Barry Linnert

Nichtsequentielle und verteilte Programmierung, SS2021

Übung 6

Tutor: Florian Alex
Tutorium 3

Rui Zhao, William Djalal, Simeon Vasilev

29. Mai 2021

1 Produktion in C (12 Punkte)

Implementieren Sie in C eine Simulation des Problems der Produzent*innen und Konsument*innen. Programmieren Sie zur Absicherung einen Monitor, der mindestens die Funktionen Ablegen und Entnehmen anbietet. Die jeweilige Anzahl der beteiligten Produzent*innen bzw. Konsument*innen soll beliebig, aber fest gewählt werden können. Die einzelnen Aktionen und der Zustand des Puffers soll jeweils ausgegeben werden. Dokumentieren Sie Ihr jeweiliges Programm und stellen Sie immer die Ausgaben des jeweiligen Programms zur Verfügung.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_THREADS 5
#define NUM_PLACES 4
#define NUM_PRODUCER 3
//here choose the number of Producers as man like
//and the rest are Consumers (which is NUM_THREADS - NUM_PRODUCER)


int last;
int buffer_free;
int buffer[NUM_PLACES];
// The data can also be placed inside the 'struct' so that it is structurally more
    beautiful.
// But for better readability of the code, i.e. to reduce a lot of 'struct name -> data',
// I chose to put the data outside the 'stuct'.

struct monitor{
        // int buffer_free;
        // int buffer[NUM_PLACES];
        pthread_mutex_t mutex;
        pthread_cond_t consume_cond, produce_cond;
};

static void init_monitor(struct monitor *act)
{
        buffer_free = NUM_PLACES;
        pthread_mutex_init(&act->mutex, 0);
        pthread_cond_init(&act->consume_cond,0);
        pthread_cond_init(&act->produce_cond,0);
```

ok

```
35          //init buffer with -1
36          for (int t = 0; t < NUM_PLACES; t++){
37                  buffer[t] = -1;
38          }
39          printf("init buffer with:\n");
40          for (int i = 0; i < NUM_PLACES; i++)
41                  {
42                          printf("%d buffer: %d\n", i, buffer[i]);
43                  }

45  }

47  static void _produce(struct monitor *act, int i, void *threadid)
48  {
49          pthread_mutex_lock(&act->mutex);

51          // 'buffer is full'
52          while( buffer_free == 0 ){
53                  pthread_cond_wait(&act->produce_cond, &act->mutex);
54          }

56          // crit.
57          buffer[last] = i;
58          printf("Producer %ld puts %d into buffer at place %d \n", (long) threadid, buffer[
        last], last);
59          last++;
60          buffer_free --;
61          for (int i = 0; i < NUM_PLACES; i++)
62          {
63                  printf("%d buffer: %d\n", i, buffer[i]);
64          }


67          pthread_cond_signal(&act->consume_cond);
68          pthread_mutex_unlock(&act->mutex);
69  }

71  static void _remove(struct monitor *act, int amount, void *threadid)
72  {
73          pthread_mutex_lock(&act->mutex);
74          // avoid warning
75          (void) amount;

77          // 'buffer is empty'
78          while( buffer_free == NUM_PLACES ){
79                  pthread_cond_wait(&act->consume_cond, &act->mutex);
80          }

82          // crit.
83          printf ("Consumer %ld takes %d from buffer at place %d \n", (long) threadid, buffer
        [last-1], last-1);
84          buffer[last-1] = -1;
85          fflush (stdout);
86          last--;
87          buffer_free ++;
88          for (int i = 0; i < NUM_PLACES; i++)
89          {
90                  printf("%d buffer: %d\n", i, buffer[i]);
91          }

93          pthread_cond_signal(&act->produce_cond);
94          pthread_mutex_unlock(&act->mutex);
95  }

97  void* Producer (void *threadid);

99  void* Consumer (void *threadid);
```

```
101 struct monitor act;

