

IPC - Interprozesskommunikation

Thomas Huth

23. Mai 2002



Zusammenfassung

Diese Proseminararbeit handelt von Kommunikation, Informationsaustausch und Synchronisation zwischen Prozessen unter Linux. Im Einzelnen geht es hierbei um die Bedeutung und Programmierschnittstellen von Pipes, Messages, Shared Memory, Semaphoren und Signalen.

Inhaltsverzeichnis

1	Einleitung	3
2	Die verschiedenen Mechanismen	4
2.1	Pipes	4
2.1.1	Normale Pipes	4
2.1.2	Named Pipes	4
2.1.3	Arbeitsweise	5
2.2	Messages	5
2.2.1	Öffnen einer Warteschlange	5
2.2.2	Senden einer Nachricht	6
2.2.3	Empfangen einer Nachricht	6
2.2.4	Verwalten einer Warteschlange	6
2.3	Shared Memory	7
2.3.1	Öffnen eines Shared-Memory-Bereichs	7
2.3.2	Anbinden eines Segments	7
2.3.3	Loslösen eines Segments	8
2.3.4	Verwalten eines Shared-Memory-Bereichs	8
2.3.5	Probleme	8
2.4	Semaphore	8
2.4.1	Öffnen einer Semaphorgruppe	9
2.4.2	Semaphor-Operationen	9
2.4.3	Verwalten einer Semaphorgruppe	10
2.5	Signale	11
2.5.1	Die verschiedenen Signale	11
2.5.2	Abfangen eines Signals	12
2.5.3	Verschicken eines Signals	12
2.5.4	Warten auf ein Signal	12
3	IPC auf der Kommandozeile	13
3.1	Named Pipes	13
3.2	Messages, Shared-Memory und Semaphore	13
3.3	Signale	13

1 Einleitung

Worum geht es bei Interprozesskommunikation (IPC)? Grob gesagt, es geht um Informationsaustausch und Synchronisation zwischen Prozessen. Ein **Prozess** ist hierbei ein ausgeführtes Programm im Speicher. D.h. zum Prozessbegriff gehört also auch noch der Ausführungszustand, wie etwa die Belegung der Programmvariablen und CPU-Register, oder wie die Position der aktuell ausgeführten Instruktion ("Program-Counter"). Diese Zustände verändern sich natürlich im Laufe der Programmausführung.

Unter Multitasking-Betriebssystemen wie Linux gibt es nun "parallele Prozesse", also Prozesse, die wirklich gleichzeitig (z.B. in Rechnern mit mehreren CPUs) oder quasi gleichzeitig (durch Time-Sharing) ablaufen. Obwohl die meisten Prozesse parallel ausgeführt werden, ohne voneinander zu wissen, existieren natürlich auch viele Situationen, in denen Prozesse untereinander Daten austauschen oder sich synchronisieren müssen. So kann es z.B. passieren, dass zwei Prozesse zum Datenaustausch einen gemeinsamen Speicherabschnitt benutzen. Wie man sich leicht vorstellen kann, dürfen aber hierbei nicht beide Prozesse gleichzeitig auf den Speicherbereich zugreifen, da es sonst leicht zu inkonsistenten Daten kommen kann. Führt ein Programm solch eine Aktion wie den Zugriff auf ein gemeinsames Speichersegment aus, dann spricht man übrigens von einem **kritischen Abschnitt**. Dies ist also eine Folge von Instruktionen in einem Programm, die nicht parallel zu einem entsprechenden kritischen Abschnitt eines anderen Prozesses ausgeführt werden dürfen. Ein modernes Multitasking-Betriebssystem sollte also folglich sowohl Mittel zur Kommunikation als auch zur Synchronisation zwischen Prozessen bereitstellen.

Welche Arten der Interprozesskommunikation bietet nun Linux (und die meisten anderen UNIX-kompatiblen Systeme)? Es gibt dazu viele Möglichkeiten, aber hier werden folgende Mechanismen genauer vorgestellt:

- Pipes
- Messages
- Shared Memory
- Semaphore
- Signale

Eine weitere recht bekannte Möglichkeit zum Datenaustausch zwischen Prozessen bieten außerdem noch die sogenannten "Sockets". Da diese jedoch schon eher zum Thema Netzwerkprogrammierung (TCP/IP) gehören, sei hier auf andere Stellen verwiesen.

