Kapitel 4

critical sections und Semaphore

Programme, die sich Resourcen teilen, müssen Codeabschnitte allein (exklusiv) ausführen, diese codeteile nennt man critical section. Um dies zu erreichen werden diese sections durch Semaphore gelockt. Dies sind Mechanismen, die anzeigen, dass momentan das Programm in einer critical section ist, und ein anderes Programm die Resource nicht verwenden darf.

4.1 Beispiel, Begriffe

Code mit einer critical section besteht üblicherweise aus 4 Teilen:

- entry section: Es wird exklusiver Zugriff verlangt.
- critical section: der Code arbeitet unter exklusiven Zugriff an der Resoource.
- exit section: Der exklusive Zugriff wird beendet.
- remainder section: der restliche Programm Code.

Wenn ein Methode dieses Problem handlen soll, sind folgende Rahmenbedingungen zu beachten:

- allein: Nur ein Process darf seine critical section ausführen.
- weiter: Ist kein Process in seiner critical section, muss es einem Process möglich sein in eine neue critical section einzutreten. Wenn es drum geht welcher Process Zugriff bekommt, sollen nur solche eine Rolle spielen, die nicht in der remainder section sind.
- maximale Wartezeit: kein Process darf ewig warten bis er in seine critical section darf. D.h. andere dürfen nur endlich oft bevorzugt werden.

Diese Problemlösestrategie nennt man symmetrisch. Dazu ein Beispiel: Mehrere Processe werden erzeugt, diese schreiben auf stderr. stderr ist eine gemeinsame Resource.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

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

{
char buffer[MAX_CANON];
char *c;
int i;
int n;
pid_t childpid;

if ( (argc != 2) || ((n = atoi(argv[1])) <= 0) ) {
    fprintf (stderr, "Usage: %s number_of_processes\n",
argv[0]);
    exit(1);
}

for (i = 1; i < n; ++i)
     if (childpid = fork())
            break;

sprintf(buffer,
    "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
    i, (long)getpid(), (long)getppid(), (long)childpid);

c = buffer;
/*****************start of critical section
*******************/
while (*c != '\0') {
    fputc(*c, stderr); c++; }
fputc('\n', stderr);
/*******************end of critical section
*******************/
exit(0);
}
```

Das Schreiben auf stderr erfolgt nicht exklusiv, d.h. die Ausgabe von verschieden Processen kann gemischt werden, wie folgendes Beispiel zeigt:

```
i:48 process ID:4046 parent ID:4045 child ID:4047
i:1 process ID:3999 parent ID:3961 child ID:4000
i:47 process ID:4045 parent ID:4044 child Ii:46 process
ID:4044 parent ID:1 child ID:4045
i:21 process ID:4019 parent ID:1 child ID:4020
```

Wir werden Methoden sehen, wie man exklusiven Zugriff erhält. Obiges Problem versucht man durch das Konzept der *atomic operation* zu lösen, dies sind Programmabschnitte, die nicht unterbrochen werden können. Dies ist aber schwierig zu erreichen, da man als Programmierer nicht weiss in wieviele Anweisungen ein Befehl vom Compiler zerlegt wird. Man sieht dies am folgenden naiven Beispiel:

```
while(lock)
;
lock=1;
<critical section>
lock=0;
<remainder section>
```

Das Testen(while) und Setzen(=1) ist nicht atomic, d.h. es kann passieren, dass zwei Processe die Schleife beenden, weil der erste unterbrochen wurde vor dem Setzen des locks. Das würde durch eine atomic Version z.B. TestAndSet ändern.

