BSP A2: Erzeuger/Verbraucher-Kooperation

Das zu entwickelnde Programm soll laufen, bis über die Tastatur ein „q“ oder „Q“ eingegeben wird. Implementieren Sie zu diesem Zweck ein Programm, das aus fünf Threads und einem Ringpuffer besteht:

Thread-Entwurf

Main_Thread:
Der Root Thread initialisiert das gesamte System: die Threads, die **Mutex-Variablen** und die **Condition-Variablen**. Anschließend geht er in den Zustand “blocked“ über und wartet, bis die von ihm erzeugten Threads terminiert sind. Bei Programmende wird das gesamte System terminiert (Threads und Mutex löschen).

Producer_1:
Dieser Thread erzeugt zyklisch alle 3 Sekunden ein Zeichen, das er in einen Ringpuffer schreibt. Er nimmt zyklisch jeweils das nächste Zeichen aus dem String “abcd ... wxyz“. Ist der Puffer voll, blockiert er bis wieder Speicherplatz im Puffer frei ist. Während dieser Wartezeit ist er im Zustand “blocked“. Producer_1 gibt auf dem Bildschirm aus, dass er ein Zeichen in den Puffer geschrieben hat.

Producer_2:
Dieser Thread ist analog zum Producer_1 Thread aufgebaut. Im Unterschied zu Producer_1 schreibt er Großbuchstaben in den Puffer.

Consumer:
Dieser Thread nimmt alle 2 Sekunden ein Zeichen aus dem Ringpuffer und gibt es auf dem Bildschirm aus. Ist der Puffer leer, wartet er im Zustand “blocked“, bis ein neues Zeichen im Puffer liegt. Hier ist ein fortlaufender Ausgabestrom gewünscht, der die Zeichen nebeneinander ausgibt – erst nach 30 Zeichen soll ein Zeilenwechsel erfolgen !

Control:
Dieser Thread steuert das Erzeuger - Verbraucher System mit Tastatureingaben:
– TE 1: Start / Stopp Producer_1
– TE 2: Start / Stopp Producer_2
– TE c oder C: Start / Stopp Consumer
– TE q oder Q: Terminiert die Threads, sodass der Main_Thread das System beendet.
– TE h: Liefert diese Liste von möglichen Eingaben.
– Alle anderen Tastatureingaben werden ignoriert.

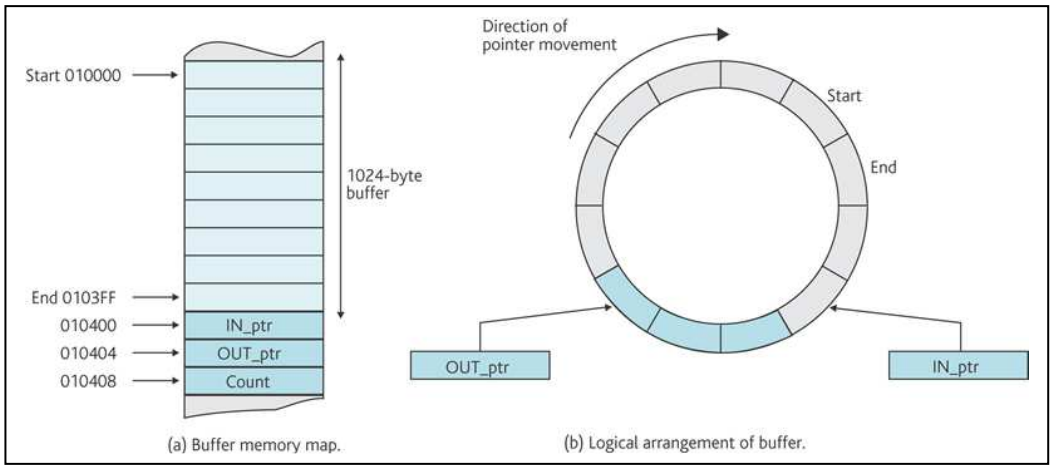
Anforderungen an die Lösung:

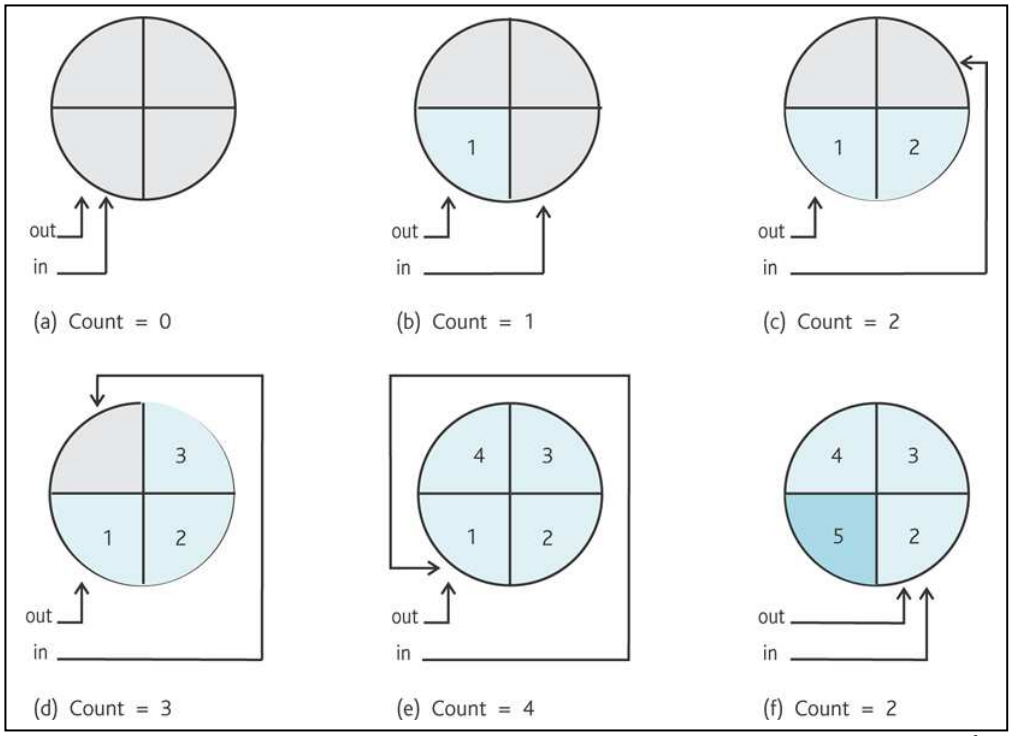
- Die Synchronisation ist so zu realisieren, dass ein Thread, der auf ein Ereignis wartet, im Zustand “blocked“ ist, d.h. Polling ist keine Lösung.
- Ein Thread, der über den Control Thread angehalten wird, blockiert, bis er erneut gestartet wird, d.h. auch hier ist Polling nicht erlaubt.
- Stellen Sie für jeden Thread FIFO-Scheduling und eine explizite Priorität ein.
- Geben Sie angemessene Fehlermeldungen aus. Greifen Sie dazu auf die Fehlerrückgabewerte der Funktionen zurück, die über errno bzw. gemäß POSIX Standard bereitgestellt werden.
- Schauen Sie die Systemcalls an, die während eines Ablaufs Ihres Programms auftreten.

Zugriffauf den Ringpuffer

- Lesen: Liefert das älteste nicht gelesene Zeichen zurück (FIFO-Prinzip). Sind keine ungelesenen Zeichen im Ringpuffer, soll die aufrufende Thread in den Zustand 'warten' gehen, bis ein Zeichen geschrieben worden ist.

- Schreiben: Schreibt ein Zeichen in den Ringpuffer. Die aufrufende Thread soll in den Zustand 'warten' gehen, wenn der Ringpuffer voll ist, bis durch die Operation 'lesen' Platz im Ringpuffer geschaffen worden ist.
- Ringpuffer angelegt in einem linearen Speicher mit oberer und unterer Begrenzung. Der Lese- sowie der Schreibpointer, das Daten-Array und die Zählvariable sollen in einer Struktur zusammengefasst werden. Ist der Ringpuffer bis zur oberen Adresse gefüllt, so kann der Schreibvorgang an der unteren Adresse fortgesetzt werden, ***sofern dort keine nicht gelesenen Zeichen gespeichert sind.***
- Die Zählvariable gibt den Füllstand mit nicht gelesenen Zeichen an. Die Betriebsituationen sind mit einfachen Grafiken für einen überschaubaren Ringpuffer zu erläutern.
- Kommentieren Sie Teilbild (f).





[A. Clements: Principles of Computer Hardware. Oxford University Press 4th edition 2006.]

