Concurrent and Real-time Systems Exercise

Threads, Producer/Consumer, & Synchronization in C

March 2017

1 Objective of Task

The objective of this task is to implement a producer/consumer structure based on POSIX multithreading and synchronisation.

2 Specification

Build a multi-threading producer/consumer demonstration program in C/POSIX fulfilling the following requirements:

1. This program should be written in ISO C11[1].
2. The threading shall be based on the POSIX Pthreads library. The program shall work on a POSIX Pthreads-compliant operating system [1] [2].
3. The program will take as input on the command line 2 numbers, each number between 1 and 10 inclusive. The first of these numbers indicate the number of producer threads to create, the second of these numbers indicates the number of consumer threads to create.
4. The main program will create the appropriate number of each of the producer and consumer threads.
5. The program will have a global work queue —maybe in the form of a linked list— of size 20 and a global boolean variable as a flag to finish. Actions on this queue will be protected from concurrent collision. The work queue should be implemented as a global data structure (can be an struct or array or whatever the student decides).
6. The main program will parse the command line, create the appropriate number of producers and consumers, sleep for 30 seconds (note this is seconds, not milliseconds), then set the global finish flag telling the consumers and producers to quit. It will then join with the threads, print a finishing message, and exit.

[1] C11 (formerly C1X) is an informal name for ISO/IEC 9899:2011 [2]
[2] It is available online at http://pubs.opengroup.org/onlinepubs/9699919799/

7. The producer threads will generate a random number between 0 and 1000, sleep for that number of milliseconds, generate a second random number between 1 and 10 inclusive. They will then attempt to put this number on the work queue. If the queue is full, they will write a message indicating that the queue is full, otherwise they will write a message indicating what the number was they put on the queue and the size of the queue after they did so. This will be done in a continuous loop until the main program sets the finish flag, at which point they will exit.

8. The consumer threads will —in a continuous loop— sleep for a random number of milliseconds between 0 and 1000, and then perform a blocking receive[3] off the work queue. Once they pull the number, they will compute the factorial of that number, print a message indicating the original number, the factorial, and the size of the queue. They will continue this until the main program sets the finish flag, at which point they will exit.

9. Critical sections should be protected using mutexes.

10. Blocking calls should be implemented using calls `pthread_cond_wait()`, `pthread_cond_signal`, and `pthread_cond_broadcast()`

11. The program must be well-documented using C comments. There shall not be an accompanying explanatory text.

12. The C file shall include a comment describing the needs —e.e. libraries, platforms— and procedure to compile, link and execute the program.

13. Any other non-specified aspect of the program is free for the student to decide about.

3 Sample output

This is a potential example —i.e. non mandatory— of the output expected from the program:

```
Producer started
Producer started
Consumer started
Consumer started
Consumer started
Producer added 4 to queue, size = 1
Consumer removed 4, computed 4! = 24, queue size = 0
.
.
.
Consumer removed 3, computed 3! = 6, queue size=4
Main program exiting after joining threads
```

[3] A blocking receive means that the consumer waits passively until an item is available to consume.

4 Completing and uploading the task

The grade will be based on the quality of the program and the comments. This is individual work. The submitter may talk and discuss aspects with other students, but the programming work —and hence the C program— shall be individual.

The procedure to submit is as follows:

1. Write a single, well commented C file.
2. The file name shall be your name + your registration number separated by an hyphen (e.g. `Sanchez-02322.c`).
3. Check that it fulfils all requirements from Section 2.
4. Remember that it must contain compilation instructions.
5. Upload the C file to the course Moodle website.

A Appendix

This last section contains C examples or thread use.

A.1 Example of using POSIX threads

An example of using pthreads in C can be found here.