```
int TestAndSet(int *target)
{
int returnval;

returnval = *target;
*target = 1;
return returnvalue;
}
```

Dann würde das Programm so aussehen:

```
while(TestAndSet(&lock)) ;
<critical section>
lock=0;
<remainder section>
```

Was aus der Vorgabenliste für das handling fehlt, ist die Gewährleistung, dass ein Process nicht endlos warten muss.

4.2 Semaphore

Diese wurden von Dijkstra 1965 vorgeschlagen zur Synchronisation und zur Realisierung exklusiver Resourcen Zugriffe. Ein Semaphor ist eine int Variable, die zwei Zugriffe erlaubt. wait und signal. wait bedeutet dass der Process wartet, bis der Semaphor positiv (>0) wird. Danach wird der Semaphor um eins erniedrigt. Die Funktion signal erhöht den Semaphor um 1. In POSIX 1b nennt man dies lock und unlock. Unser Beispiel sieht dann so aus:

```
wait(&S);
<critical section>
signal(&S);
<remainder section>
```

Wobei der Semaphor S mit 1 beginnt. Unter POSIX 1b gehts wie folgt:

```
NAME
sem_init - initialize an unnamed semaphore

SYNOPSIS
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value
);
```

Damit wird ein namenloser Semaphor erzeugt, ist der Parameter pshared verschieden von 0, wird er zwischen Prozessen geteilt (vererbt wie filedescriptoren) und value ist der ursprüngliche Wert. Zum manipulieren hat man folgende Befehle.

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
```

Mit sem_destroy wird der zuvor erzeugt namenlose Semaphor gelöscht. sem_wait wartet bis der Semaphor freischaltet. Dies blockt, d.h. der Processstatus wechselt auf sleep (no busy wait). Ähnlich ist trywait, hier wird nicht geblockt, sondern der Rückgabewert -1 und errno=EAGAIN bedeutet, dass der Process durch den Semaphor nicht freigeschaltet wurde. Mit sem_post wird der Semaphor um 1 erniedrigt (=signal). sem_getvalue erlaubt das Abfragen eines Semaphors. Dazu nochmal das Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>
#include <semaphore.h>

void main (int argc, char *argv[])
{
char buffer[MAX_CANON];
char *c;
int i;
int n;
pid_t childpid;
sem_t my_lock;

if ( (argc != 2) || ((n = atoi(argv[1])) <= 0) ) {
  fprintf (stderr, "Usage: %s number_of_processes\n",
argv[0]);
  exit(1);}
if (sem_init(&my_lock, 1, 1) == -1) {
   perror("Could not initialize mylock semaphore");
   exit(1);}

for (i = 1; i < n; ++i)
   if (childpid = fork()) break;

sprintf(buffer,
"i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
i, (long)getpid(), (long)getppid(), (long)childpid);

c = buffer;
/************************entry section *************/
if (sem_wait(&my_lock) == -1) {
  perror("Semaphore invalid");
  exit(1);}
/****************start of critical section ************/
while (*c != '\0') {
fputc(*c, stderr);
c++; }
fputc('\n', stderr);
/******************end of critical section
****************/

/*********************** exit section
*******************/
if (sem_post(&my_lock) == -1) {
perror("Semaphore done");
exit(1);}
/******************** remainder section
******************/
exit(0);
}
```

Die nächste Stufe sind named Semaphoren, diese können auch zwischen Processen ohne gemeinsamen Vorgänger geteilt werden. named Semaphore haben wie files einen Namen und Zugriffsrechte. Für den Namen gelten Konventionen wie für filenamen, daher sollten zwei Processe einen Namen beginnend mit / wählen (d.h. Angabe des absoluten Pfades), um sicherzustellen, dass sie den gleichen named Semaphor meinen. Die Routinen lauten ähnlich den file routinen:

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                unsigned long mode, unsigned int value
);
int sem_unlink(const char *name);
int sem_close(sem_t *sem);
```

Der Aufruf vom sem_open dient zum Öffnen eines vorhandenen named Semaphors, man braucht den Namen und oflag muss 0 sein. Danach kann man den Zeiger mit den bekannten Aufrufen verwenden. Um einen neuen Semaphor zu erzeugen dient die zweite Syntax. oflag hat dann den Wert O_CREAT oder O_CREAT | O_EXCL dann wird der Semaphor erzeugt, sofern er noch nicht existiert, mode gibt die Zugriffsrechte und value den ersten Wert. Ist er schon da gibt die zweite Syntax mit oflag = O_CREAT einen Zeiger auf den vorhandenen, im Fall oflag = O_CREAT | O_EXCL gibt es einen Fehler. Die anderen beiden Routinen sem_unlink und sem_close dienen Entfernen bzw. Schliessen eines named Semaphor.

