

Musterlösungen zu den Hausaufgaben auf
Blatt 5
der Übungen zur Vorlesung
“Grundlagen Betriebssysteme und Systemsoftware

G.Groh, 15.11.2008

Aufgabe 1

```
#define N 100                                /* Anzahl Plätze im Puffer */
int count = 0;                               /* Anzahl der Elemente im Puffer */

void producer(void)
{
    int item;

    while( TRUE ) {                          /* Endlosschleife */
        item = produce_item();               /* erzeuge nächstes Element */
        if( count == N ) sleep();           /* wenn Puffer voll, schlafe */
        insert_item(item);                  /* schreibe Element in Puffer */
        count = count + 1;                  /* erhöhe Anzahl der Elemente */
        if( count == 1 ) wakeup(consumer); /* war der Puffer leer? */
    }
}

void consumer(void)
{
    int item;

    while( TRUE ) {                          /* Endlosschleife */
        if( count == 0 ) sleep();           /* wenn Puffer leer, schlafe */
        item = remove_item();               /* hole Element aus Puffer */
        count = count - 1;                  /* erniedrige Anzahl der Elemente */
        if( count == N-1 ) wakeup(producer);/* war der Puffer voll? */
        consume_item(item);
    }
}
```

Erklären Sie den Ablauf des Programms!

Antwort: Es existieren zwei parallel laufende Prozesse, wobei einer vom anderen abhängig ist. Der erste Prozess beginnt zu produzieren und kann somit den zweiten "nähren", womit dieser das Produzierte abarbeiten (verbrauchen) kann. Beide Prozesse laufen in einer Endlosschleife. *item* wird produziert und verbraucht. Dies erfolgt über die Methoden *produce item()* und *consume item()*. In der Variable *count* werden die bereits Produzierten *items* gezählt. In *N* ist die Größe des verfügbaren Puffers definiert. Sie gibt an, wie viele *items* im Puffer gehalten werden können. Ist der Puffer voll, so versetzt sich der Erzeuger in den „Schlafen-Zustand“. Nachdem der Verbraucher ein *item* wieder abgebaut hat, weckt er den Erzeuger wieder auf. Sollte der Puffer leer sein, so versetzt sich der Verbraucher in den „Schlafen-Zustand“. Dieser wird wiederum vom Erzeuger wieder aufgeweckt, sobald ein neues *item* erzeugt wurde.

- Erzeuger: Nachdem dieser ein *item* erzeugt hat, überprüft er, ob der Puffer bereits voll ist. Wenn ja, wird *sleep()* aufgerufen und der Prozess zurückgestellt. Ist noch Platz im Puffer frei, so wird *item* in diesen eingefügt und der Pufferzähler *count* um eins inkrementiert. War zuvor der Puffer leer, so muss abschließend der Verbraucher aus dem "Schlafen-Zustand" wieder geweckt werden. Dies erfolgt mit Hilfe von *wakeup(consumer)*.
- Verbraucher: Ist der Puffer leer, so wird der Prozess zurück gestellt. Ansonsten wird ein *item* aus dem Puffer entnommen und der Pufferzähler um eins dekrementiert. Ist nun der Puffer um eins kleiner als die Maximalgröße, so wird *wakeup(producer)* aufgerufen, um diesen aus dem „Schlafen-Zustand“ wieder zu erwecken. Zum Abschluss wird *item* konsumiert.

Für eine ausführlichere Erläuterung siehe Tanenbaum (Engl. Ausgabe 2001) S108ff

Kann es bei dem Ablauf zu Problemen kommen? Welche?

- **Antwort:**

Angenommen der Verbraucher hat gerade die Variable *count* überprüft, ob diese 0 ist. In diesem Augenblick wird durch den Scheduler der Prozess zurückgestellt und der Erzeuger fortgesetzt. Dieser produziert ein neues *item* und fügt es in den Puffer ein. *Count* wird um eins inkrementiert und steht somit auf 1. Der Erzeuger ruft daraufhin *wakeup(consumer)* auf, obwohl dieser noch gar nicht schläft. Der Aufruf geht somit unbeachtet verloren. Nun kommt allerdings wieder der Verbraucher an die Reihe und stellt fest, dass die zuvor ausgelesene *count* Variable auf 0 war und geht schlafen. Der Erzeuger produziert nun solange weiter, bis der Puffer voll ist und geht ebenfalls schlafen. Somit kann keiner der beiden Prozesse mehr reanimiert werden.