103 int main(){
104         // init
105         pthread_t threads[NUM_THREADS];
106         int rc;
107         long t;

109         init_monitor(&act);

111         // creating threads
112         for(t=0; t < NUM_THREADS; t++) {
113                 if (t < NUM_PRODUCER){
114                         printf ("Creating Producer with ID: %ld\n", t);
115                         rc = pthread_create (&threads[t], NULL, Producer, (void *)t);
116                 }
117                 else{
118                         printf ("Creating Consumer with ID: %ld\n", t);
119                         rc = pthread_create (&threads[t], NULL, Consumer, (void *)t);
120                 }
121                 if (rc){
122                         exit (-1);
123                 }
124         }

126         // joining threads
127         for(t=0; t < NUM_THREADS; t++) {
128                 pthread_join (threads[t], NULL);
129         }
130         pthread_exit(NULL);

132         return 0;
133 }

135 void* Producer (void *threadid)
136 {
137         int i;
138         for (i= 0; i < 10; i++) {
139                 _produce(&act, i, threadid);

141         }
142         pthread_exit (NULL);
143 }

145 void* Consumer (void *threadid)
146 {
147         while (1) {
148                 _remove(&act, 1, threadid);
149         }
150         pthread_exit (NULL);
151 }
```

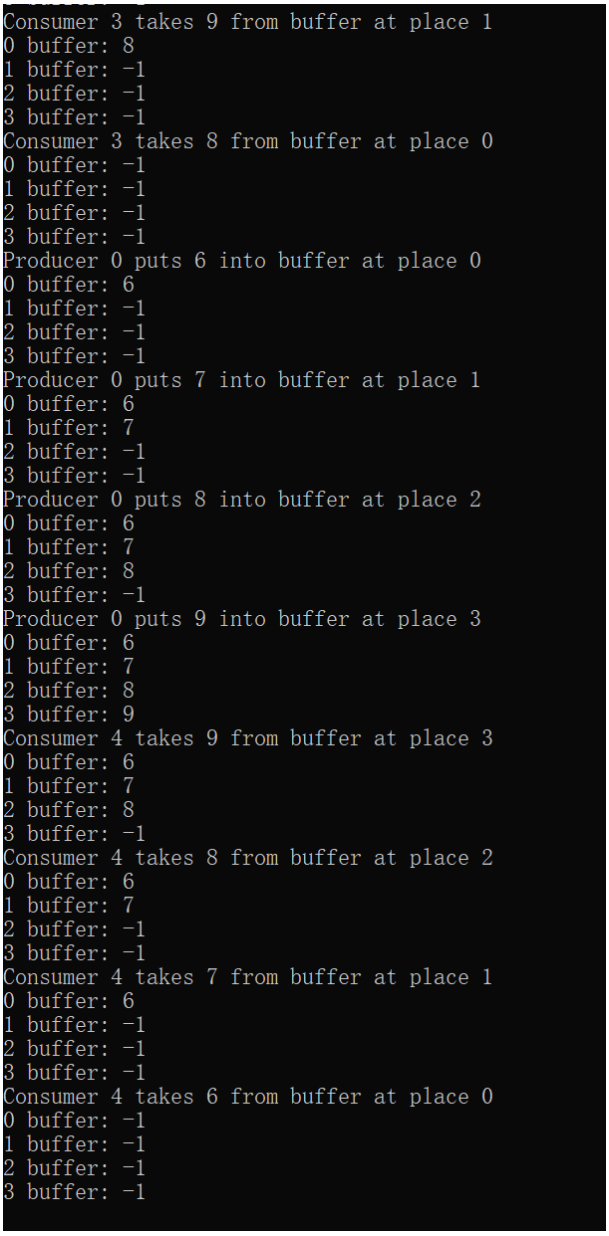

Abbildung 1: A1 test screen

Die Idee ist hier, einen NUM_PRODUCER zu definieren, der verwendet wird, um auszuwählen, wie viele 'Producer' es gibt und der Rest sind 'Consumer'.
Sowohl Produzenten als auch Konsumenten haben ihre eigenen eindeutigen IDs, z. B.
NUM_THREADS 10 : insgesamt zehn Threads
NUM_PRODUCER 6 : sechs von ihnen sind 'Producer'
Die IDs 0-5 gehören zu den 'Producer'
Die IDs 6-9 gehören zu den 'Consumer'

Da C keinen eingebauten Monitor hat, wird hier (Mutex + pthread_cond_wait/pthread_cond_signal + struct) verwendet, um die Monitor-Funktionalität zu implementieren. ✓

2 Reihenfolgeerhaltende Produktion (8 Punkte)

Erweitern Sie Ihre Lösung der Aufgabe 1 so, dass die Produkte in der Reihenfolge entnommen werden, in der sie abgelegt wurden.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_THREADS 5
#define NUM_PLACES 4
#define NUM_PRODUCER 3
//here choose the number of Producers as man like
//and the rest are Consumers (which is NUM_THREADS - NUM_PRODUCER)

int first = 0;  // index for remove
int last = 0;   // index for produce
int buffer_free;
int buffer[NUM_PLACES];
// The data can also be placed inside the 'struct' so that it is structurally more beautiful.
// But for better readability of the code, i.e. to reduce a lot of 'struct name -> data',
// I chose to put the data outside the 'stuct'.

struct monitor{
        // int buffer_free;
        // int buffer[NUM_PLACES];
        pthread_mutex_t mutex;
        pthread_cond_t consume_cond, produce_cond;
};

static void init_monitor(struct monitor *act)
{
        buffer_free = NUM_PLACES;
        pthread_mutex_init(&act->mutex, 0);
        pthread_cond_init(&act->consume_cond,0);
        pthread_cond_init(&act->produce_cond,0);

        //init buffer with -1
        for (int t = 0; t < NUM_PLACES; t++){
                buffer[t] = -1;
        }
        printf("init buffer with:\n");
