu den Flags und sollten nicht vergessen werden, da sonst der Semaphorenstapel nicht mehr zugänglich ist und sehr schwer wieder los zu werden ist.

Der Rückgabewert von **semget()** ist, man ahnt es schon, ein Deskriptor, der den Semaphorenstapel vom aufrufenden Prozess aus zugänglich macht.

### Semaphor-Operationen

In der ursprünglichen Form gab es zwei Funktionen, um einen Semaphor zu manipulieren: **up()** und **down()**. Bei der Implementierung auf POSIX konformen UNIX-Systemen und insbesondere unter Linux können die Semaphorenstapel als Ganzes manipuliert werden. Das heißt, es können auf allen Semaphoren gleichzeitig Operationen ausgeführt werden. Diese Operationen sind *atomar*, können also nicht durch Prozesswechsel auseinander gerissen werden.

Zur Manipulation von Semaphoren dient der Systemaufruf

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

Der erste Parameter (**semid**) ist der Deskriptor, der den zu mainpulierenden Semaphorenstapel kennzeichnet. Der zweite Parameter ist ein Zeiger, der auf ein Array von Strukturen (**sops**) zeigt, das die einzelnen Semaphor-Operationen enthält. Der dritte Parameter (**nsops**) gibt an, wie viele Elemente das Array **sops** enthält.

Die Elemente des Arrays **sops** sind Strukturen vom Typ **sembuf**:

```
struct sembuf {
                unsigned short sem_num;  /* semaphore number */
```

```
    short           sem_op;    /* semaphore operation
       */
    short           sem_flg;   /* operation flags */
```

Die Elemente haben folgende Bedeutungen

**sem_num** : Die Nummer des zu manipulierenden Semaphors (von 0 bis **nsems** - 1)

**sem_op** : Die auszuführende Operation. Ist **sem_op** ungleich Null, wird diese Zahl zu dem aktuellen Wert des Semaphors addiert. Je nach Vorzeichen von **sem_op** wird der Wert des Semaphors also erhöht oder erniedrigt. Würde der Wert des Semaphors nach der Operation kleiner als Null, blockiert der aufrufende Prozess so lange, bis ein anderer Prozess den Semaphor durch eine entsprechende Erhöhung des Semaphor-Wertes diesen wieder „ins Plus“ bringt. Ist **sem_op** gleich Null, blockiert der aufrufende Prozess so lange, bis der entsprechende Semaphor gleich Null ist.

**sem_flg** : enthält Flags zur Beeinflussung des Verhaltens von **semop()**. Dabei werden zwei Flags beachtet:

**IPC_NOWAIT** : Verhindert ein Blockieren des Prozesses und kehrt stattdessen mit einem Fehler zurück

**SEM_UNDO** : Weist das Betriebssystem an, die Operation rückgängig zu machen, wenn der aufrufende Prozess beendet wird.

Beide Flags können, wie üblich, bitweise verodert werden.

### Eigenschaften des Semaphoresatzes beeinflussen

Um die Struktur **semid_ds** mit Daten zu füllen oder geänderte Daten in den Semaphorensatz zurück zu schreiben, dient der Systemaufruf

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

Die Parameter haben folgende Bedeutungen:

**semid** : Deskriptor des Semaphorenstapels

**semnum** : Nummer eines Semaphors innerhalb des Stapels

**cmd** : Kommando, das ausgeführt werden soll

**IPC_STAT** : Lesen der Struktur **semid_ds**

**IPC_SET** : Schreiben der Struktur **semid_ds**

**GETVAL/SETVAL** : Abfragen/Setzen des Wertes von Semaphor **semnum**

**GETALL/SETALL** : Abfragen/Setzen der Werte aller Semaphoren

**arg** : Dieser Parameter ist als Union definiert, weil an dieser Stelle je nach dem Wert von **cmd** eine unterschiedliche Datenstruktur benötigt wird:

```
union semun {
     int              val;    /* Value for SETVAL */
     struct semid_ds *buf;    /* Buffer for IPC_STAT,
         IPC_SET */
     unsigned short  *array;  /* Array for GETALL, SETALL */
};
```

### Beispielprogramme

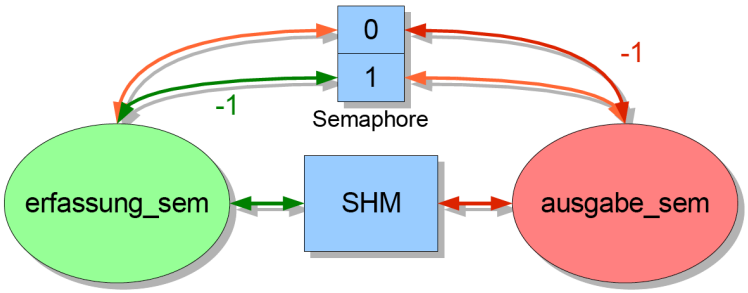


**Abbildung 10:** *Schematische Darstellung der Funktionsweise der Studierendenverwaltung via Shared Memory mit Synchronisation durch Semaphore. Dargestellt ist der Anfangszustand der Semaphore unmittelbar nach dem Start der Programme.*

Als Beispiel für die Anwendung von Semaphoren wird noch einmal das Beispiel der Studierendenverwaltung via Shared Memory aufgegriffen. Nur wird jetzt die Synchronisierung der Zugriffe auf den Shared Memory Bereich mit Semaphoren geregelt (vgl. Abbildung 10).

