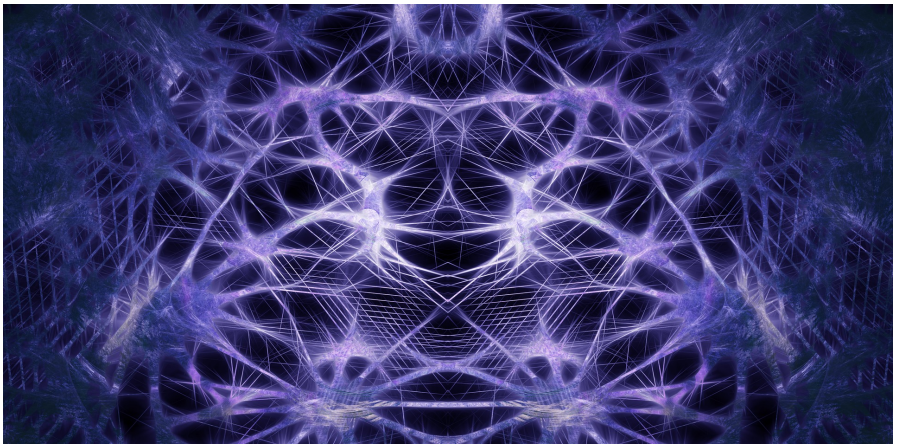

Übung 05

Nichtsequentielle und Verteilte Programmierung

Sebastian Faase, Lutz Schäfer

Tutorium Leon Dirmeier

Freie Universität Berlin

23. Mai 2019

I Produktion in C (12 Punkte)

Um unsere Implementierung mit den Vorgaben dieser Aufgabe zu starten, nutzten wir

```
./sim -p 5 -c 5
```

alternativ auch

```
./sim
```

da die Parameter als *default* gewählt wurden.

Die Simulation hat einige Kommandozeilenparameter, die wir in der Simulation der World behandeln [I.1]. In der Welt erstellen wir ein Warehouse und in diesem wird ein Storage initialisiert. [I.2]. Anschliessend werden die Producer und Consumer erzeugt. Der Einfachheit halber ist die Funktionalität der produceItem [I.4] und consumeItem [I.7] einfach gehalten. Die Funktionen putItemIntoBuffer [I.5] und removeItemFromBuffer [I.8] werden durch die Semaphoren *fillCount*, zur Anzeige der besetzten Stellen und *emptyCount*, zur Anzeige der freien Stellen geschützt.

Da unsere Ausgabe (auf einer Intel(R) Core(TM) m7-6Y75 CPU) zig tausende Zeilen pro Sekunde erzeugt, sei hier nur ein Auszug der Ausgabe Wiedergegeben [I.9].

Listing I.1: simulation.c

```c
int main(int argc, char *argv[])
{
  int help_is_set = 0;
  parameter param = {.producer_count = 5,
                     .consumer_count = 5,
                     .flag = 0}; // flag = 0 <-> false, flag = 1 <-> true


  /* parsing commandline arguments */
  int64_t opt;
  while ( -1 != (opt = getopt(argc, argv, "hdp:c:")) )
  {
    switch (opt) {
      case 'h':
        printhelp(stdout, argv[0]);
        help_is_set = 1;
        break;
      case 'd':
        param.flag=1;
        break;
      case 'p':
        param.producer_count = atoi(optarg);
        /* if there is no philos or we are in an alternete universe abort */
        break;
      case 'c': /* this serves as a timeout */
        param.consumer_count = atoi(optarg);
        break;
      default: /* .?. */
        printhelp(stderr, argv[0]);
        exit(EXIT_FAILURE);
```

```
    }
  }


  /* if no help shall be displayed, start the world */
  if (!help_is_set)
  {
    run(&param);
  }

  exit(EXIT_SUCCESS);
}
```

Listing I.2: world.c⟶run

```
int run(parameter * param)
{
  size_t  BUFFERLEN = 8;


  pthread_t producer_threads[param->producer_count];
  pthread_t consumer_threads[param->consumer_count];

  p_warehouse_t warehouse = malloc(sizeof(warehouse_t));
  warehouse->flag = param->flag;

  warehouse->storage = calloc(BUFFERLEN, sizeof(char) );
  for (size_t i = 0; i < BUFFERLEN; i++) { warehouse->storage[i] = '_'; }
  warehouse->length = BUFFERLEN;
  warehouse->p_index = 0;
  warehouse->c_index = 0;
  pthread_mutex_init(&warehouse->p_index_l, NULL);
  pthread_mutex_init(&warehouse->c_index_l, NULL);

  sem_init(&warehouse->fillCount , 0, 0);           // items produced
  sem_init(&warehouse->emptyCount, 0, BUFFERLEN);   // remaining space


  for (size_t i = 0; i < param->producer_count; i++)
  {
    if ( 0 != (pthread_create(&producer_threads[i],
                              NULL,
                              producer,
                              (void*) warehouse)))
    {
      /* abort if not all threads can be created */
      perror("cannot create pthread");
      exit(EXIT_FAILURE);
    }
  }
```

```
  for (size_t i = 0; i < param->consumer_count; i++)
  {
    if ( 0 != (pthread_create(&consumer_threads[i],
                              NULL,
                              consumer,
                              (void*) warehouse)))
    {
      /* abort if not all threads can be created */
      perror("cannot create pthread");
      exit(EXIT_FAILURE);
    }
  }

  for (size_t i = 0 ; i < param->producer_count ; i++)
  {
    pthread_join(producer_threads[i], NULL);
  }

  for (size_t i = 0 ; i < param->consumer_count ; i++)
  {
    pthread_join(consumer_threads[i], NULL);
  }

  exit(EXIT_SUCCESS);
}
```