        for (int i = 0; i < NUM_PLACES; i++)
                {
                        printf("%d buffer: %d\n", i, buffer[i]);
                }

}

static void _produce(struct monitor *act, int i, void *threadid)
{
        pthread_mutex_lock(&act->mutex);

        // 'buffer is full'
        while( buffer_free == 0 ){
                pthread_cond_wait(&act->produce_cond, &act->mutex);
        }

        // crit.
        buffer[last] = i;
        printf("Producer %ld puts %d into buffer at place %d \n", (long) threadid, buffer[last], last);
        last = (last + 1) % NUM_PLACES;
        buffer_free --;
```

```
61          for (int i = 0; i < NUM_PLACES; i++)
62          {
63                  printf("%d buffer: %d\n", i, buffer[i]);
64          }


67          pthread_cond_signal(&act->consume_cond);
68          pthread_mutex_unlock(&act->mutex);
69  }

71  static void _remove(struct monitor *act, int amount, void *threadid)
72  {
73          pthread_mutex_lock(&act->mutex);
74          // avoid warning
75          (void) amount;

77          // 'buffer is empty'
78          while( buffer_free == NUM_PLACES ){
79                  pthread_cond_wait(&act->consume_cond, &act->mutex);
80          }

82          // crit.
83          printf ("Consumer %ld takes %d from buffer at place %d \n", (long) threadid, buffer
        [first], first);
84          buffer[first] = -1;
85          fflush (stdout);
86          first = (first + 1) % NUM_PLACES;
87          buffer_free ++;
88          for (int i = 0; i < NUM_PLACES; i++)
89          {
90                  printf("%d buffer: %d\n", i, buffer[i]);
91          }

93          pthread_cond_signal(&act->produce_cond);
94          pthread_mutex_unlock(&act->mutex);
95  }

97  void* Producer (void *threadid);

99  void* Consumer (void *threadid);

101 struct monitor act;

103 int main(){
104         // init
105         pthread_t threads[NUM_THREADS];
106         int rc;
107         long t;

109         init_monitor(&act);

111         // creating threads
112         for(t=0; t < NUM_THREADS; t++) {
113                 if (t < NUM_PRODUCER){
114                         printf ("Creating Producer with ID: %ld\n", t);
115                         rc = pthread_create (&threads[t], NULL, Producer, (void *)t);
116                 }
117                 else{
118                         printf ("Creating Consumer with ID: %ld\n", t);
119                         rc = pthread_create (&threads[t], NULL, Consumer, (void *)t);
120                 }
121                 if (rc){
122                         exit (-1);
123                 }
124         }

126         // joining threads
127         for(t=0; t < NUM_THREADS; t++) {
```

```
128                     pthread_join (threads[t], NULL);
129             }
130             pthread_exit(NULL);

132             return 0;
133     }

135     void* Producer (void *threadid)
136     {
137             int i;
138             for (i= 0; i < 10; i++) {
139                     _produce(&act, i, threadid);

141             }
142             pthread_exit (NULL);
143     }

145     void* Consumer (void *threadid)
146     {
147             while (1) {
148                     _remove(&act, 1, threadid);
149             }
150             pthread_exit (NULL);
151     }
```