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

#define NUM_T1       4
#define NUM_T2       4

bool done = false;
int count = 0 ;
pthread_mutex_t count_mutex;

void *T1(void *t)
{
   int tid = *((int*)t);
   printf("Thread␣%d␣started.\n",tid);
        while(!done) {
                pthread_mutex_lock(&count_mutex);
                count++ ;
                printf("T1␣[%d]␣count␣=␣%d\n", tid, count);
                pthread_mutex_unlock(&count_mutex);
                sleep(1);
        }
   printf("T1␣thread␣%d␣done.\n",tid);
   pthread_exit(NULL);
}

void *T2(void *t)
{
   long tid = *((int*)t);
   printf("Thread␣%ld␣started.\n",tid);
        while(!done) {
                pthread_mutex_lock(&count_mutex);
                count-- ;
                printf("T2␣[%ld]␣count␣=␣%d\n", tid, count);
                pthread_mutex_unlock(&count_mutex);
                sleep(2);
        }
   printf("T2␣thread␣%ld␣done.\n",tid);
   pthread_exit(NULL);
```

```
}


int main (int argc, char *argv[])
{
   pthread_t thread[NUM_T1 + NUM_T2];
   pthread_attr_t attr;
   int t;
   void *status;
   int tid[1000];

  /* Initialize the tids */
   for (t = 0; t < 1000; t++)
     tid[t] = t;


   /* Initialize mutex and condition variable objects */
   pthread_mutex_init(&count_mutex, NULL);
   //pthread_cond_init (&count_cond, NULL);

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_T1; t++) {
      pthread_create(&thread[t], &attr, T1, (void *)&tid[t]);
   }

   for(t=0; t<NUM_T2; t++) {
      pthread_create(&thread[t+NUM_T1], &attr, T2, (void *)&tid[t]);
   }

   sleep(5);
   done = true ;
   /* Free attribute and wait for the other threads */
   for(t=0; t<NUM_T1+NUM_T2; t++)
      pthread_join(thread[t], &status);

   printf("Main: program completed. Exiting. Count = %d\n",count);
   /* Clean up and exit */
   pthread_attr_destroy(&attr);
   pthread_mutex_destroy(&count_mutex);
   //pthread_cond_destroy(&count_cond);
   pthread_exit(NULL);

}

Program execution example:

Thread 0 started.
```

```
T1 [0] count = 1
Thread 1 started.
Thread 3 started.
T1 [3] count = 2
T1 [1] count = 3
Thread 2 started.
T1 [2] count = 4
Thread 0 started.
T2 [0] count = 3
Thread 1 started.
T2 [1] count = 2
Thread 2 started.
T2 [2] count = 1
Thread 3 started.
T2 [3] count = 0
T1 [0] count = 1
T1 [3] count = 2
T1 [1] count = 3
T1 [2] count = 4
T2 [0] count = 3
T2 [1] count = 2
T1 [0] count = 3
T2 [3] count = 2
T2 [2] count = 1
T1 [1] count = 2
T1 [3] count = 3
T1 [2] count = 4
T1 [0] count = 5
T1 [1] count = 6
T1 [2] count = 7
T1 [3] count = 8
T2 [0] count = 7
T2 [1] count = 6
T2 [3] count = 5
T2 [2] count = 4
T1 [0] count = 5
T1 [1] count = 6
T1 [2] count = 7
T1 [3] count = 8
T1 thread 0 done.
T1 thread 1 done.
T1 thread 2 done.
T1 thread 3 done.
T2 thread 1 done.
T2 thread 2 done.
T2 thread 3 done.
T2 thread 0 done.
Main: program completed. Exiting. Count = 8
```

A.2 Example of passing structs

An example of passing malloc'd structs to a thread in C can be found here.

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

#define NUM_T1 4
#define NUM_T2 4

bool done = false;
int count = 0 ;
pthread_mutex_t count_mutex;

#define MSGLEN 32

struct thread_dat {
 int id ;
 char message[MSGLEN];
};

void *T1(void *t)
{
 struct thread_dat *pData = (struct thread_dat*)t ;

 printf("%s.\nThread␣%d␣started.\n", pData->message, pData->id);

 while(!done) {
  pthread_mutex_lock(&count_mutex);
  count++ ;
  printf("T1␣[%d]␣count␣=␣%d\n", pData->id, count);
  pthread_mutex_unlock(&count_mutex);
  sleep(1);
 }

 printf("T1␣thread␣%d␣done.\n",pData->id);
 pthread_exit(NULL);
}

void *T2(void *t)
{
 struct thread_dat *pData = (struct thread_dat*)t ;

 printf("%s.\nThread␣%d␣started.\n", pData->message, pData->id);

 while(!done) {
  pthread_mutex_lock(&count_mutex);
  count-- ;
```

```
  printf("T2 [%d] count = %d\n", pData->id, count);
  pthread_mutex_unlock(&count_mutex);
  sleep(2);
 }

 printf("T2 thread %d done.\n",pData->id);
 pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
 pthread_t thread[NUM_T1 + NUM_T2];
 pthread_attr_t attr;
 int t;
 void *status;
 struct thread_dat *pthreadData = malloc(sizeof(struct thread_dat) *
  (NUM_T1 + NUM_T2));

 if(!pthreadData) {
   fprintf(stderr, "Cannot allocate thread data!\r\n");
  return 255 ;
 }
 /* Initialize mutex and condition variable objects */
 pthread_mutex_init(&count_mutex, NULL);

 /* Initialize and set thread detached attribute */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for(t=0; t<NUM_T1; t++) {
  pthreadData[t].id = t ;
  snprintf(pthreadData[t].message, MSGLEN,
    "Welcome to thread %c", 'A'+t) ;
  pthread_create(&thread[t], &attr, T1, (void *)&pthreadData[t]);
 }

 for(t=0; t<NUM_T2; t++) {
  pthreadData[t+NUM_T1].id = t+NUM_T1 ;
  snprintf(pthreadData[t+NUM_T1].message, MSGLEN,
    "Welcome to thread %c", 'Z'-t) ;
  pthread_create(&thread[t+NUM_T1], &attr, T2,
    (void *)&pthreadData[t+NUM_T1]);
 }

 sleep(5);
 done = true ;

 /* Free attribute and wait for the other threads */
 for(t=0; t<NUM_T1+NUM_T2; t++)
 pthread_join(thread[t], &status);
```

```
 printf("Main: program completed. Exiting. Count = %d\n",count);

 /* Clean up and exit */
 if(pthreadData) free(pthreadData);
 pthread_attr_destroy(&attr);
 pthread_mutex_destroy(&count_mutex);
 //pthread_cond_destroy(&count_cond);
 pthread_exit(NULL);
}
```