Listing I.3: prosumer.c⟶produce

```
void * producer(void * args)
{
  warehouse_t* warehouse = (warehouse_t*)  args;
  setvbuf(stdout, NULL, _IOLBF, 0);
  int flag = 1;
  while (flag)
  {
    produceItem();
    /* blocked ??? */
    if (warehouse->flag )
    {
      if (-1 == sem_trywait(&warehouse->emptyCount) )
      {
        printf("Opps...\n");
        continue;
      }
    } else {
    /* wait until free space to write */
    sem_wait(&warehouse->emptyCount);
    }
```

```
    putItemIntoBuffer(warehouse);
    sem_post(&warehouse->fillCount);
    printbuffer(warehouse->storage, warehouse->length);
  }
  return NULL;
}
```

Listing I.4: prosumer.c⟶produceItem

```
void produceItem()
{
  printf("produce Item\n");
}
```

Listing I.5: prosumer.c⟶putItemIntoBuffer

```
void putItemIntoBuffer(warehouse_t * warehouse)
{
  /* lock index */
  pthread_mutex_lock(&warehouse->p_index_l);
  warehouse->storage[warehouse->p_index] = 'x';
  warehouse->p_index = ( warehouse->p_index + 1 ) % warehouse->length;
  pthread_mutex_unlock(&warehouse->p_index_l);
}
```

Listing I.6: prosumer.c⟶consumer

```
void * consumer(void * args)
{
  warehouse_t* warehouse = (warehouse_t*)  args;
  setvbuf(stdout, NULL, _IOLBF, 0);
  int flag = 1;
  while (flag)
  {
    /* wait until something is in store */
      sem_wait(&warehouse->fillCount);
      removeItemFromBuffer(warehouse);
      /* announce more space */
      sem_post(&warehouse->emptyCount);
      consumeItem();
    printbuffer(warehouse->storage, warehouse->length);
  }
  return NULL;
}
```

Listing I.7: prosumer.c⟶consumeItem

```
void consumeItem()
{
  printf("consume Item\n");
}
```

Listing I.8: prosumer.c⟶removeItemFromBuffer

```
void removeItemFromBuffer(warehouse_t * warehouse)
{
  /* lock index */
  pthread_mutex_lock(&warehouse->c_index_l);
  warehouse->storage[warehouse->c_index] = '_';
  warehouse->c_index = ( warehouse->c_index + 1 ) % warehouse->length;
  pthread_mutex_unlock(&warehouse->c_index_l);
}
```

Listing I.9: output1

```
produce Item
[ x x x x x x x x ]
produce Item
consume Item
[ x x x x x x x x ]
consume Item
[ x x x x x x x x ]
consume Item
[ x x x x x _ x x ]
consume Item
[ x x x x x _ _ x ]
consume Item
[ x x x x x _ _ _ ]
consume Item
[ _ x x x x _ _ _ ]
consume Item
[ _ _ x x x _ _ _ ]
consume Item
[ _ _ _ x x _ _ _ ]
consume Item
[ _ _ _ _ x _ _ _ ]
consume Item
[ _ _ _ _ _ _ _ _ ]
[ _ _ _ _ _ x _ _ ]
produce Item
[ _ _ _ _ _ _ x _ ]
produce Item
[ _ _ _ _ _ _ x x ]
[ _ _ _ _ _ _ x x ]
produce Item
[ x _ _ _ _ _ x x ]
produce Item
[ x x _ _ _ _ x x ]
produce Item
[ x x x _ _ _ x x ]
produce Item
[ x x x x _ _ x x ]
produce Item
```

```
[ x x x x x _ x x ]
produce Item
[ x x x x x x x x ]
produce Item
consume Item
[ x x x x x x x x ]
consume Item
[ x x x x x x x x ]
consume Item
[ x x x x x x x _ ]
consume Item
[ _ x x x x x x _ ]
consume Item
[ _ _ x x x x x _ ]
consume Item
[ _ _ _ x x x x _ ]
```

II Optimierte Produktion (8 Punkte)

Wir verwenden zur Lösung dieser Aufgabe in unserer Implementierung die Funktion *sem_trywait* anstelle von *sem_wait* [II.1]. Diese Funktion blockt den aufrufenden Prozess nicht.

Listing II.1: prosumer.c⟶produce

```
if (warehouse->flag )
{
  if (-1 == sem_trywait(&warehouse->emptyCount) )
  {
    printf("Opps...\n");
    continue;
  }
} else {
/* wait until free space to write */
sem_wait(&warehouse->emptyCount);
}
```

III Bewertung der Lösung (10 Punkte)

Unsere Lösung hat zwei kritische Abschnitte. Der zur Verfügung stehende Speicherplatz (Ringbuffer) wird durch Semaphoren geschützt. Die Implementierung der ersten Aufgabe blockiert dabei den aufrufenden Thread falls kein Speicherplatz zur Verfügung steht. Die Implementierung der zweiten Aufgabe blockiert an dieser Stelle nicht und verzichtet auf den Schreibprozess. Der zweite kritische Abschnitt ist das Lesen und Schreiben der Indexe des Ringbuffers. Diese werden durch Mutexe geschützt.