In diesem Fall werden zwei Semaphore benutzt: ein „Schreib-Semaphor“ und ein „Lese-Semaphor“. dadurch lassen sich der schreibende Prozess (**erfassung_sem**) und der lesende Prozess (**ausgabe_sem**) gezielt steuern. Die Synchronisation läuft dann so ab:

- Zu Beginn ist der Schreib-Semaphor 1 und der Lese-Semaphor 0
- Am Anfang der Hauptschleife versucht jeder Prozess, seinen eigenen Semaphor zu dekrementieren. Beim Schreib-Semaphor geht das problemlos. Beim Lese-Semaphor blockiert der Prozess, weil der vorher schon 0 war.
- Nach der Dateneingabe inkrementiert der schreibende Prozess den Lese-Semaphor und gibt dadurch den lesenden Prozess frei. Der führt nun die geplante Dekrementierung des Lese-Semaphors durch, der damit wieder den Wert 0 hat. Auf diese Weise ist gewährleistet, dass der lesende Prozess jeden Datensatz nur einmal liest. Denn sobald der Prozess wieder am Anfang der Hauptschleife ist, versucht er wieder, seinen eigenen Semaphor zu dekrementieren und blockiert dabei (s. oben).
- Der schreibende Prozess durchläuft die Hauptschleife zum zweiten Mal und blockiert bei dem Versuch, seinen eigenen Semaphor zu dekrementieren.
- Der lesende Prozess liest das Shared Memory aus und verarbeitet die Daten. Danach inkrementeiert er den Schreib-Semaphor und gibt dadurch den schreibenden Prozess wieder frei. Der dekrementiert nun seinen eigenen Semaphor, wie geplant. Dadurch hat dieser wieder den Wert 0. Damit ist gewhrleistet, dass ein neuer Datensatz erst dann in das Shared Memory geschrieben wird, wenn der alte gelesen wurde.