Die im folgenden erwähnten Funktionen und Strukturen beziehen sich übrigens alle auf die Programmiersprache C, denn für systemnahe Programmierung unter Linux ist C nunmal die Standardprogrammiersprache.

2 Die verschiedenen Mechanismen

2.1 Pipes

Wohl jeder Linux-User hat schon mal auf der Shell etwas wie

```
cat readme.txt | more
```

einggegeben, um die Ausgabe eines Programms zur Eingabe eines anderen zu machen. Diesen Mechanismus bezeichnet man als eine Pipe ("Röhre", durch welche die Daten geleitet werden). Natürlich kann man diese Pipes auch in eigenen Programmen benutzen, nicht nur auf der Shell.

Pipes kann man sich, wie der Name schon andeutet, wirklich wie eine Röhre vorstellen: Ein Prozess füllt an einem Ende der Pipe Bytes hinein, und am anderen Ende empfängt ein anderer Prozess die gesendeten Daten. Die Pipe ist hierbei eine Einbahnstraße, d.h. auf der einen Seite kann nur hineingeschrieben und auf der anderen Seite nur gelesen werden. Die Daten werden hierbei nach dem FIFO-Prinzip gehandhabt: "First in = first out", d.h. das was zuerst hineingeschrieben wird, kann auch als erstes wieder herausgelesen werden.

Man muss zwischen zwei Arten von Pipes unterscheiden: Zum einen "normale" Pipes, die meist zwischen einem Eltern- und einem Kindprozess eingesetzt werden, und die "Named Pipes", die als FIFO-Dateien im Dateisystem erzeugt werden können.

2.1.1 Normale Pipes

Mit dem Systemaufruf

```
int pipe ( int filedes[2] )
```

kann man in seinem Programm eine "anonyme" Pipe anlegen. In dem als Parameter übergebenen Feld erhält man dann zwei Dateideskriptoren, *filedes[0]* kann zum Lesen aus der Pipe und *filedes[1]* zum Schreiben in die Pipe benutzt werden. Meist wird man nach dem Aufruf von `pipe()` irgendwann den aktuellen Prozess mittels `fork()` splitten, um sich dann zwischen Eltern- und Kindprozess über die Pipe austauschen zu können.

Man kann übrigens mit der Funktion `dup2()` diese Dateideskriptoren auch auf die Standardeingabe bzw. -ausgabe umbiegen, so dass man dann z.B. die Standardausgabe eines Kindprozesses erhalten und weiterverwerten kann.

2.1.2 Named Pipes

Während die normale Pipe nur zwischen Eltern- und Kindprozessen Sinn macht, da nur sie mittels der Dateideskriptoren auf die Pipe zugreifen können, taucht eine "Named Pipe" wie eine normale Datei im Dateisystem auf und kann somit von beliebigen Prozessen benutzt werden. Eine solche FIFO-Datei kann mit folgenden Funktionen erstellt werden:

```
int mkfifo ( const char *name, mode_t mode )  
oder  
int mknod ( const char *name, mode_t mode, dev_t dev )
```

Die Parameter der Funktionen bedeuten hierbei folgendes: *name* ist der Name der FIFO-Datei, die erzeugt werden soll. In *mode* werden jeweils die Dateizugriffrechte kodiert (z.B. 0640 für

Benutzer-RW und Gruppen-RO). Bei `mknod()` muss außerdem in Mode noch kodiert werden, dass es sich um eine Pipe handelt, was durch bitweiser Oder-Verknüpfung der Zugriffsrechte mit der Konstanten `S_IFIFO` geschieht. Der Parameter *dev* bei `mknod()` ist für Pipes unwichtig und kann auf 0 gesetzt werden.

Die Pipe-Dateien lassen sich dann einfach wie gewöhnliche Dateien mit den Systemaufrufen `open()` öffnen und später mit `close()` wieder schließen.

2.1.3 Arbeitsweise

Hat man also entweder eine normale Pipe oder eine Named-Pipe mit einer der oben genannten Möglichkeiten zum Lesen bzw. Schreiben geöffnet, kann wie bei einer normalen Datei mit den Systemaufrufen `read()` bzw. `write()` von der Pipe gelesen bzw. in die Pipe geschrieben werden.