4.3 System V Semaphore

Semaphore sind Teil des System V Interprocess Communication Pakets (IPC), welches nicht nur Sempahore sondern auch shared memory und message queues enthält. IPC ist nicht Teil von POSIX 1, wird aber in Spec 1170 standardisiert. Die Semaphore Datenstruktur ist Teil des Kernels, und wird über ein handle mittels Routinen manipuliert, der Benutzer erhält nur ein handle. Dies ist wie bei den filehandles.

4.3.1 Erzeugen

Die System V Routinen arbeiten mit einem semaphore set, die ist ein Feld von Semaphoren, wobei ein Semaphore eine Struktur ist, die mindestens folgende Komponenten enthält:

- eine int Zahl, die den Wert enthält (≥ 0)
- die PID des letzten Process, der manipuliert hat.
- die Anzahl der Processe, die warten, dass der Wert steigt.
- die Anzahl der Processe, die warten, dass der Wert 0 wird.

Die System V Routinen erlauben zu warten (blocking) bis ein Semaphore 0 wird oder bis ein Semaphore grösser wird. Um ein Semaphor zu erzeugen hat man den Aufruf

```
NAME
semget - get a semaphore set identifier

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget ( key_t key, int nsems, int semflg )
```

Der Aufruf liefert ein handle mit dem im weiteren gearbeitet werden kann, oder der Rückgabewert ist -1 und errno wird gesetzt. Der erste Parameter ist ein Schlüssel (int) der das semaphore set identifiziert. Für diesen Schlüssel gibt es drei Möglichkeiten. Mit dem Macro IPC_PRIVATE wird vom System ein Schlüssel geliefert, man kann selber eine Zahl sich überlegen, oder mit ftok(3) kann man aus einem Pfadnamen einen Schlüssel generieren. Der zweite Parameter ist die Anzahl der Semaphore in dieser Menge. Der letzte Parameter setzt die flags beim Erzeugen, wie schon bekannt gibt es IPC_CREAT, IPC_EXCL und weitere flags, die die Zugriffsrechte steuern, dazu ein Beispiel:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define PERMS S_IRUSR|S_IWUSR
#define SET_SIZE 3

int semid;

if ((semid = semget(IPC_PRIVATE, SET_SIZE, PERMS)) < 0)
   perror("Could not create new private semaphore");
```

Die Flag Parameter bedeuten, dass nur Processe mit gleichen UID Lesen und Schreiben dürfen. Der Verwendung von IPC_PRIVATE bedeutet ferner, dass auf alle Fälle ein neues semaphore set erzeugt wird. Nachfolgend ein Beispiel, wo ein Benutzer key verwendet wird:

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include <errno.h>

#define PERMS S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define SET_SIZE 1
#define KEY ((key_t)99887)

int semid;
if ((semid = semget(KEY, SET_SIZE, PERMS | IPC_CREAT)) <
0)
  fprintf(stderr, "Error creating semaphore with key %d:
%s\n",
               (int)KEY, strerror(errno));
```

Diesmal wird nur ein Semaphore erzeugt mit mode 0666 und Schlüsselwert KEY. Ist ein sempahore set mit diesem Schlüssel schon vorhanden, so wird ein handle darauf zurückgegeben, es sei denn das flag beim Aufruf wird durch PERMS | IPC_CREAT | IPC_EXCL ersetzt. Dann liefert der Aufruf einen Fehler und errno = EEXIST. Die dritte Möglichkeit ist mittels ftok:

```
NAME
ftok - convert a pathname and a project identifier to a
System V IPC key

SYNOPSIS
# include <sys/types.h>
# include <sys/ipc.h>

key_t ftok ( char *pathname, char proj )
```

aus einem Pfadnamen und einem zusätzlichen identifier einen Schlüssel abzuleiten. Dies wird im folgenden Programm verwendet:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include <errno.h>