Die Semaphore funktionieren also hnlich wie die Drehkreuze im Freibad oder in der U-Bahn, die immer nur eine Person durchlassen und dann wieder blockieren, bis man seine Münze oder Karte eingegeben hat.

```
/*************************************************************
 * studenten_sem.h                                           *
 *************************************************************/

#define SHMKEY 4711
#define SEMKEY 1492
#define MYMODE 0600

typedef struct datensatz {
```

```
        char name [20];
        char vorname [20];
        char matnr [6];
} datensatz;
```

```
/************************************************************
 * erfassung_sem.c                                          *
 * Sender fuer eine Kommunikation via Shared Memory und     *
 * Synchronisation des Zugriffs mit Semaphoren.             *
 *                              Peter Vaeterlein 2006-11-25 *
 ************************************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include "studenten_sem.h"

main()
{
  datensatz *student; /* Pointer, der spaeter auf das SHM zeigt
      */
  int dum;

  int shmid;
  int semid;
  struct sembuf sops;
  struct shmid_ds shmbuf;
  union semun {
          int val;
          struct semid_ds *buf;
          ushort *array;
  } dumun;                        /* Dummy-Union fuer semctl */

                                  /* Semaphorensatz erzeugen */

  if (( semid = semget ( SEMKEY, 2, IPC_CREAT | MYMODE )) < 0 ) {
          perror ( "Erzeugen des Semaphorensatzes" );
          exit (1);
  }
                                   /* Semaphore initialisieren:
                                      Semaphor #0 (Lesen) = 0
                                      Semaphor #1 (Schreiben) = 1
                                          */
  sops.sem_flg = 0;
  dumun.val = 0;
  if (( dum = semctl ( semid, 0, SETVAL, dumun )) < 0 )
  {
    perror ( "Initialisierung des Lese-Semaphors" );
  }
  dumun.val = 1;
  if (( dum = semctl ( semid, 1, SETVAL, dumun )) < 0 )
  {
```

```
    perror ( "Initialisierung des Schreib-Semaphors" );
  }

                                /* SHM anfordern */
  if (( shmid = shmget ( SHMKEY, sizeof ( datensatz ),
                         IPC_CREAT | MYMODE )) < 0 )
  {
    perror ( "Anfordern von SHM" );
    exit (3);
  }

                /* SHM ankoppeln (im virtuellen Speicher
                                   verankern */

  if (( student = ( datensatz * ) shmat ( shmid, NULL, 0 ))
       < (datensatz *) NULL ) {
    perror ( "Ankoppeln des SHM" );
    exit (3);
  }

  while ( 1 )
    {           /* Schreib-Semaphor dekrementieren, um
                                   zu testen, ob Schreibzugriffe
                                       erlaubt
                                   sind */
      sops.sem_num = 1;
      sops.sem_op = -1;
      if (( dum = semop ( semid, &sops, 1 )) < 0 )
      {
        perror ( "Dekrementieren des Schreib-Semaphors" );
      }
      printf ( "\nDatensatz fuer neue(n) Student(in):\n" );
      printf ( "Name           : " );
      scanf ( "%s", student->name );

                /* Nach weiteren Daten nur fragen, wenn
                   das erste Zeichen kein "." war       */

      if ( student->name [0] != '.' ) {
        printf ( "Vorname        : " );
        scanf ( "%s", student->vorname );
        printf ( "Matrikelnummer : " );
        scanf ( "%s", student->matnr );
      }
                                /* Lesezugriffe wieder zu
                                    erlauben */
      sops.sem_num = 0;
      sops.sem_op = 1;
      dum = semop ( semid, &sops, 1 );

      if ( student->name [0] == '.' ) {
        break;
      }
    }
                /* SHM abkoppeln */
  shmdt ( student );
}
```

```
/************************************************************
 * ausgabe_sem.c                                            *
 * Empfaenger fuer eine Kommunikation via Shared Memory und *
 * Synchronisation des Zugriffs mit Semaphoren.             *
 *                              Peter Vaeterlein 2006-11-25 *
 ************************************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include "studenten_sem.h"

main()
{
  datensatz *student;       /* Pointer, der spaeter auf das SHM
      zeigt */
  int dum;

  int shmid;
  int semid;                /* Semaphorensatz-ID */
  struct sembuf sops;       /* Puffer fuer Semaphorenkommandos */
  struct shmid_ds shmbuf;   /* Dummypuffer fuer shmctl */
  union semun {
          int val;
          struct semid_ds *buf;
          ushort *array;
  } dumun;                  /* Dummy-Union fuer semctl */

                            /* Semaphorensatz mit zwei
                               Semaphoren oeffnen:
                                            0: Lesen
                                            1: Schreiben */

  if (( semid = semget ( SEMKEY, 2, IPC_CREAT | MYMODE )) < 0 ) {
    perror ( "Erzeugen des Semaphorensatzes" );
    exit (1);
  }
  sops.sem_flg = 0;

                             /* SHM anfordern */

  if (( shmid = shmget ( SHMKEY, sizeof ( datensatz ),
                         IPC_CREAT | MYMODE )) < 0 ) {
    perror ( "Anfordern von SHM" );
    exit (2);
  }
                             /* SHM ankoppeln (im virtuellen
                                Speicher
                                   verankern */

  if (( student = ( datensatz * ) shmat ( shmid, NULL, 0 ))
       < (datensatz *) NULL ) {
          perror ( "Ankoppeln von SHM ");
          exit (4);
```

```
  }

  while ( 1 )
  {
                                  /* Dekrementieren des
                                      Lese-Semaphors, um
                                     zu testen, ob Lesezugriffe
                                         erlaubt sind */
    sops.sem_num = 0;
    sops.sem_op = -1;
    if (( dum = semop ( semid, &sops, 1 ) ) < 0 )
    {
      perror ( "Dekrementieren des Semaphors" );
    }
                                  /* Datenausgabe */

    if ( student->name [0] == '.' )
    break;
    printf ( "Neuer Student: %s %s (Matr.Nr.: %s )\n",
             student->vorname,
             student->name,
             student->matnr );
                                  /* Schreib-Semaphor
                                      inkrementieren, um
                                     Schreibzugriffe wieder zu
                                         erlauben */
    sops.sem_num = 1;
    sops.sem_op = 1;
    dum = semop ( semid, &sops, 1 );
    sleep ( 2 );
  }
                                  /* SHM abkoppeln */
  shmdt ( student );
                                 /* SHM loeschen */

  shmctl ( shmid, IPC_RMID, &shmbuf );

                                 /* Semaphore loeschen */

  semctl( semid, 0, IPC_RMID, dumun );
  semctl( semid, 1, IPC_RMID, dumun );
}
```

**Übung:** Übersetzen Sie die Programme und testen Sie sie so wie die erste Variante. Studieren Sie das Verhalten auch mit strace.

Überprüfen Sie mit dem Kommando **ipcs**, welche IPC Objekte zu welchem Zeitpunkt vorhanden sind.