Zu beachten ist noch, dass ein Prozess blockiert wird, wenn aus einer leeren Pipe gelesen werden soll oder kein Schreiberprozess auf der anderen Seite vorhanden ist. Ein schreibender Prozess wiederum wird blockiert, wenn die Pipe voll ist (in dem Pipe-Buffer steht nur eine begrenzte Anzahl an Bytes zur Verfügung) oder wenn es auf der anderen Seite der Pipe kein lesenden Prozess gibt. Die Blockierung lässt sich bei Named-Pipes jedoch auch umgehen, wenn bei `open()` das Flag `O_NONBLOCK` (oder `O_NDELAY`) benutzt wird.

2.2 Messages

Messages sind kleine Nachrichten variabler Länge, die jeweils aus einem Typ zur Identifikation und einer Folge aus Bytes als Nachrichtentext bestehen. Dabei wird der Austausch der Botschaften vom Kernel als Warteschlangen realisiert, d.h. sind mehrere Nachrichten in einer Warteschlange, kann die erste Nachricht, die von einem Prozess abgeschickt wurde, auch als erste wieder nach dem FIFO-Prinzip von einem anderen Prozess empfangen werden.

2.2.1 Öffnen einer Warteschlange

Die gewünschte Warteschlange muss vor Benutzung von einem Prozess zuerst mit einem eindeutigen Schlüssel geöffnet bzw. angelegt werden. Dies geschieht mit folgender Systemfunktion:

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

Der Parameter *key* ist dabei der eindeutige Schlüssel der gewünschten Queue. Benutzt man als Schlüssel die Konstante `IPC_PRIVATE`, wird eine neue Queue eingerichtet. Eine neue Warteschlange kann jedoch auch erstellt werden, indem man im Parameter *msgflg* das Bit mit der Konstanten `IPC_CREAT` setzt. In diesem Fall kann man sich dann den Fehler, dass die Queue zu dem angegebenen Schlüssel schon existiert, mit dem Flag `IPC_EXCL` im *msgflg*-Parameter melden lassen.

In den untersten 9 Bits von *msgflg* werden zudem beim Neuanlegen einer Messagequeue die Zugriffsrechte in der üblichen oktalen Form angegeben, d.h. je drei Bits für "Owner", "Group" und "Others". Dabei werden jedoch nur die Bits für Lese- und Schreibzugriffe beachtet, das X-Bit hat keine Bedeutung.

Schlägt das Öffnen der Warteschlange nicht fehl, erhält man von `msgget()` die ID zurück, mit der man mit den im Folgenden erläuterten Funktionen die Messagequeue benutzen kann.

2.2.2 Senden einer Nachricht

Mit der Funktion `msgsnd()` kann ein Prozess eine Nachricht verschicken. Ihr Prototyp hat folgendes Aussehen:

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

Der erste Parameter gibt hierbei die oben erwähnte Queue-ID an. Der Parameter `msgsz` ist die Länge der Nachricht, und in `msgflg` können Flags wie `IPC_NOWAIT` spezifiziert werden, um ein Blockieren des Senderprozesses zu verhindern (z.B. wenn die Queue voll ist). Durch den Parameter `msgp` wird schließlich die Nachricht selbst übergeben, wobei die Struktur `msgbuf` wie folgt aufgebaut sein sollte:

```
struct msgbuf {
    long mtype;           /* Message Typ, muss > 0 sein */
    char mtext[SIZE];    /* Message Daten */
};
```

Die Größe des Arrays `mtext` sollte natürlich die gleiche sein wie im Parameter `msgsz` angegeben.

2.2.3 Empfangen einer Nachricht

Eine abgeschickte Nachricht muss natürlich auch von jemanden gelesen werden können. Dies geschieht mit dieser Funktion:

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)
```

Gelesen wird aus der Queue mit der ID `msqid` und die Nachricht mit der angegebenen Länge `msgsz` wird in der Struktur, auf die `msgp` zeigt, abgelegt. Eine Nachricht wird nach dem Lesen aus der Queue gelöscht, so dass sie nicht von mehreren lesenden Prozessen empfangen werden kann. Sollte die Nachricht länger sein als in `msgsz` angegeben, so wird ein Fehler gemeldet, es sei denn, man hat im Parameter `msgflg` das Bit `MSG_NOERROR` gesetzt, dann wird nämlich die Nachricht auf die Größe `msgsz` zurechtgestutzt. Auch hier kann ein Blockieren des Prozesses mit den Bit `IPC_NOWAIT` in `msgflg` verhindert werden.

Eine Besonderheit bei dieser Funktion ist der Parameter `msgtyp`. Mit ihm kann angegeben werden, auf welchen Nachrichtentyp gewartet werden soll. Ist `msgtyp=0`, wird die älteste Nachricht, egal von welchem Typ, gelesen. Ist `msgtyp>0`, dann wird die älteste Botschaft von genau diesem Typ gelesen, es sei denn, es wurde das Flag `MSG_EXCEPT` in `msgflg` gesetzt, denn dann wird diejenige älteste gelesen, bei welcher der Typ ungleich `msgtyp` ist. Ist `msgtyp<0`, wird die älteste Nachricht vom kleinsten Typ kleiner oder gleich dem Betrag von `msgtyp` gelesen.

2.2.4 Verwalten einer Warteschlange

Wie vielleicht bekannt, gibt es bei Dateien eine Funktion namens `ioctl()`, mit der sich diverse Parameter für den Dateizugriff einstellen lassen. Für Messagequeues gibt es nun eine ähnliche Funktion, um ihre Eigenschaften abzufragen und zu verändern:

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

In *msgid* wird wieder die Queue-ID übergeben und mit *cmd* kann eines der folgenden Kommandos ausgewählt werden:

- IPC_STAT : Speichert Informationen über die Queue in der Struktur, auf die *buf* zeigt.
- IPC_SET : Setzt einige Queue-Parameter auf die entsprechenden Felder der in der durch *buf* übergebenen Struktur (z.B. Zugriffsrechte etc.).
- IPC_RMID : Löschen der Queue aus dem System.

Die genaue Dokumentation der *msgid_ds*-Struktur würde hier zu weit führen, es sei daher auf Literatur wie [1] oder die man-Page von `msgctl()` verwiesen.

2.3 Shared Memory

Die bisher genannten Möglichkeiten zur IPC haben noch einen großen Nachteil: Da die auszutauschenden Daten immer mit Hilfe des Kernels verschickt werden, sind sie noch relativ langsam. Für zeitkritische Anwendungen liegt es auf der Hand, dass die Prozesse sich ja direkt ein Stück Speicher zum Datenaustausch teilen könnten - und genau dies wird mit "Shared Memory" verwirklicht.

2.3.1 Öffnen eines Shared-Memory-Bereichs

Ähnlich wie bei den Messages, muss ein Prozess erstmals auch für einen gemeinsamen Speicherbereich sich eine ID mit dem der folgenden Funktion zuweisen lassen:

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

Der Parameter *key* ist hierbei wieder IPC_PRIVATE oder ein eindeutiger Schlüssel mit dem alle beteiligten Prozesse sich identifizieren. Im Falle von IPC_PRIVATE sucht das System einen freien Key und legt jedenfalls ein neues Segment an. Mit *size* wird natürlich die Größe des gemeinsamen Speichersegments angegeben und mit *shmflg* können wieder einige Flags gesetzt werden: Mit den untersten 9 Bits werden wie üblich die Zugriffsrechte für "Owner", "Group" und "Others" angegeben. Mit dem Bit IPC_CREAT kann man erreichen, dass ein neues Segment angelegt wird, falls zu dem übergebenen Schlüssel noch keines existiert. Und mit IPC_EXCL kann man sich zusätzlich noch den Fehler beim Neuanlegen eines Segmentes melden lassen, dass das Segment zu dem angegebenen Key schon existiert.

2.3.2 Anbinden eines Segments

Hat man schließlich mit `shmget()` erfolgreich eine ID erhalten, kann man das Shared-Memory-Segment mit folgendem Befehl zur Benutzung an den Adressraum des Prozesses anbinden:

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

Dabei ist *shmid* die von `shmget()` erhaltene ID. Mit *shmaddr* kann man eine Adresse angeben, an der das Speichersegment angebinden werden soll. Man kann auch das System entscheiden lassen, wenn man 0 für *shmaddr* übergibt. In *shmflg* können wieder Flags gesetzt werden: Mit SHM_RDONLY kann das Speichersegment später nur zum Lesen benutzt werden, sonst kann gelesen und geschrieben werden. Und mit SHM_RND wird die übergebene Adresse auf ein Vielfaches der Systemkonstanten SHMLBA abgerundet. Dies hat den Sinn, dass kein Platz

am Anfang einer Speicherseite verschwendet wird. Als Rückgabewert erhält man von `shmat()` im Erfolgsfall die Speicheradresse, über welche man das Shared Memory Segments ansprechen kann.

