CS6233
Final – Part2
Programming Problem
60 minutes
10 points

Please submit your C-code for the following programming problem via the assignment page at NYU classes.

This problem is **open book, notes, class page, man pages, etc.**

You must **work alone** and may not consult or seek help from others; locally, remotely, via internet forums, etc.

You may perform your work on a Linux virtual machine or a Linux laptop.

PROGRAMMING PROBLEM

A **barrier** is a tool for synchronizing the activity of several threads running concurrently. When a thread reaches a **barrier point**, it cannot proceed until all other threads have reached that point as well. Only when the last thread reaches the barrier point, all threads are released and can resume concurrent execution.

Assume that the number of threads executing concurrently is ***n***, thus the **barrier needs to ensure all *n* threads have reached the barrier before allowing any of them to proceed**.

A skeleton program is provided to you (below). You need to implement the barrier function `barrier_point()` and its initialization function `barrier_init()`. (**You are NOT to use the `pthread_barrier_t` provided in pthreads, you are supposed to implement this yourself)**

The program's `main()` routine calls `barrier_init()` and then creates ***n*** threads. The threads are implemented using a function: `thread_func()`.

The `thread_func()` (running ***n*** times) prints the thread number and then sleeps for a random amount of time between 1 and 5 seconds. When it wakes up, it waits for the barrier point (by calling `barrier_point()`) and then prints its thread number again before exiting.

Hint:
You may use a counter that is protected from critical race conditions via a mutex (or a binary semaphore), and a semaphore for waiting and for signaling/releasing threads.

```
/* CS6233 Final Exam - programming problem - barrier implementation */

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

// function prototypes
void *thread_func(void *param);
int barrier_point();
int barrier_init();


// number of threads - obtained from an input argument
int n;

// Variables uses by the barrier
int count;
pthread_mutex_t count_lock;
sem_t semaphore;



int main(int argc, char *argv[])
{
      if (argc != 2) {
            printf("Error - Usage: ./final <number of threads>\n");
            return -1;
      }

      int i;

      n = atoi(argv[1]);

      pthread_t thread_ids[n];
      int thread_num[n];

    // Initialize the barrier
      if ( barrier_init() != 0){
        printf("Error: barrier_init() failed\n");
            return -1;
    }

    // create the threads
      for (i = 0; i < n; i++) {
         thread_num[i] = i;
            pthread_create(&thread_ids[i], 0, thread_func, &thread_num[i]);
      }

    // wait for the threads to finish
      for (i = 0; i < n; i++)
            pthread_join(thread_ids[i], 0);

      return 0;
}
```

```
// The thread function
void *thread_func(void *param)
{
      int seconds;
      int thread_number = *((int*) param);

      /* Sleep for a random period of time */
      seconds = (int) ( (rand() % 5) + 1);
      printf("Thread %d going to sleep for %d seconds\n", thread_number,
seconds);
      sleep(seconds);

      /* Wait at the barrier point */
      printf("Thread %d is into the barrier\n", thread_number);
      barrier_point();

      /* Now we're out of the barrier point */
      printf("Thread %d is out of the barrier\n", thread_number);

      return NULL;
}


int barrier_init()
{

   // Initialize your mutex that protects the counter

   // Initialize your semaphore (used for signaling)

   // Initialize your counter here

      return 0;
}

/* The barrier point function */
int barrier_point()
{

   // put your code here


      return 0;
}
```