Barry Linnert

Nichtsequentielle und verteilte Programmierung, SS2021

Übung 5

Tutor: Florian Alex
Tutorium 3

Rui Zhao, William Djalal, Simeon Vasilev

22. Mai 2021

1 Produktion in C (12 Punkte)

Implementieren Sie in C eine Simulation des Problems der Produzent*innen und Konsument*innen. Die jeweilige Anzahl der beteiligten Produzent*innen bzw. Komsument*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.

Der Quellcode ist ausgerichtet, aber nach dem Hochladen des Codes auf LaTeX tritt eine gewisse Fehlausrichtung auf. Wir können den Grund dafür nicht finden. Der hochgeladene Quellcode ist schöner.

Sieht doch ok aus?

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

#define NUM_THREADS 10
#define NUM_PLACES 3
#define NUM_PRODUCER 6
//here choose the number of Producers as man like
//and the rest are Consumers (which is NUM_THREADS - NUM_PRODUCER)

sem_t empty;  // amount data in buffer
sem_t full;  // free places in buffer
sem_t mutex;  // critical section

int last;
int buffer[NUM_PLACES];

void* Producer (void *threadid)
{
  int i;
  for (i= 0; i < 10; i++) {
   sem_wait(&full);
   sem_wait(&mutex);
   buffer[last] = i;
   printf("Producer %ld puts %d into buffer at place %d \n", (long) threadid, buffer[last],
     last);
   last++;
   for (int i = 0; i < NUM_PLACES; i++)
```

```
    {
     printf("%d buffer: %d\n", i, buffer[i]);
    }
    sem_post(&mutex);
    sem_post(&empty);
  }
  pthread_exit (NULL);
}

void* Consumer (void *threadid)
{
  while (1) {
    sem_wait(&empty);
    sem_wait(&mutex);
    printf ("Consumer %ld takes %d from buffer at place %d \n", (long) threadid, buffer[last
     -1], last-1);
    buffer[last-1] = -1;
    fflush (stdout);
    last--;
    for (int i = 0; i < NUM_PLACES; i++)
    {
     printf("%d buffer: %d\n", i, buffer[i]);
    }
    sem_post(&mutex);
    sem_post(&full);
  }
  pthread_exit (NULL);
}

int main (/*int argc, char *argv[]*/)
{
 // init
 pthread_t threads[NUM_THREADS];
 int rc;
 long t;

 // init semaphores
 sem_init(&empty, 0, 0);
  sem_init(&full, 0, NUM_PLACES);
  sem_init(&mutex, 0, 1); // crit. sec.

  for (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]);
   }

 // creating threads
 for(t=0; t < NUM_THREADS; t++) {
  if (t < NUM_PRODUCER){
   printf ("Creating Producer with ID: %ld\n", t);
   rc = pthread_create (&threads[t], NULL, Producer, (void *)t);
  }
   else{
    printf ("Creating Consumer with ID: %ld\n", t);
    rc = pthread_create (&threads[t], NULL, Consumer, (void *)t);
   }
   if (rc){
    exit (-1);
   }
  }

 // joining threads
  for(t=0; t < NUM_THREADS; t++) {
   pthread_join (threads[t], NULL);
```

```
  }
 pthread_exit(NULL);

 // release semaphores
  sem_destroy(&mutex);
  sem_destroy(&full);
  sem_destroy(&empty);

  return 0;
}
```

```
zhaor@6P0ONT1-88WD3YW:~/alp4/u5$ gcc -std=c11 -Wall -Wextra -pedantic -pthread a1.c -o a1
zhaor@6P0ONT1-88WD3YW:~/alp4/u5$ ./a1
init buffer with:
0 buffer: -1
1 buffer: -1
2 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
Producer 0 puts 1 into buffer at place 1
0 buffer: 0
1 buffer: 1
2 buffer: -1
Creating Consumer with ID: 2
Producer 1 puts 0 into buffer at place 2
0 buffer: 0
1 buffer: 1
2 buffer: 0
Creating Consumer with ID: 3
Consumer 2 takes 0 from buffer at place 2
0 buffer: 0
1 buffer: 1
2 buffer: -1
Consumer 2 takes 1 from buffer at place 1
0 buffer: 0
1 buffer: -1
2 buffer: -1
Consumer 3 takes 0 from buffer at place 0
0 buffer: -1
1 buffer: -1
2 buffer: -1
Producer 1 puts 1 into buffer at place 0
0 buffer: 1
1 buffer: -1
2 buffer: -1
Consumer 2 takes 1 from buffer at place 0
0 buffer: -1
1 buffer: -1
2 buffer: -1
```

Abbildung 1: short example with 2 Producer 2 Consumer

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'

✓

12/12

Die IDs 0-5 gehören zu den 'Producer'
Die IDs 6-9 gehören zu den 'Consumer'

2 Optimierte Produktion (8 Punkte)

Erweitern Sie Ihre Lösung der Aufgabe 1 so, dass es nie zu einem Warten eines produzierenden Threads auf Grund eines (vollständig) gefüllten Puffers kommen kann.

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

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

sem_t empty;  // amount data in buffer
sem_t full;  // free places in buffer
sem_t mutex;  // critical section

int last;
int *buffer;
int buffer_free = 1;
int buffer_size = 1;
//initialize buffer size, if not enough, then enlarge it later


void* Producer (void *threadid)
{
  int i;
  for (i= 0; i < 2; i++) {
   sem_wait(&mutex);


   if (buffer_free <= 0)
   {
    buffer_size = buffer_size * 2;
    // double the buffer space, when not enough
    buffer = realloc(buffer, sizeof(int) * buffer_size);
    // initialize new buffer value with -1
    for (int t = buffer_size / 2 ; t < buffer_size; t++){
     buffer[t] = -1;
    }
    buffer_free = buffer_size / 2;

    printf("---new buffer, with size: %d:\n", buffer_size);
    for (int i = 0; i < buffer_size; i++)
     {
      printf("%d buffer: %d\n", i, buffer[i]);
     }

   }
   else{
    buffer_free --;
   }

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

