

Allerlei Nützliches

1 Semaphore

1.1 Allgemein

Semaphore sind unter System V IPC erweitert:

- Ganze Arrays von Semaphoren können auf einmal angelegt werden.
- In einer Operation können mehrere Semaphore auf einmal modifiziert werden.
- Das Inkrementieren und Dekrementieren kann in größeren Schritten als 1, bzw. -1 erfolgen.
- Semaphor-Operationen können nach Beendigung des Prozesses rückgängig gemacht werden, d.h. bei Aufruf von `exit()` werden alle rückgängig zu machenden Operationen zurückgesetzt.

1.2 Header

Benötigt werden:

```
<sys/ipc.h>  
<sys/sem.h>  
<sys/types.h>
```

1.3 Anlegen eines Semaphors

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

- `key`: Schlüssel für das Semaphor-Array, `IPC_PRIVATE` als spezieller Wert, wenn kein existierendes Objekt genommen werden soll
- `nsems`: Anzahl der Semaphore
- `semflg`:
 - 0400: Leserecht Erzeuger
 - 0200: Schreibrecht Erzeuger
 - usw.
 - `IPC_CREAT`: anlegen, wenn es das Semaphor nicht gibt
 - `IPC_EXCL`: wenn `IPC_CREAT` gesetzt ist, und der Semaphor existiert, kehrt die Funktion mit dem Fehler `EEXIST` zurück
- mehrere flags werden durch `|` verodert

- Rückgabewert: Filedescriptor, negativer Wert bei Fehlern

Typischer Aufruf:

```
int semid;

if (0 > (semid = semget(0x2a, 10, (IPC_CREAT | 0666))))
{
    perror("`problem with semget()");
    exit(1);
}
```

Es wird ein Array mit zehn Semaphoren mit Schlüssel *0x2a* angelegt, Lese- und Schreibrechte für alle. Wenn das Semaphorarray noch nicht existiert, wird es neu angelegt. Die einzelnen Semaphor werden über Nummern von 0 bis 9 angesprochen.

1.4 Operationen auf Semaphoren

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

- *semid*: Id des Semaphorarrays
- *sops*: Array mit den Operationen, Index 0 erste Operationen, Index 1 zweite, usw.
- *nsops*: Anzahl der Operationen
- Rückgabewert: negativer Wert bei Fehlern

Operationen:

```
struct sembuf
{
    ushort sem_num; //Index des Semaphors
    short sem_op; //auszuführende Operation
    short sem_flg; //Flags
}
```

Flags bei Operationen:

- *IPC_NOWAIT*: nichtblockierender Aufruf, d.h. wenn die Semaphoroperation nicht ausgeführt werden kann, wird nicht gewartet, sondern man kommt sofort zurück auf dem Funktionsaufruf
- *SEM_UNDO*: diese Operation wird bei Beendigung des Prozesses rückgängig gemacht

Typischer Aufruf:

```
struct sembuf sops[SEM_NUM];

sops[0].sem_num = 1;
sops[0].sem_op = 1;
sops[0].sem_flg = 0;
```

```

if(0 > semop(semid, &(sops[0]), 1))
{
    perror("`problem with semop()');
    exit(1);
}

```

Es wird eine Operation auf dem Semaphor mit der id *semid* ausgeführt. Es handelt sich um ein klassisches *up*, d.h. der Zähler des Semaphors wird um eins erhöht. Ein *down* sieht analog folgendermaßen aus: `sops[0].sem_op = -1` Wird hier 0 angegeben, wartet der Prozess darauf, dass der Zähler des Semaphors den Wert 0 annimmt!

1.5 Kommandos auf Semaphoren

```
int semctl(int semid, int semnum, int cmd, ...);
```

- *semid*: id des Semaphorarrays
- *semnum*: Nummer eines Semaphors
- *cmd*: auszuführendes Kommando
- `...`: Argumente für das Kommando
- Rückgabewert: hängt vom Kommando ab, negativ bei Fehler

Kommandos:

- *IPC_STAT*: Infos, Argument wird benötigt
- *IPC_SET*: Werte für die Struktur angeben, z.B. Zugriffsrechte, Argument wird benötigt, *semnum* wird ignoriert
- *IPC_RMID*: Semaphorarray löschen, alle wartenden Prozesse werden aufgeweckt, kein Argument, *semnum* wird ignoriert
- *GETALL*: die Werte aller Semaphore zurückgeben, Argument wird für die Rückgabe benötigt, *semnum* wird ignoriert
- *GETNCNT*: Anzahl der Prozesse, die auf den *semnum*-ten Semaphor warten, Argument wird nicht benötigt, Rückgabewert ist die Anzahl
- *GETPID*: PID des Prozesses, der *semop()* auf dem *semnum*-ten Semaphor zuletzt ausgeführt hat, ist der Rückgabewert
- *GETVAL*: gibt den Wert des *semnum*-ten Semaphor zurück
- *GETZCNT*: Anzahl der Prozesse, die darauf warten, dass der *semnum*-te Semaphor den Wert 0 annimmt
- *SETALL*: alle Semaphore werden gesetzt, Argument wird benötigt, *semnum* wird ignoriert
- *SETVAL*: einen Semaphor setzen, Argument wird benötigt