1. Nutzen Sie den folgenden Rumpf-Code entsprechend der Aufgabenstellung.
2. Geben Sie vorab eine Art von Kommunikationsdiagramm an, dass die Threads, den FIFO und die Mutexe mit ihren Wechselwirkungen incl. der Terminierungssignale darstellt.
3. Erstellen Sie eine Zugehörigkeitstabelle zu den Mutexe, den Threads und den Condition-Variablen sowie den Testkriterien.
4. Formulieren Sie Erläuterungen zur Pointer-Semantik.
5. Fassen Sie Bebachtungen zur Bildschirmausgabe zusammen.

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#define MAX 16

pthread_mutex_t rb_mutex = PTHREAD_MUTEX_INITIALIZER;
…
pthread_cond_t not_empty_condvar = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full_condvar = PTHREAD_COND_INITIALIZER;
…
int thread_id[4] = {0,1,2,3};

typedef struct {
      int buffer[MAX];
      int *p_in;
      int *p_out;
      int count;
}rb;
rb x = { {0}, NULL, NULL, 0};
rb *p_rb = &x;

#define  p_start (int *)(p_rb -> buffer)
#define  p_end (int *)((p_rb -> buffer) + MAX-1)

void* p_1_w(void *pid);
void* p_2_w(void *pid);
void* consumer(void *pid);
void* control(void *pid);

int main(int argc, char* argv[])
{      int i;
      pthread_t threads[4];
      printf("Start des Beispiels \n");
      //printf("Argumente verfuegbar: ARGC\n", 3*argc);
…
      p_rb -> p_in = p_start;
      p_rb -> p_out = p_start;
      p_rb -> count = 0;
      printf("Counter value %d\n", p_rb ->count);
      pthread_create(&threads[0], NULL, …, (void *)thread_id);
      pthread_create(&threads[1], NULL, …, (void *)&thread_id[1]);
      pthread_create(&threads[2], NULL, …, (void *) &thread_id[2]);
…
      for(i = 0; i<4; i++)
      pthread_join(threads[i], NULL);
      printf("Ende nach Join der Threads\n");
      return 0;
}
```

```c
void* write_c(void *pid)
{        int i = 0;
         int z_var = 0;
// kein static erforderlich, da jeder Thread seinen eigenen
Kontext hat
printf("Start Schreiben; %d: \n", *(int*)pid);
while(1)
{
i++;
z_var++;
     if (z_var > 9) z_var = 0;
     pthread_mutex_lock(&rb_mutex);
     while(p_rb -> p_in == p_rb -> p_out && p_rb ->count ==MAX)
     {
          printf("Full: wait\n");
          pthread_cond_wait(&not_full_condvar, &rb_mutex);
          printf("Aufgewacht: count %d, Thread_Nr. %d\n",
                                        p_rb -> count,
                                        *(int*)pid);
          }
          *(p_rb -> p_in) = z_var;
          (p_rb -> p_in)++ ;
          if((p_rb -> p_in) > p_end)
               {
                    p_rb -> p_in = p_start;
               }
          (p_rb -> count)++;
          if(p_rb -> count != 0)
             {
               printf("Buffer nicht leer, signalisiert\n");
               pthread_cond_signal(&not_empty_condvar);
               }
          pthread_mutex_unlock(&rb_mutex);
          sleep(1);
          }
     return (NULL);
}
```

```
void* read_rb(void *pid)
{    int i = 0;
     printf("Start Lesen; %d: \n", *(int*)pid);
     while(1)
     {
          i++;
          pthread_mutex_lock(&rb_mutex);
          while(p_rb -> count == 0)
          {
          printf("Empty: wait\n");
          pthread_cond_wait(&not_empty_condvar, &rb_mutex);
          printf("Aufgewacht: count %d, Thread_Nr. %d\n",
                                        p_rb -> count,
                                   *(int*)pid);
          }
          (p_rb -> count)--;
printf("Ältestes Zeichen ausgeben: %d:\n", *(p_rb -> p_out));
          (p_rb -> p_out)++;
          if((p_rb -> p_out) > p_end)
               {
                    p_rb -> p_out = p_start;
               }
          if(p_rb -> count <= MAX)
               {
          printf("Buffer nicht voll, signalisiert\n");
                    pthread_cond_signal(&not_full_condvar);
               }
          pthread_mutex_unlock(&rb_mutex);
          sleep(1);
          }
          return (NULL);
}
```