#define PERMS S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define SET_SIZE 2

void main(int argc, char *argv[])
{
int semid;
key_t mykey;

if (argc != 3) {
  fprintf(stderr, "Usage: %s filename id\n", argv[0]);
  exit(1); }
if ((mykey = ftok(argv[1], atoi(argv[2]))) == (key_t)
-1) {
  fprintf(stderr, "Could not derive key from filename
%s: %s\n",
      argv[1], strerror(errno));
  exit(1); }
else if ((semid = semget(mykey, SET_SIZE, PERMS
|IPC_CREAT)) < 0) {
  fprintf(stderr, "Error creating semaphore with key %d:
%s\n",
      (int)mykey, strerror(errno));
  exit(1); }

printf("semid = %d\n", semid);
exit(0);
}
```

und eine Folge von Beispielaufrufen:

```
$ a.out /tmp 1
semid = 0
$ a.out /tmp 1
semid = 0
$ a.out /tmp 2
semid = 1
$ a.out /tmp 1
semid = 0
$ a.out /tmp 2
semid = 1
$ a.out /tmp 3
semid = 2
$ a.out /tmp 1
semid = 0
```

4.3.2 Operationen

Ein Process kann ein Semaphor incrementieren, decrementieren und auf 0 Testen, dazu:

```
NAME
semop - semaphore operations

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop ( int semid, struct sembuf *sops, unsigned
nsops)
```

Dabei ist der 1. Parameter das handle aus semget. Der zweite Paraemter ist ein Feld der Länge nsops von Operationen. Eine Operation ist eine Struktur mit den Komponenten

- short sem_num: welches Semaphore
- short sem_op: welche Operation
- short sem_flg: welche Flags bei der Operation

Ist sem_op > 0, wird die Zahl zum Wert addiert, und die Process, die auf eine Erhöhung warten werden gestartet. Ist sem_op $= 0$, wird der Process geblockt und er wartet darauf, dass der Semaphor 0 wird. Ist sem_op < 0, wird der Wert abgezogen, vorausgesetzt das Ergebnis ist ≥ 0. Geht das nicht wartet der Process bis der Semaphor

grösser wird. Ist das Ergebnis nach dem Abzug = 0, werden die Processe gestartet die auf Semaphor = 0 warten. Dazu ein Beispiel, welches eine Routine set_sembuf_struct, die die sembuf Struktur setzt, verwendet:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
struct sembuf GET_TAPES[2];
struct sembuf RELEASE_TAPES[2];

set_sembuf_struct(&(GET_TAPES[0]), 0, -1, 0);
set_sembuf_struct(&(GET_TAPES[1]), 1, -1, 0);
set_sembuf_struct(&(RELEASE_TAPES[0]), 0, 1, 0);
set_sembuf_struct(&(RELEASE_TAPES[1]), 1, 1, 0);

Process 1:
semop(S, GET_TAPES, 1);
<use tape A>
semop(S, RELEASE_TAPES, 1);

Process 2:
semop(S, GET_TAPES, 2);
<use tapes A and B>
semop(S, RELEASE_TAPES, 2);

Process 3:
semop(S, GET_TAPES + 1, 1);
<use tape B>
semop(S, RELEASE_TAPES + 1, 1);
```

wenn der semop Aufruf durch ein Signal unterbrochen wird liefert er -1 und errno=EINTR, ferner ist gewährleistet, dass die Manipilation eines Semaphors atomic ist. Nun noch eine Implementation des 1. Beispiels mittels System V Semaphoren (unter Zuhilfename einer Routine remove_semaphore):

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <limits.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define PERMS S_IRUSR | S_IWUSR
#define SET_SIZE 2

void set_sembuf_struct(struct sembuf *s, int semnum,int
semop, int semflg);
int remove_semaphore(int semid);

void main (int argc, char *argv[])
{
char buffer[MAX_CANON];
char *c;
int i;
int n;
pid_t childpid;
int semid;
int semop_ret;
struct sembuf semwait[1];
struct sembuf semsignal[1];
int status;

if ( (argc != 2) || ((n = atoi (argv[1])) <= 0) ) {
  fprintf (stderr,"Usage:%s number_of_processes\n",argv[0]);
  exit(1);}