Argumente:

```

union semun
{
    int val;                //Wert für SETVAL
    struct semid_ds *buf;   //Puffer für IPC_STAT und IPC_SET
    ushort *array,         //Feld für GETALL und SETALL
    struct seminfo *__buf; //Puffer für IPC_INFO
}

```

Typische Aufrufe:

```

if(0 > semctl(semid, 10, IPC_RMID))
{
    perror("`problem with semctl'");
    exit(1);
}

```

Das Semaphorarray mit der id \$semid\$ wird sofort gelöscht.

```

union semun args;
args.val = 3;

if(0 > semctl(semid, 1, SETVAL, args))
{
    perror("`problem with semctl'");
    exit(1);
}

```

Der zweite Semaphor im Array mit der Id *semid* und dem Index 1 wird auf den Wert 3 gesetzt. Typischer Aufruf zum Initialisieren von Semaphoren.

2 Shared Memory

2.1 Header

Benötigt werden:

```

<sys/ipc.h>
<sys/shm.h>
<sys/types.h>

```

2.2 Anlegen von Shared Memory

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

- *key*: Schlüssel für das Shared Memory, *IPC_PRIVATE* als spezieller Wert, wenn keine existierendes Objekt genommen werden soll
- *size*: Größe des Shared Memory
- *shmflg*:

– 0400: Leserecht Erzeuger

- 0200: Schreibrecht Erzeuger
 - usw.
 - *IPC_CREAT*: anlegen, wenn es das Shared Memory nicht gibt
 - *IPC_EXCL*: wenn *IPC_CREAT* gesetzt ist, und das Shared Memory existiert, kehrt die Funktion mit dem Fehler *EXISTS* zurück
- mehrere flags werden durch | verodert
 - Rückgabewert: Filedescriptor, negativer Wert bei Fehlern

Typischer Aufruf:

```
typedef struct
{
    int numberOfReindeers;
    int numberOfElfes;
} shared_t;

int shmid;
shared_t *shmPtr;

if(0 > (shmid = shmget(0x2a, sizeof(shared_t), (IPC_CREAT | 0666))))
{
    perror("`problem with semget()');
    exit(1);
}
```

Ein Shared Memory mit dem Schlüssel *0x2a* der Größe *sizeof(shared_t)* wird angelegt (sofern es noch nicht existiert). Alle haben Lese- und Schreibrechte.

2.3 Shared Memory in Memory des Prozesses einblenden

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

- *shmid*: ID des Shared Memory
- *shmaddr*: Adresse des Shared Memory in der Memory Map des aufrufenden Prozesses, bei 0 wird eine passende freie Speicherstelle ausgewählt
- *shmflg*: Flags
- Rückgabewert: Filedescriptor, negativ, wenn's nicht klappt

Flags:

- *SHM_RND*: wenn die Adresse nicht 0 ist, dann wird die in *shmaddr* angegebene Adresse abgerundet auf den nächsten mehrfachen *SHMLBA*-Wert, sonst muss *shmaddr* in einen Seitenrahmen passen, d.h. Anfang einer Page sein
- *SHM_REMAP*: Mapping soll bestehendes Mapping ersetzen (nur Linux)
- *SHM_RDONLY*: readonly

Typischer Aufruf:

```

if(0 > (shmPtr = shmat(shmid), 0, 0))
{
    perror("`problem with shmat()");
    exit(1);
}

```

Das Shared Memory mit der Id *shmid* wird in die Memory Map des aufrufenden Prozesses eingebunden. Die Adresse wird vom Betriebssystem passend ausgewählt.

2.4 Shared Memory aus Memory des Prozesses ausblenden

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

- *shmaddr*: Zeiger auf Shared Memory im Prozess
- Rückgabewert: negativ, wenn's nicht klappt

Typischer Aufruf:

```

if(0 > shmdt(shmPtr)) //bei exit automatisch ausgeführt, aber Memory besteht
{
    perror("`problem with shmdt()");
    exit(1);
}

```

Das Shared Memory, auf das der Pointer *shmPtr* zeigt, wird aus der Memory Map des aufrufenden Prozesses ausgeblendet.

2.5 Kommandos auf Shared Memory

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

- *shmid*: Id des Shared Memory
- *cmd*: auszuführendes Kommando
- *buf*: Argument für Kommando
- Rückgabewert: hängt vom Kommando ab, negativ bei Fehler

Kommandos:

- *IPC_STAT*: Infos, Argument wird benötigt
- *IPC_SET*: Werte für die Struktur angeben, z.B. Zugriffsrechte, Argument wird benötigt
- *IPC_RMID*: Shared Memory löschen, nach dem letzten *shmdt()*, kein Argument
- *SHM_LOCK*: nur Superuser, kein Swapping möglich, d.h. die Seiten können nicht auf die Festplatte ausgelagert werden
- *SHM_UNLOCK*: nur Superuser, Swapping wieder möglich

Typischer Aufruf:

```
if(0 > shmctl(shmid, IPC_RMID, NULL)
{
    perror("`problem with shmctl'");
    exit(1);
}
```

Das Shared Memory mit der Id *shmid* soll gelöscht werden. Das eigentliche Löschen findet aber erst statt, wenn alle Prozesse, die das Shared Memory eingebunden haben, ein *shmdt()* ausgeführt haben, d.h. es sichergestellt ist, dass kein Prozess mehr darauf zugreift.