Aufgabe 1.2

```
#define N 100                                     /* Anzahl Plätze im Puffer */
int count = 0;                                    /* Anzahl der Elemente im Puffer */

void producer(void)
{
    int item;

    while( TRUE ) {                               /* Endlosschleife */
        item = produce_item();                   /* erzeuge nächstes Element */
        if( count == N ) sleep();               /* wenn Puffer voll, schlafe */
        insert_item(item);                      /* schreibe Element in Puffer */
        count = count + 1;                      /* erhöhe Anzahl der Elemente */
        if( count == 1 ) wakeup(consumer);     /* war der Puffer leer? */
    }
}

void consumer(void)
{
    int item;

    while( TRUE ) {                               /* Endlosschleife */
        if( count == 0 ) sleep();               /* wenn Puffer leer, schlafe */
        item = remove_item();                   /* hole Element aus Puffer */
        count = count - 1;                      /* erniedrige Anzahl der Elemente */
        if( count == N-1 ) wakeup(producer);   /* war der Puffer voll? */
        consume_item(item);
    }
}
```

Wenn hier unterbrochen wird kann es zu race kommen

Aufgabe: Verändern Sie den Code so, dass die genannten Probleme nicht mehr auftreten können! Verwenden Sie dazu Semaphoren!

- Notieren Sie eine geeignete Definition eines Typs *semaphore*!
- Deklarieren Sie Semaphoren, die Sie zur Lösung der Probleme einsetzen wollen!

- Die Lösung besteht darin die Zugriffe auf *count* zu synchronisieren. Dazu sind drei Semaphoren notwendig. Eine, die den Zugriff auf die kritischen Bereiche steuert, eine, die leere Pufferplätze speichert und eine, die volle Pufferplätze speichert:

```
#define N 100 /* Anzahl Plätze im Puffer */  
typedef int semaphore; /* Semaphoren sind spezielle Integerwerte */  
semaphore mutex = 1; /* steuert den Zugriff auf kritische Bereiche */  
semaphore empty = N; /* zählt leere Pufferplätze */  
semaphore full = 0; /* zählt volle Pufferplätze */
```

Aufgabe 1.3

Neuer Code:

```
void producer(void) {
    int item;
    while (TRUE){
        item = produce_item(); /*
erzeuge Element */
        p(&empty); /* erniedrige
Zähler für leere Plätze;
blockiere ggf.*/
        p(&mutex); /* trete in den
kritischen Bereich ein;
blockiere ggf. */
        insert_item(item); /* lege
Element in Puffer */
        v(&mutex); /* verlasse
kritischen Bereich */
        v(&full); /* erhöhe Anzahl
der vollen Plätze */
    }
}
```

```
void consumer(void) {
    int item;
    while (TRUE){
        p(&full); /* erniedrige
Zähler für volle Plätze;
blockiere ggf. */
        p(&mutex); /* trete in den
kritischen Bereich ein;
blockiere ggf. */
        item = remove_item(); /*
nimm Element aus Puffer */
        v(&mutex); /* verlasse
kritischen Bereich */
        v(&empty); /* erhöhe Anzahl
der leeren Plätze */
        consume_item(item); /*
verbrauche element */
    }
}
```

Aufgabe 2

Process S

```
{
  while (TRUE)
  {
    <zur Rampe fahren>;
    <1 Schrank aufladen>;
    <Rampe verlassen>;
  }
}
```

Process C

```
{
  while (TRUE)
  {
    <zur Rampe fahren>;
    <1 Couch aufladen>;
    <Rampe verlassen>;
  }
}
```

Process F

```
{
  while (TRUE)
  {
    <zur Rampe fahren>;
    <1 Schrank entladen>;
    <1 Couch entladen>;
    <Rampe verlassen>;
  }
}
```

Aufgabe 2.1

Angenommen, in der Lagerhalle befinden sich s Schränke und c Couches. Wir verwenden dann je einen Semaphor für die drei Ungleichungen $20 \geq s + c$, $s \geq 0$, $c \geq 0$. Für die exklusive Benutzung der Rampe verwenden wir einen boolschen Semaphor (Mutex).