Die Idee ist hier, zwei Indizes (Index last, first) hinzuzufügen.

”last”wird verwendet, um den Producer zu leiten, wo er die Ware lagern soll. Der Index wird nach jeder Ablegung erhöht.
(last = (last + 1) % NUM_PLACES;)

”first”wird verwendet, um den Consumer zu zeigen, wo er die Ware abholen kann. In ähnlicher Weise wird der Index nach jeder Abholung inkrementiert.
(first = (first + 1) % NUM_PLACES;)

Dadurch wird sichergestellt, dass die Produkte in der Reihenfolge entnommen werden, in der sie abgelegt wurden.

✓

```
zhaor@6POONT1-88WD3YW:~/alp4/u6$ gcc -std=c11 -Wall -Wextra -pedantic -pthread a2.c -o a2
zhaor@6POONT1-88WD3YW:~/alp4/u6$ ./a2
init buffer with:
0 buffer: -1
1 buffer: -1
2 buffer: -1
3 buffer: -1
Creating Producer with ID: 0
Creating Producer with ID: 1
Producer 0 puts 0 into buffer at place 0
0 buffer: 0
1 buffer: -1
2 buffer: -1
3 buffer: -1
Creating Producer with ID: 2
Producer 0 puts 1 into buffer at place 1
0 buffer: 0
1 buffer: 1
2 buffer: -1
3 buffer: -1
Producer 0 puts 2 into buffer at place 2
0 buffer: 0
1 buffer: 1
2 buffer: 2
3 buffer: -1
Producer 0 puts 3 into buffer at place 3
0 buffer: 0
1 buffer: 1
2 buffer: 2
3 buffer: 3
Creating Consumer with ID: 3
Creating Consumer with ID: 4
Consumer 3 takes 0 from buffer at place 0
0 buffer: -1
1 buffer: 1
2 buffer: 2
3 buffer: 3
Consumer 3 takes 1 from buffer at place 1
0 buffer: -1
1 buffer: -1
2 buffer: 2
3 buffer: 3
Consumer 3 takes 2 from buffer at place 2
0 buffer: -1
1 buffer: -1
2 buffer: -1
3 buffer: 3
Consumer 3 takes 3 from buffer at place 3
0 buffer: -1
1 buffer: -1
2 buffer: -1
3 buffer: -1
```