/* Create a semaphore containing a single element */
if ((semid = semget(IPC_PRIVATE, SET_SIZE, PERMS)) ==
-1) {
  fprintf(stderr, "[%ld]:Could not access
semaphore:%s\n",
     (long)getpid(), strerror(errno));
  exit(1); }

/* Initialize the semaphore element to 1 */
set_sembuf_struct(semwait, 0, -1, 0);
set_sembuf_struct(semsignal, 0, 1, 0);

if (semop(semid, semsignal, 1) == -1) {
  fprintf(stderr, "[%ld]: semaphore increment failed -
%s\n",
      (long)getpid(), strerror(errno));
  if (remove_semaphore(semid) == -1)
         fprintf(stderr,"[%ld],could not delete
semaphore-%s\n",
              (long)getpid(), strerror(errno));
  exit(1);}
```

```
for (i = 1; i < n; ++i)
if (childpid = fork()) break;
sprintf(buffer,
"i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
i, (long)getpid(), (long)getppid(), (long)childpid);
c = buffer;

/************************entry section *************/
while(( (semop_ret = semop(semid, semwait, 1)) == -1)
               &&(errno == EINTR)) ;
if (semop_ret == -1)
    fprintf(stderr,"[%ld]:semaphore decrement failed -
%s\n",
         (long)getpid(), strerror(errno));
else {
/*****************start of critical section ********/
  while (*c != '\0') {
  fputc(*c, stderr); c++; }
  fputc('\n', stderr);
  /*******************end of critical section ****/
  /*********************** exit section *********/
  while(((semop_ret = semop(semid, semsignal, 1)) ==
-1)
              &&(errno == EINTR)) ;
  if (semop_ret == -1)
    fprintf(stderr,"[%ld]:semaphore increment
failed-%s\n",
          (long)getpid(), strerror(errno));
}
/******************** remainder section
****************/
while((wait(&status) == -1) && (errno == EINTR)) ;
if (i == 1) /* the original process removes the
semaphore */
  if (remove_semaphore(semid) == -1)
       fprintf(stderr,"[%ld],could not delete
semaphore-%s\n",
            (long)getpid(), strerror(errno));

exit(0);
}
```

4.3.3 Semaphore control

Um Semaphore abzufragen oder zu setzen gibt es die Funktion semctl:

```
NAME
 semctl - semaphore control operations
SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, [union semun
arg]);
```

Dabei sind semid und semnum handle und Index im semaphore set. Das vierte Argument wird nur manchmal benötigt und ist eine union mit folgenden Komponenten:

- int val;
- ...

Als cmd kommen in Frage:

- GETVAL: Abfrage des Wertes
- SETVAL: Abfrage des Wertes auf arg.val
- GETPID: Abfrage der PID des letzten Process, der manipuliert hat
- GETNCNT: Abfrage der Anzahl der Processe, die auf increment warten
- GETZCNT: Abfrage der Anzahl der Processe, die auf =0 warten
- IPC_RMID: Löschen des Semaphors

Die Routine remove_semaphore aus dem vorherigen Beispiel, schaut dann so aus:

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

int remove_semaphore(int semid)
{
return semctl(semid, 0, IPC_RMID);
}
```

4.4 Shell

Es gibt unter System V auch Shellroutinen zum Management des IPC:

```
$ ipcs

------ Shared Memory Segments --------
shmid owner perms bytes nattch status

------ Semaphore Arrays --------
semid owner perms nsems status
0 axel 666 2
1 axel 666 2
2 axel 666 2

------ Message Queues --------
msqid owner perms used-bytes messages

$ ipcrm sem 0
$ ipcrm sem 1
$ ipcrm sem 2
```

4.5 Aufgabe

Man schreibe eine Hotelverwaltung: Es gibt einen Manager (server), der mehrere Betten zu vergeben hat, in dem ein Gast (client) 10 sec schlafen darf. Man implementiere dies mittels System V Semaphoren. Der server wird als erstes im Hintergrund gestartet und läuft bis zum Ausloggen. Die clients (=Nachfrage nach einem Bett) werden in beliebiger Zahl im Hintergrund gestartet. Der Aufruf ist:

server Anzahl-Betten&
client&