EECS 338 Homework 6

General requirements:

- Due on the posted due date.
- Upload a single, compressed file (e.g. zip) to Blackboard that contains all required files.
- Include either a single makefile that compile all programs or a separate makefile for each.
- **Include a typed document with the program output for all problems.**
- All work should be your own, as explained in the Academic Integrity policy from the syllabus. File sharing is prohibited.

1. Implement a semaphore-based solution for the bounded-buffer problem presented in Section 5.7.1 of the textbook. Use multithreading with one thread that is a producer and another thread that is a consumer. Create a shared, circular buffer (array). Design an approach to write/read *N* values to/from the buffer, where *N* is larger than the buffer size. The producer should concurrently insert all *N* values individually (one element per loop iteration) into the buffer and then terminate. The consumer should concurrently retrieve each buffer value individually, printing out each value retrieved. Note that the producer and consumer both know the value of *N*. **Include a typed document with the program output.** Below is sample output where the producer inserts values 10, 20, ...100. It is recommended that you test your solution by temporarily removing the semaphores to ensure that you are not coincidentally getting the correct result.

```
10
20
30
40
50
60
70
80
90
100
```

   Tip (where are the buffer indexes?): The bounded buffer example in section 5.7.1 does not show the in/out buffer indexes that are given in Section 5.1. However, you will need those!

2. Create a multithreaded program for the readers/writers solution from Section 5.7.2. The writer will periodically change the buffer, and the readers will continuously check for changes and print them out. Threads will share a string buffer, an integer that indicates the buffer "version" (described below), and the synchronization variables described in Section 5.7.2. The one parent and two child threads should work as described below. **Include a typed document with the program output.**

   a. The parent is the writer and should write two or more strings to the buffer, sleeping between each buffer update. Each time a string is written to the buffer, an integer "version" variable should be updated to indicate to readers that the buffer has changed. For example, you could initialize the version to 0, increment it to 1 after the 1st string, increment it to 2, after the 2nd string, etc.

   b. Create at least two child threads that are readers. Each reader should run continuously until the last buffer update has been processed, terminating on some predetermined value of the "version" variable. For example, using 4 strings would mean that the reader can terminate after the version variable has been changed to 4. Note that the reader does not just iterate only once per buffer update. When a reader detects that the buffer has changed (by checking the "version" variable), it should print its own thread ID using "pthread_self()" followed by the current string in the buffer. The sample program "pthread_self.c" demonstrates how to use "pthread_self()" to get the thread ID.

Below is an example of the program output in which the writer creates four strings: “Roses are red”, “Violets are blue”, “Synchronization is”, and “Quite fun to do”. Note that the order in which the child threads print, relative to each other, is not controlled. The thread IDs are 66960 and 67216.

```
66960: Roses are red
67216: Roses are red
67216: Violets are blue
66960: Violets are blue
66960: Synchronization is
67216: Synchronization is
66960: Quite fun to do
67216: Quite fun to do
```

Tip (does the writer need a loop?): The instructions for the writer in Problem #2 do not require a loop to print the different strings. You may also attempt to use a loop if you wish to research how to use a 2-dimensional string array.

Tip (the “version” variable): The purpose of the “version” variable is to allow the readers to run freely and primarily use semaphores to prevent conflicts with the critical section. The “version” variable serves as a form of additional synchronization, but it doesn’t prevent readers from reading. Readers may be able to read thousands of times between buffer updates, and the “version” variable is meant to ensure that the reader prints the buffer only when there is something new.

Tip (why do readers do the same thing?): You may wonder why there are two readers printing out the same data. The purpose of the assignment is to show that both readers can work concurrently. Printing the data happens to be nearly the simplest thing we can do. Also, the buffer is extremely simple in that it only holds one string. You might imagine a full document and multiple readers that each look at different parts of the document.

Rubric:

Item	**Points**
Makefile(s)	4
1: Basic programming requirements (declarations, etc.)	3
1: Basic multithreading (creating threads, etc.)	5
1: One thread puts data into buffer correctly	10
1: Other thread removes data from buffer correctly	10
1: Use of semaphores for synchonization	10
2: Basic programming requirements (declarations, etc.)	3
2: Basic multithreading (creating threads, etc.)	5
2: Writer - semaphores/synchronization	10
2: Writer - updating buffer	10
2: Reader - semaphores/synchronization	10
2: Reader - reading and printing buffer	10
Report with output	10
Total	100