Program execution example:

```
Welcome to thread A.
Thread 0 started.
T1 [0] count = 1
Welcome to thread B.
Thread 1 started.
T1 [1] count = 2
Welcome to thread C.
Thread 2 started.
T1 [2] count = 3
Welcome to thread D.
Thread 3 started.
T1 [3] count = 4
Welcome to thread Z.
Thread 4 started.
T2 [4] count = 3
Welcome to thread Y.
Thread 5 started.
T2 [5] count = 2
Welcome to thread X.
Thread 6 started.
T2 [6] count = 1
Welcome to thread W.
Thread 7 started.
T2 [7] count = 0
T1 [1] count = 1
T1 [3] count = 2
T1 [2] count = 3
T1 [0] count = 4
T2 [4] count = 3
T1 [0] count = 4
T1 [2] count = 5
T2 [5] count = 4
T2 [6] count = 3
T1 [3] count = 4
T1 [1] count = 5
T2 [7] count = 4
T1 [2] count = 5
T1 [3] count = 6
```

```
T1 [1] count = 7
T1 [0] count = 8
T2 [4] count = 7
T2 [6] count = 6
T2 [7] count = 5
T2 [5] count = 4
T1 [2] count = 5
T1 [3] count = 6
T1 [0] count = 7
T1 [1] count = 8
T1 thread 2 done.
T1 thread 3 done.
T1 thread 0 done.
T1 thread 1 done.
T2 thread 6 done.
T2 thread 7 done.
T2 thread 4 done.
T2 thread 5 done.
Main: program completed. Exiting. Count = 8
```

A.3 Example of using conditional waits

An example of using conditional waits and broadcasts in C can be found here.

