Programming Assignment 4
Producer-Consumer Problem

Objective

The objective of this assignment is to provide a semaphore-based solution to the producer consumer problem using a bounded buffer.

Assignment: Using Threads and Mutex/Counting Semaphores

In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes. The solution uses three semaphores: *empty* and *full*, which count the number of empty and full slots in the buffer, and *mutex*, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for *empty* and *full* and a mutex *lock*, rather than a binary semaphore, to represent mutex. The producer and consumer running as separate threads will move items to and from a buffer that is synchronized with the *empty, full*, and *mutex* structures.

The Shared Buffer

Internally, the buffer will consist of a fixed-size array of type buffer item (which will be defined using a typedef). The array of *buffer_item* objects will be manipulated as a wrap-around buffer . The definition of buffer item, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef struct buffer_item {
     uint8_t  data[30];
     uint16_t cksum;
} BUFFER_ITEM;

#define BUFFER SIZE 10
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears below:

```
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
  /* insert item into buffer
   return 0 if successful, otherwise
   return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
  /* remove an object from buffer
   placing it in item
   return 0 if successful, otherwise
   return -1 indicating an error condition */
}
```

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the e`mpty` and `full` semaphores. The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

```
int main(int argc, char *argv[]) {
 /* 1. Get command line arguments argv[1],argv[2],argv[3] */
 /* 2. Initialize buffer */
 /* 3. Create producer thread(s) */
 /* 4. Create consumer thread(s) */
 /* 5. Sleep */
 /* 6. Exit */
}
```

The Producer and Consumer Threads

The producer thread(s) will create the data item(s) which includes the checksum and random data (produced using the `rand()` function, which produces random integers between 0 and *`RAND MAX`*). The consumer thread(s) read the shared memory buffer of `item(s),` calculate the checksum and compare that with the value stored in the shared memory buffer to ensure that the data did not get corrupted. An outline of the producer and consumer threads appears below:

```
void *producer(void *param) {
  buffer_item item;

  while (true) {
    /* release the CPU randomly to simulate preemption */
   if ((rand() % 100) < 40) {   // ~40% chance
          sched_yield();         // voluntarily yield CPU
   }
    /* generate the item*/
    if (insert_item(item))
      fprintf("report error condition");
}

void *consumer(void *param) {
  buffer_item item;

  while (true) {
   /* release the CPU randomly to simulate preemption */
   if ((rand() % 100) < 40) {   // ~40% chance
          sched_yield();         // voluntarily yield CPU
   }
    if (remove_item(&item))
      fprintf("report error condition");

   /* verify the item removed matches the item inserted */

}
```

(Note: where the pseudo-code states generating a random number create a data item with 30 random numbers – also don't forget to calculate a checksum).

The producer/consumer program (*prodcon.c*) takes three arguments from the command line (no prompting the user from within the program).

1. To start the *prodcon* program

   *./**prodcon** <delay> <#producer threads> <#consumer threads>*where the argument <*delay* > indicates how long to sleep (in seconds) before terminating and *#producer threads*> indicates number of threads and <*#consumer threads*> indicates the number of consumer threads.

After the sleep has expired in the main process it is to execute a thread cancellation since the producer and consumer threads are in infinite loops. After cancelling the thread, the main process is to wait for thread to be cancelled before exiting the program. Note: By default, cancellation only takes effect at cancellation points (i.e. *pthread_testcancel()).* If the thread never hits one, it won't exit.

Error Handling

Perform the necessary error checking to ensure the correct number of command-line parameters. If the consumer detects a mismatched checksum it is to report the error along with the expected checksum and the calculated checksum and exit the program.

Grading

The program will be graded on the basic functionality, error handling and how well the implementation description was followed. Be sure to name your program **prodcon.c** (no extra characters, capitals) Note that documentation and style are worth 10% of the assignment's grade!

Submission

The source code for program should be available on Canvas along with a README/Output file that provides any documentation and sample output.