Wir brauchen also folgende Semaphore:

- Boolescher Semaphor `mutexRampe` mit Startwert 1; zu Beginn ist damit die Rampe frei
- Semaphor `lagerhalle` mit Startwert 20; verhindert die Anfahrt des Fabrikanten F, wenn die Lagerhalle nicht Platz für mindestens 2 Möbelstücke hat
- Semaphor `schränkZahl` mit Startwert 0; zählt die Anzahl der Schränke in der Lagerhalle; verhindert die Anfahrt des Ausfahrers S, falls sich kein Schrank in der Lagerhalle befindet
- Semaphor `couchZahl` mit Startwert 0; zählt die Anzahl der Couches in der Lagerhalle; verhindert die Anfahrt des Ausfahrers S, falls sich keine Couch in der Lagerhalle befindet.

Aufgabe 2.2

Die P- und V-Operationen und die Deklarationen müssen folgendermaßen eingebaut werden:

```
mutexRampe(1);  
lagerhalle(20);  
schrankZahl(0);  
couchZahl(0);
```

Process S

```
{  
  while (TRUE)  
  {  
    P(schrankZahl);  
    P(mutexRampe);  
    <zur Rampe fahren>;  
    <1 Schrank aufladen>;  
    V(lagerhalle);  
    <Rampe verlassen>;  
    V(mutexRampe);  
  }  
}
```

Process C

```
{  
  while (TRUE)  
  {  
    P(couchZahl);  
    P(mutexRampe);  
    <zur Rampe fahren>;  
    <1 Couch aufladen>;  
    V(lagerhalle);  
    <Rampe verlassen>;  
    V(mutexRampe);  
  }  
}
```

Process F

```
{  
  while (TRUE)  
  {  
    P(lagerhalle);  
    P(lagerhalle);  
    P(mutexRampe);  
    <zur Rampe fahren>;  
    <1 Schrank entladen>;  
    V(schrankZahl);  
    <1 Couch entladen>;  
    V(couchZahl);  
    <Rampe verlassen>;  
    V(mutexRampe);  
  }  
}
```

Die P-Operation für den Semaphor `lagerhalle` muss zweimal aufgerufen werden um sicherzustellen, dass zwei Plätze in der Lagerhalle frei sind.

Aufgabe 3

```
#include <stdio.h>
#define FALSE 0
#define TRUE 1

void critical_region(int thread_index);

int b[2]; /*global variable => all values initialy 0 (false)*/
int x = 0; /*global variable the two threads shall modify => should be 3 after modification*/

void critical_region(int thread_index) { /*thread_index is 0 or 1*/
    int other; /*number of the other thread..*/
    other = 1 - thread_index; /*.. is the opposite of thread_index*/
    while (b[other]); /*wait while other thread in critical region*/
    b[thread_index] = TRUE; /*thread wants to enter critical region*/
    /*-----*/
    /******Critical Region******/
    printf("Thread%d in critical region!!\n", thread_index);
    int tmp_x = x;
    tmp_x = tmp_x + thread_index + 1;
    x = tmp_x;
    printf("Thread%d in critical region!!, x = %d\n", thread_index, x);
    /******End of Critical Region******/
    /*-----*/
    b[thread_index] = FALSE; /*this thread is leaving critical region*/
}
}
```

Aufgabe 3.1 Lösungsvorschlag

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define FALSE 0
#define TRUE 1

void *critical_region(void *thread);

int b[2]; /*global variable => all values initially 0 (false)*/
int x = 0; /*global variable the two threads shall modify => should be 3 after modification*/

void *critical_region(void *thread_index_ptr) { /*thread_index is 0 or 1*/
    int thread; /*thread_index*/
    thread = *(int*)thread_index_ptr; /*get thread_index from pointer*/
    int other; /*number of the other thread..*/
    other = 1 - thread; /*.. is the opposite of thread*/
    while (b[other]); /*wait while other thread in critical region*/
    sleep(2);
    b[thread] = TRUE; /*thread wants to enter critical region*/
    /*-----*/
    /******Critical Region******/
    printf("Thread%d in critical region!!\n", thread);
    int tmp_x = x;
    sleep(2);
    tmp_x = tmp_x + thread + 1;
    x = tmp_x;
    printf("Thread%d in critical region!!, x = %d\n", thread, x);
    /******End of Critical Region******/
    /*-----*/
    b[thread] = FALSE; /*this thread is leaving critical region*/
    return NULL;
}
```

Aufgabe 3.1 Lösungsvorschlag