Abbildung 2: A2 test screen

3 Unterscheidung des Konsums (10 Punkte)

Erweitern Sie Ihre Lösung der Aufgabe 2 so, dass die Konsument*innen jeweils beim Entnehmen eine beliebig, aber fest gewählte Anzahl von Produkten abholen.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_THREADS 5
#define NUM_PLACES 4
#define NUM_PRODUCER 3
//here choose the number of Producers as man like
//and the rest are Consumers (which is NUM_THREADS - NUM_PRODUCER)


int last;
int buffer_free;
int buffer[NUM_PLACES];
// The data can also be placed inside the 'struct' so that it is structurally more beautiful.
// But for better readability of the code, i.e. to reduce a lot of 'struct name -> data',
// I chose to put the data outside the 'stuct'.

struct monitor{
        // int buffer_free;
        // int buffer[NUM_PLACES];
        pthread_mutex_t mutex;
        pthread_cond_t consume_cond, produce_cond;
};

static void init_monitor(struct monitor *act)
{
        buffer_free = NUM_PLACES;
        pthread_mutex_init(&act->mutex, 0);
        pthread_cond_init(&act->consume_cond,0);
        pthread_cond_init(&act->produce_cond,0);

        //init buffer with -1
        for (int t = 0; t < NUM_PLACES; t++){
                buffer[t] = -1;
        }
        printf("init buffer with:\n");
        for (int i = 0; i < NUM_PLACES; i++)
                {
                        printf("%d buffer: %d\n", i, buffer[i]);
                }

}

static void _produce(struct monitor *act, int i, void *threadid)
{
        pthread_mutex_lock(&act->mutex);

        // 'buffer is full'
        while( buffer_free == 0 ){
                pthread_cond_wait(&act->produce_cond, &act->mutex);
        }

        // crit.
        buffer[last] = i;
        printf("Producer %ld puts %d into buffer at place %d \n", (long) threadid, buffer[last], last);
        last++;
        buffer_free --;
```

```
61          for (int i = 0; i < NUM_PLACES; i++)
62          {
63                  printf("%d buffer: %d\n", i, buffer[i]);
64          }


67          pthread_cond_signal(&act->consume_cond);
68          pthread_mutex_unlock(&act->mutex);
69  }

71  static void _remove(struct monitor *act, int amount, void *threadid)
72  {
73          pthread_mutex_lock(&act->mutex);

75          // check 'enough goods in one time ?'
76          while( (buffer_free + amount) > NUM_PLACES ){
77                  printf("not enough goods in one time, waiting...\n");
78                  pthread_cond_wait(&act->consume_cond, &act->mutex);
79          }

81          // crit.
82          for (int i = 0; i < amount; i++)
83          {
84                  printf ("Consumer %ld takes %d from buffer at place %d \n", (long) threadid
        , buffer[last-1], last-1);
85                  buffer[last-1] = -1;
86                  fflush (stdout);
87                  last--;
88                  buffer_free ++;
89          }

91          for (int i = 0; i < NUM_PLACES; i++)
92          {
93                  printf("%d buffer: %d\n", i, buffer[i]);
94          }

96          pthread_cond_signal(&act->produce_cond);
97          pthread_mutex_unlock(&act->mutex);
98  }

100 void* Producer (void *threadid);

102 void* Consumer (void *threadid);

104 struct monitor act;

106 int main(){
107         // init
108         pthread_t threads[NUM_THREADS];
109         int rc;
110         long t;

112         init_monitor(&act);

114         // creating threads
115         for(t=0; t < NUM_THREADS; t++) {
116                 if (t < NUM_PRODUCER){
117                         printf ("Creating Producer with ID: %ld\n", t);
118                         rc = pthread_create (&threads[t], NULL, Producer, (void *)t);
119                 }
120                 else{
121                         printf ("Creating Consumer with ID: %ld\n", t);
122                         rc = pthread_create (&threads[t], NULL, Consumer, (void *)t);
123                 }
124                 if (rc){
125                         exit (-1);
126                 }
127         }
```