```
    last++;
    buffer_free --;
    for (int i = 0; i < buffer_size; i++)
    {
     printf("%d buffer: %d\n", i, buffer[i]);
    }

    sem_post(&mutex);
    sem_post(&empty);

   }
   pthread_exit (NULL);
}

void* Consumer (void *threadid)
{
   while (1) {
    sem_wait(&empty);
    sem_wait(&mutex);
    printf ("Consumer %ld takes %d from buffer at place %d \n", (long) threadid, buffer[last
     -1], last-1);
    buffer[last-1] = -1;
    fflush (stdout);
    last--;
    for (int i = 0; i < buffer_size; i++)
    {
     printf("%d buffer: %d\n", i, buffer[i]);
    }
    buffer_free ++;
    sem_post(&mutex);

   }
   pthread_exit (NULL);
}

int main (/*int argc, char *argv[]*/)
{
 // init
 pthread_t threads[NUM_THREADS];
 int rc;
 long t;

 // init semaphores
 sem_init(&empty, 0, 0);
  sem_init(&mutex, 0, 1); // crit. sec.

  //initialize buffer, if not enough, then enlarge it later
  buffer = malloc(sizeof(int) * buffer_size);

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

 // creating threads
 for(t=0; t < NUM_THREADS; t++) {
  if (t < NUM_PRODUCER){
   printf ("Creating Producer with ID: %ld\n", t);
   rc = pthread_create (&threads[t], NULL, Producer, (void *)t);
  }
   else{
    printf ("Creating Consumer with ID: %ld\n", t);
    rc = pthread_create (&threads[t], NULL, Consumer, (void *)t);
```

```
122     }
123     if (rc){
124      exit (-1);
125     }
126    }

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

134   // release semaphores
135    sem_destroy(&mutex);
136    sem_destroy(&empty);

138    free(buffer);

140    return 0;
141 }
```

```
zhaor@6P0ONT1-88WD3YW:~/alp4/u5$ gcc -std=c11 -Wall -Wextra -pedantic -pthread a2.c -o a2
zhaor@6P0ONT1-88WD3YW:~/alp4/u5$ ./a2
---init buffer, with size: 1:
0 buffer: -1
Creating Producer with ID: 0
Creating Producer with ID: 1
Producer 0 puts 0 into buffer at place 0
0 buffer: 0
Creating Consumer with ID: 2
---new buffer, with size: 2:
0 buffer: 0
1 buffer: -1
Producer 0 puts 1 into buffer at place 1
0 buffer: 0
1 buffer: 1
---new buffer, with size: 4:
0 buffer: 0
1 buffer: 1
2 buffer: -1
3 buffer: -1
Producer 1 puts 0 into buffer at place 2
0 buffer: 0
1 buffer: 1
2 buffer: 0
3 buffer: -1
Producer 1 puts 1 into buffer at place 3
0 buffer: 0
1 buffer: 1
2 buffer: 0
3 buffer: 1
Consumer 2 takes 1 from buffer at place 3
0 buffer: 0
1 buffer: 1
2 buffer: 0
3 buffer: -1
Consumer 2 takes 0 from buffer at place 2
0 buffer: 0
1 buffer: 1
2 buffer: -1
3 buffer: -1
Consumer 2 takes 1 from buffer at place 1
0 buffer: 0
1 buffer: -1
2 buffer: -1
3 buffer: -1
Consumer 2 takes 0 from buffer at place 0
0 buffer: -1
1 buffer: -1
2 buffer: -1
3 buffer: -1
```

Abbildung 2: short example with 2 Producer 1 Consumer

Die Idee hier ist, den 'buffer' einfach zu verdoppeln, wenn nicht genügend Speicherplatz vorhanden ist. (Verdopplung dient zur Verbesserung der Effizienz. Wenn jedes Mal nur wenig Platz schaffen, führt dies zu wiederholten 'realloc'.)
Auf diese Weise hört der ProduzentInnen nie auf zu produzieren.

✓

8/8

3 Bewertung der Lösungen (10 Punkte)

Bewerten Sie Ihre Lösungen aus Aufgabe 1 und 2 in Hinblick auf die Anforderungen zur Sicherung des kritischen Abschnitts.

- Der kritischen Abschnitt ist durch mutual exclusion (Semaphore Mutex) zuverlässig geschützt. Durch Semaphore Mutex kann nur ein Thread auf dem kritischen Abschnitt operieren, z. B. Änderungen am Puffer vornehmen, Elemente produzieren oder konsumieren usw. ✓
- Die Lösung wird in höheren Programmiersprachen verwendet. ✓
- Diese Lösung führt nicht zu Deadlocks, da die von verschiedenen Threads benötigten die kritischen Ressourcen eindeutig sind, d. h. Operationen auf dem Speicherplatz (Puffer). Die zweite Bedingung für das Auftreten von Deadlocks ist nicht erfüllt. ? ok... ✓

Alle Anforderungen zur Sicherung des kritischen Abschnitts sind erfüllt.

weitere Anforderungen: kleiner Overhead und Fairness 6/10

Literatur

26/30