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

#define NUM_T1          2
#define NUM_T2            4
#define MAX_COUNT 10

bool done = false;       //  Flag to indicate completion
int count = 0;         //  Count simulating buffer
pthread_mutex_t count_mutex;     //  Mutex variable
pthread_cond_t count_cond;     //  Conditional variable

void * T1 (void *t)
{
  int tid = * (int *)t;
  printf ("T1[%d] started\n", tid);
  while (!done)
    {
      pthread_mutex_lock (&count_mutex);
      if (count == MAX_COUNT)
  {
    printf ("T1[%d] Count already at max\n", tid);
  }
```

```
      else
  {
    count++;
          pthread_cond_signal (&count_cond);
    printf ("T1 [%d] count = %d\n", tid, count);
  }
      pthread_mutex_unlock (&count_mutex);
      sleep (1);
    }
  printf ("T1 [%d] thread done.\n", tid);
  pthread_exit (NULL);
}

void *
T2 (void *t)
{
  int tid = * (int *)t;
  printf ("T2 [%d] started\n", tid);
  while (!done)
    {
      pthread_mutex_lock (&count_mutex);
      if (count <= 0)
  {
    printf ("T2[%d] waiting for buffer\n", tid);
    pthread_cond_wait (&count_cond, &count_mutex);
          printf ("T2[%d] released from wait\n", tid);
  }
      if (!done)
  {
    count--;
    printf ("T2[%d] count = %d\n", tid, count);
  }
      else
  {
    printf ("T2[%d] Broadcast received with done flag set\n", tid);
  }
      pthread_mutex_unlock (&count_mutex);
      sleep (2);
    }
  printf ("T2[%d] thread done.\n", tid);
  pthread_exit (NULL);
}


int main (int argc, char *argv[])
{
  int thread_id [NUM_T1 + NUM_T2];
  pthread_t thread[NUM_T1 + NUM_T2];
  pthread_attr_t attr;
  int t;
```

```
  void *status;

  /* Initialize mutex and condition variable objects */

  pthread_mutex_init (&count_mutex, NULL);
  pthread_cond_init (&count_cond, NULL);

  /* Initialize and set thread detached attribute */

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);

  /* Spawn the threads */

  for (t = 0; t < NUM_T1; t++)
    {
      thread_id[t] = t;
      pthread_create (&thread[t], &attr, T1, &thread_id[t]);
    }

  for (t = 0; t < NUM_T2; t++)
    {
      thread_id[NUM_T1+t] = NUM_T1 + t;
      pthread_create (&thread[t + NUM_T1], &attr, T2, &thread_id[NUM_T1+t]);
    }

  sleep (5);
  done = true;

  /* Free attribute and wait for the other threads */

  pthread_cond_broadcast (&count_cond);

  for (t = 0; t < NUM_T1 + NUM_T2; t++)
    pthread_join (thread[t], &status);

  printf ("Main: program completed. Exiting. Count = %d\n", count);

  /* Clean up and exit */

  pthread_attr_destroy (&attr);
  pthread_mutex_destroy (&count_mutex);
  pthread_cond_destroy (&count_cond);
  pthread_exit (NULL);

}

Program execution example:

T1[0] started
```

```
T1[1] started
T1[0] count = 1
T2 [2] started
T2[2] count = 0
T1[1] count = 1
T2 [3] started
T2[3] count = 0
T2 [4] started
T2[4] waiting for buffer
T2 [5] started
T2[5] waiting for buffer
T1[1] count = 1
T2[4] released from wait
T2[4] count = 0
T1[0] count = 1
T2[5] released from wait
T2[5] count = 0
T2[2] waiting for buffer
T2[3] waiting for buffer
T1[1] count = 1
T2[2] released from wait
T2[2] count = 0
T1[0] count = 1
T2[3] released from wait
T2[3] count = 0
T2[4] waiting for buffer
T2[5] waiting for buffer
T1[1] count = 1
T2[4] released from wait
T2[4] count = 0
T1[0] count = 1
T2[5] released from wait
T2[5] count = 0
T2[2] waiting for buffer
T1[1] count = 1
T2[2] released from wait
T2[2] count = 0
T2[3] waiting for buffer
T1[0] count = 1
T2[3] released from wait
T2[3] count = 0
T2[4] thread done.
T1[1] thread done.
T1[0] thread done.
T2[5] thread done.
T2[2] thread done.
T2[3] thread done.
Main: program completed. Exiting. Count = 0
```

References

[1] IEEE. Std 1003.1-2008, 2016 Edition] Information Technology–Portable Operating System Interface (POSIX)– Part 1: System Application: Program Interface (API) [C Language]. International Standard, 2016.

[2] ISO/IEC. 9899:2011 - information technology - programming languages - C. International Standards, 2011.