```
129         // joining threads
130         for(t=0; t < NUM_THREADS; t++) {
131                 pthread_join (threads[t], NULL);
132         }
133         pthread_exit(NULL);

135         return 0;
136 }

138 void* Producer (void *threadid)
139 {
140         int i;
141         for (i= 0; i < 10; i++) {
142                 _produce(&act, i, threadid);

144         }
145         pthread_exit (NULL);
146 }

148 void* Consumer (void *threadid)
149 {
150         while (1) {
151                 // Here can choose the amount of goods to take from the buffer at a time.
152                 int amount = 2;
153                 _remove(&act, amount, threadid);
154         }
155         pthread_exit (NULL);
156 }
```

Die Anzahl der auf einmal aufzunehmenden Waren kann über ”amount” in der Funktion ”Consumer”beliebig ($0 < amount <= NUM\_PLACES$) eingestellt werden.
Vor jeder Abholung wird geprüft, ob genügend Waren vorhanden sind. Wenn nicht, wird gewartet, bis sich genügend Waren im Buffer befinden.(pthread_cond_wait(act->consume_cond, act->mutex);)

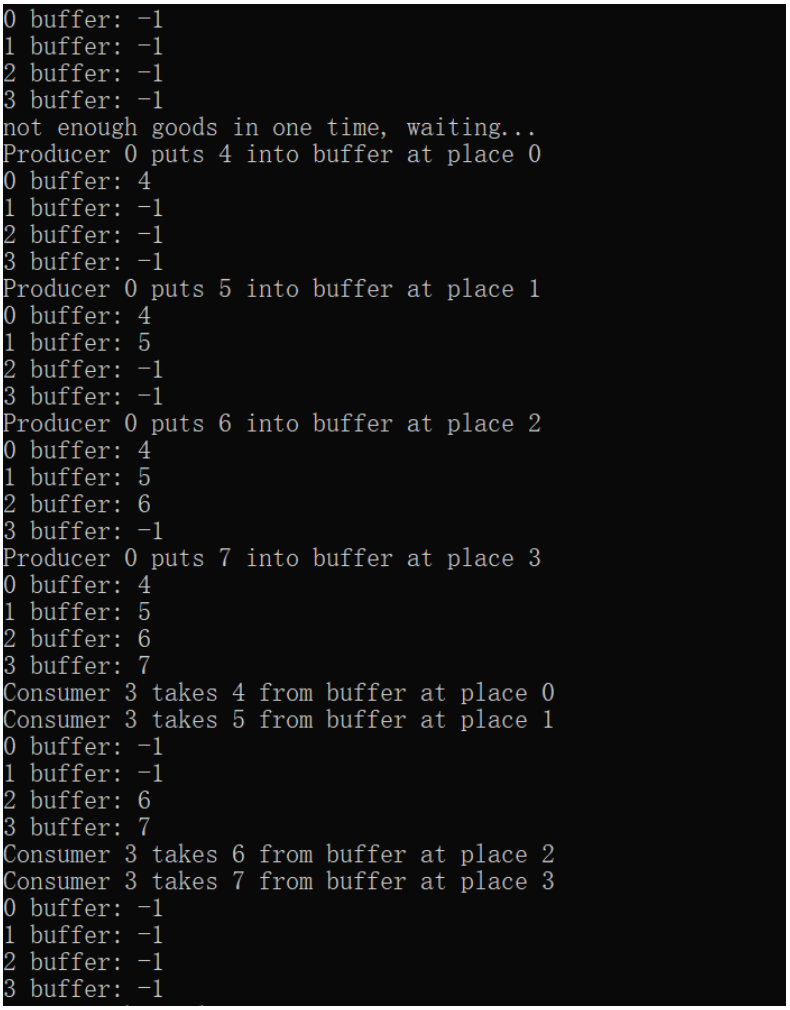

Abbildung 3: short example with amount = 2

✓

Literatur

30/30

Sehr schön :)