```
int main() {  
  
    int thread_index0 = 0;  
    int thread_index1 = 1;  
    pthread_t t0, t1; /*two threads*/  
  
    /*Create two threads that call function critical_region passing their index*/  
    pthread_create(&t0, NULL, critical_region, &thread_index0);  
    pthread_create(&t1, NULL, critical_region, &thread_index1);  
  
    /*main thread waits for the two child threads to finish*/  
    pthread_join(t0, NULL);  
    pthread_join(t1, NULL);  
  
    printf("All threads finished!\n");  
  
    return 0;  
}
```

Beispielablauf, während dem beide Threads in den kritischen Bereich eintreten:

t0	t1	Wert
		b[0] == b[1] == FALSE
	while (b[0]);	b[0] == FALSE
while (b[1]); b[0] = TRUE; -kritischer Bereich	b[1] = TRUE; -kritischer Bereich	b[1] == FALSE b[0] == TRUE b[1] == TRUE

Test: Nach while und ggf. nach lesen von x sleep()
=> x nach Durchlauf der beiden Threads nicht 3

Erster Lösungsansatz: Erst setzen, dann testen.

```
b[thread] = TRUE; /*thread wants to enter critical
    region*/

while (b[other]) {
}; /*wait while other thread in critical region*/

/*****Critical Region*****/
...
/*****End of Critical Region*****/

b[thread] = FALSE; /*this thread is leaving critical
    region*/
```

⇒ wA ist gewährleistet, aber es ist eine Verklemmung möglich!

Test: Nach $b[thread] = TRUE$ sleep()
=> Programm bleibt hängen

Aufgabe 3.3 Lösungsvorschlag

Zweiter Lösungsansatz: Zurücknehmen des eigenen Wunsches, wenn der andere Thread seinen Wunsch bereits geäußert hat.

```
b[thread] = TRUE; /*thread wants to enter critical region*/

while (b[other]) {
    b[thread] = FALSE; /*revoke wish to enter critical region*/
    sleep(5); /*sleep five seconds*/
    b[thread] = TRUE; /*try again*/
}; /*wait while other thread in critical region*/

/*****Critical Region*****/
...
/*****End of Critical Region*****/

b[thread] = FALSE; /*this thread is leaving critical region*/
```

⇒ wA ist gewährleistet und keine Verklemmung

Jedoch: Aushungern möglich. *Einseitig:* Einer kommt immer an die Reihe und beidseitig, wenn beide immer gleichzeitig testen

Aufgabe 3.3 Lösungsvorschlag

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define FALSE 0
#define TRUE 1

void *critical_region(void *thread);

int b[2]; /*global variable => all values initialy 0 (false)*/
int x = 0; /*global variable the two threads shall modify => should be 3 after modification*/

void *critical_region(void *thread_index_ptr) { /*thread_index is 0 or 1*/
    int thread; /*thread_index*/
    thread = *(int*)thread_index_ptr; /*get thread_index from pointer*/
    int other; /*number of the other thread..*/
    other = 1 - thread; /*.. is the opposite of this thread*/
    b[thread] = TRUE; /*thread wants to enter critical region*/
    sleep(2);
    while (b[other]) {
        b[thread] = FALSE; /*revoke wish to enter critical region*/
        sleep(5); /*sleep five seconds*/
        b[thread] = TRUE; /*try again*/
    }; /*wait while other thread in critical region*/
    /*-----*/
    /******Critical Region******/
    printf("Thread%d in critical region!!\n", thread);
    int tmp_x = x;
    tmp_x = tmp_x + thread + 1;
    x = tmp_x;
    printf("Thread%d in critical region!!, x = %d\n", thread, x);
    /******End of Critical Region******/
    /*-----*/
    b[thread] = FALSE; /*this thread is leaving critical region*/
    return NULL;
}
```

Aufgabe 3.3 Lösungsvorschlag

```
int main() {  
  
    int thread_index0 = 0;  
    int thread_index1 = 1;  
    pthread_t t0, t1; /*two threads*/  
  
    /*Create two threads that call function critical_region passing their index*/  
    pthread_create(&t0, NULL, critical_region, &thread_index0);  
    pthread_create(&t1, NULL, critical_region, &thread_index1);  
  
    /*main thread waits for the two child threads to finish*/  
    pthread_join(t0, NULL);  
    pthread_join(t1, NULL);  
  
    printf("All threads finished!\n");  
  
    return 0;  
}
```