EECS 338 Homework 7

General requirements:

- See posted due date.
- Create a typed report for discussion questions and program output.
- Upload a single, compressed file (e.g. zip) to Canvas that contains all required files (programs + report).
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- Include either a single makefile that compiles all programs or a separate makefile for each.

1. Create a combination of server/reader/writer programs for a readers/writers solution from Section 5.7.2. The server creates and configures the shared memory, while multiple readers and writers work with a shared string message. Complete the tasks described below. **Include a typed document with the program output.**

   a. Create a server program that creates the shared memory for the semaphores, synchronization variables, and a string message. It allows the user to decide when to terminate the server, after which it unlinks all shared memory. The purpose of the server is have a single process that "owns" all of the shared memory. *The server does not communicate with the clients*. Below is an example of how it might appear in a terminal, where the user presses the ENTER key following a system call for keyboard input (shown as <*ENTER*>).

   ```
   Press ENTER to quit and unlink shared memory. <ENTER>
   Unlinking shared memory.
   ```

   b. Create a writer program that allows the user to repeatedly choose to either quit or write to the shared string message. When the user chooses to write:
      - Print the value of "read_count" if and only if any readers are reading. Remember to use "mutex".
      - Printing the old message.
      - Ask for the new message and write it to the shared string.
      - Follow the synchronization algorithm from Section 5.7.2.

   c. Create a reader program that allows the user to repeatedly choose to either quit or read the shared string message for a fixed period of time. When the user chooses to read:
      - Print the value of "read_count" appropriately.
      - Printing the current message.
      - Sleep for a fixed period of time (e.g. 2 seconds).
      - Follow the synchronization algorithm from Section 5.7.2.

   Below is an example of program output with two readers and one writer (server output not shown). Gaps are shown in the output only to illustrate the timing of user input (in **bold**).

Reader instance #1:	*Reader instance #2:*	*Writer instance:*
`0. Quit` `1. Read message` `Enter your choice: 1` `Readers: 1` `Message: Original msg.` `This message will be` `valid for 2 seconds.`		*<Writer starts>* `0. Quit` `1. Write message` `Enter your choice: 1` `Waiting on 1 reader(s)...`

<Reader finishes> 0. Quit 1. Read message Enter your choice: *<No user input yet>* *<User enters choice>* Enter your choice: **1** Readers: 2 Message: Hello This message will be valid for 2 seconds. 0. Quit 1. Read message Enter your choice: **0**	*<Reader starts>* 0. Quit 1. Read message Enter your choice: **1** Readers: 1 Message: Hello This message will be valid for 2 seconds. 0. Quit 1. Read message Enter your choice: **0**	*<Reader finishes>* Old message: Original msg. Enter new message: Hello 0. Quit 1. Write message Enter your choice: **0**

2. Repeat problem #2 using the "Second readers-writers" solution, where writers do not starve. **Include a typed document with the program output.** Use the algorithm shown here:

   https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem#Second_readers-writers_problem

   In the Wikipedia description, there are 4 semaphores, and the pseudocode uses P() for wait and V() for post/signal. (See Section 5.6 of the textbook for a discussion of the "P" and "V" terms.)

   Tip (so much shared memory!): Optional sample code is provided for a function that can be used to efficiently create shared memory. The file "memory_example.c" has a function named "create_shm" that can make the steps more encapsulated. Students are not required to use this approach, however. Use the following to compile it:

   gcc memory_example.c -o memory_example -lpthread –lrt

3. Revise the solution for the bounder-buffer problem from HW #6 using *two consumers*, two condition variables (described below), and the monitor algorithm described in Sect. 5.8.3 of the textbook. Each consumer should print a unique ID number to identify which consumer printed the item. However, you do not need to print the value of the mutex semaphore. You may start with the posted solution or use your original solution if it was correct. **Include a typed document with the program output.** Use the following algorithm for synchronization:

   a. Use two condition variables: one for the condition "not full" and another for the condition "not empty".
   b. Use a "count" variable to indicate how many new values are available in the buffer (see section 5.1 of the textbook).
   c. The producer checks the "count" variable and waits on the "not full" condition if the buffer is full. It signals the "not empty" condition after it produces an item.
   d. Each of the two consumers checks the "count" variable and waits on the "not empty" condition if the buffer is empty. Each signals the "not full" condition variable after it consumes (prints) an item.

   Adhere to the following monitor requirements:
   - For concurrency, the loops should not be part of any monitor operations.

- Monitor operations should include only necessary critical section operations.
- Monitor operations should use the same entry/exit logic described in Sect. 5.8.3 of the textbook and demonstrated in the monitor example code provided in class.

Below is a summary of the producer/consumer synchronization algorithm and an example of possible output. In the example output, the producer transfers 10 values from an array, and each consumer gets 5 values. The number of iterations (10 and 5) is defined in advance, but the output order could be different each time the program is executed.

One Producer	*Each Consumer*	*Example output*
• If buffer is full, wait on "not full". • Produce an item. • Signal "not empty".	• If buffer is empty, wait on "not empty". • Consume an item (print ID and item). • Signal "not full".	`Child #2 consumed data = 10` `Child #1 consumed data = 20` `Child #1 consumed data = 30` `Child #2 consumed data = 40` `Child #2 consumed data = 50` `Child #1 consumed data = 60` `Child #2 consumed data = 70` `Child #1 consumed data = 80` `Child #1 consumed data = 90` `Child #2 consumed data = 100`

4. Repeat the bounder-buffer problem from above using Java. Use the same monitor requirements with a ReentrantLock and two Condition variables for the ReentrantLock. The sample programs Monitor1.java and Monitor2.java demonstrate the necessary techniques.