2.3.3 Loslösen eines Segments

Ein mit `shmat()` angebundener Speicherabschnitt kann, wenn er nicht mehr gebraucht wird, mit folgender Funktion wieder aus dem Adressraum des Prozesses entfernt werden:

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

Als Parameter muss hierbei die Adresse des entsprechenden Speichersegments angegeben werden.

2.3.4 Verwalten eines Shared-Memory-Bereichs

Wie bei den Messages, gibt es bei Shared Memory ebenfalls eine Funktion für diverse Verwaltungsaufgaben:

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

Die Bedeutung der Parameter ist hierbei ähnlich wie bei `msgctl()`. Auch hier stehen für `cmd` die Möglichkeiten `IPC_STAT` und `IPC_SET` zum Auslesen und Setzen der Segment-Optionen (wie Zugriffsrechte etc.) zur Verfügung. Mit `IPC_RMID` kann der Shared-Memory-Bereich wieder endgültig entfernt werden. Unter Linux gibt es für Prozesse mit `root-UID` außerdem noch die Möglichkeit, mit dem Kommando `SHM_LOCK` zu verhindern, dass das Speichersegment auf die Festplatte ausgelagert wird. Und mit `SHM_UNLOCK` kann dies schließlich dem System wieder erlaubt werden.

2.3.5 Probleme

Wie man sich denken kann, ist Shared Memory die schnellste Möglichkeit zur Interprozesskommunikation. Leider hat sie den Nachteil, dass man die Prozesse noch zusätzlich synchronisieren muss - man stelle sich nur einmal vor, ein Prozess liest z.B. eine Liste aus dem gemeinsamen Segment komplett ein, obwohl der schreibende Prozess die Liste erst zur Hälfte erstellen konnte, bevor er vom Scheduler unterbrochen wurde. Netterweise gibt es unter Unix bzw. Linux auch Unterstützung vom Kernel, um solche Situationen zu meistern, z.B. mit Semaphoren...

2.4 Semaphore

Semaphore stellen mächtige Synchronisationsmechanismen dar, die einen sicheren gegenseitigen Ausschluss ("mutual exclusion", kurz "mutex") von Prozessen in kritischen Abschnitten ermöglichen (aber bei Fehlprogrammierung auch schnell zu Verklemmungen führen). Klassisch gesehen gibt es zwei elementare Semaphoroperationen, `P()` und `V()`, wobei `P()` zur "Reservierung" dient, d.h. ein Prozess testet damit, ob ein bestimmtes Betriebsmittel ("Resource") noch verfügbar ist und wird im negativen Fall vom Betriebssystem schlafen gelegt. `V()` ist das Gegenteil, damit kann ein Prozess eine "Resource" wieder freigeben, hierbei werden dann eventuell andere, wartende Prozesse wieder aufgeweckt.

Dabei wird die `P()`-Funktion im allgemeinen so realisiert, dass eine Variable um 1 dekrementiert wird. Falls die Variable hierbei negativ wird, legt das Betriebssystem den Prozess

schlafen (d.h. die Variable sollte anfangs sinnvollerweise auf einen positiven Wert vorinitialisiert sein). Bei V() wird die Variable wieder inkrementiert und ein evtl. wartender Prozess aufgeweckt, wenn die Variable noch kleiner als 1 ist. Für eine ausführlichere Erklärung über die beiden klassischen Operationen P() und V() sei an dieser Stelle jedoch auf entsprechende Literatur verwiesen (z.B.[3]).

Die Semaphore unter Unix/Linux sind nämlich noch um einiges mächtiger als die beiden klassischen Operationen P() und V(), so können z.B. in einem Systemaufruf mehrere Semaphore gleichzeitig geändert werden.

2.4.1 Öffnen einer Semaphoregruppe

Ähnlich wie bei den Messages und Shared-Memory, muss auch bei Semaphoren zuerst einmal wieder eine entsprechende Gruppe geöffnet bzw. angelegt werden:

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

Die Parameter *key* und *semflg* verhalten sich hierbei wieder wie bei shmget() bzw. msgget(), d.h. *key* sollte ein gültiger Schlüssel oder IPC_PRIVATE sein und in *semflg* werden die Zugriffsrechte und evtl. die Flags IPC_CREAT und IPC_EXCL gesetzt. Mit dem Parameter *nsems* wird die Anzahl der zu reservierenden Semaphore spezifiziert, wobei dieser Parameter 0 sein kann, wenn man den entsprechenden IPC-Bereich nur öffnen (also nicht anlegen) will. Als Rückgabewert erhält man die ID der entsprechenden Semaphore-Gruppe oder -1 falls ein Fehler auftrat.

2.4.2 Semaphore-Operationen

Nachdem man mit semget() eine Semaphore-Gruppe geöffnet hat, kann man nun die eigentlichen Semaphore-Operationen ausführen:

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

Der Parameter *semid* gibt hierbei die von semget() erhaltene ID an. Die einzelnen Semaphore-Operationen werden durch eine Liste der Länge *nsops* angegeben, auf die der Zeiger *sops* verweist. Die Struktur *sembuf* sieht hierbei wie folgt aus:

```
struct sembuf {
    short sem_num;      /* Nummer des Semaphore */
    short sem_op;      /* Semaphore-Operation */
    short sem_flg;     /* Flags */
}
```

Der Wert von *sem_op* wird dabei vorzeichenbehaftet zur Zählervariablen des Semaphores mit der Nummer *sem_num* addiert, jedoch nur wenn die Variable durch die Operation nicht negativ wird, denn dann wird der Prozess schlafen gelegt. Benutzt man also nur ein Semaphore und nur die Werte -1 und 1 für *sem_op*, kann man damit die klassischen Funktionen P() und V() leicht nachbilden. Falls man jedoch mehrere Semaphore auf einmal mit semop() verändern will, gilt: Wenn möglich werden alle angegebenen Semaphore-Operationen ausgeführt. Sollte dies nicht machbar sein, weil sonst eine der Semaphorevariablen negativ würde, dann wird vorerst keine der Operationen ausgeführt und der Prozess schlafen gelegt, bis alle Operationen

möglich sind. Setzt man *sem_op* auf 0, dann wartet der Prozess durch diesen Aufruf übrigens darauf, dass die entsprechende Semaphorvariable den Wert 0 erreicht.

Der letzte Eintrag *sem_flg* in der Struktur dient schließlich noch der genaueren Kontrolle über die entsprechende Semaphor-Operation. Hier kann das Flag `IPC_NOWAIT` gesetzt werden, mit dem man `semop()` dazu veranlassen kann, mit einer Fehlermeldung abzubrechen anstatt den Prozess schlafen zu legen, falls eine der Semaphor-Operationen nicht gleich ausführbar ist. Mit dem Flag `SEM_UNDO` kann dem System außerdem mitgeteilt werden, die entsprechende Semaphor-Operation mitzuprotokollieren, damit sie bei einem Abbruch des Prozesses (z.B. bei CTRL-C) automatisch rückgängig gemacht wird. Somit kann verhindert werden, dass andere Prozesse noch durch das entsprechende Semaphor blockiert werden, obwohl der Prozess, der für das Sperren des Semaphores verantwortlich war, schon lang beendet wurde.

2.4.3 Verwalten einer Semaphorgruppe

Wie schon bei Messages und Shared-Memory, gibt es auch bei der Familie der Semaphor-Funktionen einen Aufruf für diverse Kontroll-Operationen:

```
union semun {
    int val;                /* Wert für SETVAL */
    struct semid_ds *buf;   /* Buffer für IPC_STAT, IPC_SET */
    unsigned short int *array; /* Array für GETALL, SETALL */
};
int semctl ( int semid, int semnum, int cmd, union semun arg )
```

Der Parameter *semid* ist natürlich wieder die ID von `semget()`. *semnum* gibt die Nummer des Semaphores (falls benötigt) an. Für *cmd* kommen einige Operationen in Frage, die interessantesten hierbei sind:

- `IPC_STAT` : Es werden allgemeine Informationen (z.B. Zugriffsrechte) über die Semaphorgruppe in einem *semid_ds* Buffer zurückgegeben.
- `IPC_SET` : Übernimmt einige Werte aus dem *semid_ds* Buffer (z.B. Zugriffsrechte).
- `IPC_RMID` : Löscht die entsprechende Semaphorgruppe.
- `GETALL` : Gibt die Werte aller Semaphorvariablen in einem Array zurück.
- `GETNCNT` : Ermittelt die Anzahl der Prozesse, die auf ein Inkrementieren des Semaphors *semnum* warten.
- `GETPID` : Die ID des Prozesses, der als letztes eine Operation auf dem Semaphor *semnum* ausgeführt hat, wird zurückgegeben.
- `GETVAL` : Wert einer Semaphorvariablen ermitteln.
- `GETZCNT` : Ermittelt die Anzahl der Prozesse, die darauf warten, dass das entsprechende Semaphor den Wert 0 annimmt.
- `SETALL` : Setzt die Werte aller Semaphorvariablen auf die des übergebenen Arrays.
- `SETVAL` : Wert einer Semaphorvariablen setzen.

Eine ausführlichere Liste findet sich z.B. in der man-Page zu `semctl()`.

2.5 Signale

Mit Signalen ist wohl jeder Linux-Benutzer schon mal in Berührung gekommen, und zwar durch das Kommando "kill", mit dem sich Signale an Prozesse schicken lassen, um z.B. einen ("hängenden") Prozess zu beenden. Signale kann man sich wie Interrupts bei der Hardware vorstellen: Wenn ein Signal eintrifft, wird die Ausführung des entsprechenden Prozesses unterbrochen und stattdessen der entsprechende Signal-Handler ausgeführt, sofern der Prozess eine Handler-Routine installiert hat. Tatsächlich werden auch einige der Signale gesendet, nachdem ein Hardware-Interrupt aufgetreten ist. Die meisten anderen Signale werden bei diversen Ereignissen vom Kernel an Prozesse verschickt.

Da Signale jedoch auch von Prozess zu Prozess gesendet werden können, sind sie auch begrenzt zur Interprozesskommunikation geeignet. Besonders die beiden Signale SIGUSR1 und SIGUSR2 sind für diesen Fall von Interesse.

2.5.1 Die verschiedenen Signale

Es gibt mehrere Typen von Signalen, die auch unterschiedliche Eigenschaften in Bezug auf die Ausführung des Prozesses haben, wenn dieser keinen Signal-Handler installiert hat bzw. das Signal nicht ignoriert. Folgende Tabelle gibt eine Übersicht über die wichtigsten Signale:

Signalname	Nummer	Aktion	Bemerkung
SIGHUP	1	A	Verbindung beendet (z.B. Modem, Terminal)
SIGINT	2	A	Interrupt-Signal von der Tastatur
SIGQUIT	3	A	Quit-Signal von der Tastatur
SIGILL	4	A	Falsche Instruktion
SIGTRAP	5	C	Überwachung/Stop Punkt
SIGABRT	6	C	Abbruch
SIGFPE	8	C	Fehler bei Fließkomma-Operation
SIGKILL	9	AE	Beendigungssignal
SIGUSR1	10	A	Benutzer-definiertes Signal 1
SIGSEGV	11	C	Ungültige Speicherreferenz
SIGUSR2	12	A	Benutzer-definiertes Signal 2
SIGPIPE	13	A	Schreiben in eine Pipe ohne Leseprozess
SIGALRM	14	A	Zeitsignal von alarm().
SIGTERM	15	A	Beendigungssignal
SIGCHLD	17	B	Kind-Prozess beendet
SIGCONT	18		Fortfahren, wenn gestoppt
SIGSTOP	19	DE	Prozessstop

Tabelle 1: Signale unter Linux (basierend auf [5])

Ein mit "B" in der Aktions-Spalte der Tabelle markiertes Signal wird hierbei einfach ignoriert, wenn der Prozess keinen Signal-Handler installiert hat. Der Buchstabe "A" dagegen bedeutet, dass der Prozess bei dem entsprechenden Signal abgebrochen wird, sofern er keinen Handler installiert hat oder das Signal explizit ignoriert. Die mit "E" markierten Signale (SIGSTOP und -KILL) lassen sich jedoch nicht abfangen oder ignorieren. Bei Signalen mit der Markierung "C" gibt es noch einen Core-Dump und bei "D" wird der Prozess schlafen gelegt.

2.5.2 Abfangen eines Signals

Zum Setzen eines eigenen Signal-Handlers benutzt man folgende Funktion:

```
void (* signal( int signum, void (*handler)(int) ) )(int)
```

Die Nummer des Signals wird in *signum* angegeben, die Adresse des Signal-Handlers im Parameter *handler*. Dabei können als *handler* auch die Konstanten SIG_DFL und SIG_IGN benutzt werden, um auf das Standardverhalten zurückzuschalten bzw. das Signal zu ignorieren. Als Rückgabewert erhält man den vorherigen Wert von *handler*.

2.5.3 Verschicken eines Signals

Verschicken kann man Signale mit diesem Systemaufruf:

```
int kill ( pid_t pid, int sig )
```

kill() sendet das Signal mit der Nummer *sig* an den Prozess mit der Prozess-ID *pid*. Ein Spezialfall hierbei ist, wenn *pid* auf 0 gesetzt wird oder negativ ist. Bei 0 wird das Signal nämlich an alle Prozesse gesendet, die in der gleichen Gruppe wie der aktuelle Prozess sind. Bei -1 wird das Signal an wirklich alle Prozesse im System gesendet, und bei einer anderen negativen Nummer an alle Prozesse, bei denen die Gruppennummer gleich dem Betrag der angegebenen Nummer ist. Diese Spezialfälle werden aber wohl eher selten zur "richtigen" IPC gebraucht, sondern eher zum Beenden aller Prozesse beim Herunterfahren des Systems.

2.5.4 Warten auf ein Signal

Schließlich gibt es noch die Möglichkeit, auf eintreffende Signale in einem Programm zu warten:

```
int pause ( void )
```

pause() wartet einfach solange, bis ein Signal eintrifft. So bleibt dem Programmierer das Schreiben von eigenen Warteschleifen erspart.

3 IPC auf der Kommandozeile

Wer sich nun genauer mit dem Thema IPC unter Linux beschäftigen will, muss nicht einmal gleich zum C-Compiler greifen, es existieren auch einige Tools für die Kommandozeile.

3.1 Named Pipes

Die schon als Funktionen beschriebenen Kommandos "mkfifo" und "mknod" gibt es auch als normale Programme, mit denen Pipe-Dateien leicht von der Shell aus angelegt werden können. Tippt man z.B. in einer Shell folgendes:

```
mkfifo test.fifo
cat test.fifo
```

Dann erscheint dort alles auf der Konsole, was man anderswo in diese Fifo-Datei hinein schreibt (z.B. mit "echo Hallo >test.fifo").

3.2 Messages, Shared-Memory und Semaphore

Auch für Messages, Shared-Memory und Semaphore gibt es zwei Programme: Mit "ipcs" kann man sich eine Liste mit den vorhandenen Message-Queues, Shared-Memory Segmenten und Semaphoren samt ihren zugehörigen Attributen wie Schlüssel und IDs anzeigen lassen. Entdeckt man hierbei einen Bereich, der gar nicht mehr existieren sollte (z.B. weil das entsprechende Programm längst beendet oder abgestürzt ist), kann man ihn über seine ID mit dem Programm "ipcrm" aus dem System löschen.

3.3 Signale

Schließlich sei hier nochmals das Programm "kill" erwähnt, mit dem sich alle möglichen Signale verschicken lassen - nicht nur um Prozesse zu beenden. So lässt sich z.B. mancher Hintergrundprozess (daemon) wie der "inetd" mit dem zweckentfremdeten Signal SIGHUP dazu überreden, seine Konfigurationsdateien neu einzulesen, ohne dass man ihn komplett neu starten muss. Oder man hält vorübergehend einen rechenzeitintensiven Prozess mittels SIGSTOP an. Später, wenn man nicht mehr auf die volle CPU-Leistung angewiesen ist, kann man ihn dann mit dem Signal SIGCONT wieder zum Leben erwecken.

Literatur

- [1] Ch. Dannegger, P. Geugelin-Dannegger: **Parallele Prozesse unter UNIX: Simulation und Anwendung bekannter Synchronisationsmethoden in C unter UNIX**, Hanser Verlag, ISBN 3-446-15911-8, 1991
- [2] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner: **Linux-Kernel-Programmierung**, Addison-Wesley Verlag, 3. Auflage, ISBN 3-89319-939-X, 1995
- [3] J.L. Keedy: **Technische Informatik - Betriebssystemkonzepte (Skript)**, Fakultät für Informatik Universität Ulm, 1998
- [4] **Linux Programmer's Manual**, diverse Linux Man-Pages zum Thema IPC.
- [5] **"man 7 signal"**, Man-Page mit Liste der verfügbaren Signale.