In this assignment, you will practice using mutexes and condition variables by implementing two versions of the readers and writers problem. The first version must favor readers and the second version must favor writers. Section 2.5.2 of your textbook describes this classical interprocess communication problem. You should also watch the video *The Readers and Writers Problem* before beginning this assignment. For this assignment, you may work individually or in a group with one other student.

Setup

In CodeLite on your virtual machine:

- Create two C++ projects named Project4_RW_V1 and Project4_RW_V2.
- For this project, you do not need to create a class. You may write all of your code in the main.cpp file.

Submitting Your Project

Submit the source code (.cpp files) from each of your projects in separate zip files. Name the zip files YourLastNameYourFirstNameProject4_V1.zip and YourLastNameYourFirstNameProject_V2.zip. Upload these zip file to Canvas before the due date.

Important submission notes for students working in a group:

If you worked with another student, name the zip files using the last names of both students. For example, name the version 1 submission:

FirstPersonsLastNameSecondPersonsLastNameProject4_V1.zip

Use a similar style for the version 2 submission.

One person should upload the zip files to Canvas. The other person should upload a single document stating who their team member was.

Version 1

Write a multi-threaded C++ program that gives readers priority over writers in accessing a shared variable. If any readers are waiting, they have priority over writer threads. Writers can only write when there are no readers waiting. It is acceptable to have multiple processes reading the shared variable at the same time. However, if one process is updating (writing) the shared variable, no other process may have access to the shared variable. The program outlined in Figure 2-48 of your textbook implements this approach.

Implementation Notes for Version 1

- You must use the Pthreads (POSIX threads) library for threads and mutexes. You do not need to implement your own semaphores.
- Your shared data must be an integer variable.
- Multiple readers and writers must be supported, and you should easily be able to vary the number of readers and writers. (Use constants for the number of readers and the number of writers.)
- The start routines for both readers and writers must have one int parameter, the thread number.
- Each reader thread must read the shared variable *n* times before exiting. *n* must be represented as a constant in your program (for example, const int NUM_READS = 5) **Hint:** Replace the *while(TRUE)* loops in the solution given in the book with a *for* loops that iterates NUM_READS times.
- Writers must write a random integer to a shared integer variable.
    - The C++ rand class can be used to generate random numbers. https://en.cppreference.com/w/cpp/numeric/random/rand
    - Remember to seed the random number generator before you use it. You only need to do this once. I would put the code to seed the random number generator in the main method.
- Writers must write the shared variable *m* times before exiting. **Hint:** Replace the *while(TRUE)* loop in the solution given in the book with a *for* loop that iterates *m* times.
- Readers must print:
    - the thread type (reader), thread number and value read
    - the number of readers present in the critical region (reading the database) when the value is read
- Sample output for readers

```
reader 2 read 17 3 reader(s)
```

- Writers must print:
    - the thread type (writer), thread number and value written
    - The number of readers present in the critical region (reading the database) when the value is written (should be 0).
- Sample output for writers

```
writer 2 wrote 147 0 reader(s)
```

- Be sure your output is aligned in a way that is easy to read. This will make testing and debugging your program easier.
- Before a reader or writer attempts to access the shared variable, it should wait some random amount of time. This will help ensure that reads and writes do not occur all at once.
    - Use the function usleep() to put a thread to sleep. It takes one argument, a time in microseconds. I had my threads sleep randomly between 10 and 100 microseconds each time before accessing the shared variable. You will need to experiment to find random times that work for you.

Sample Output for Version 1

Remember that the order in which your threads execute will be different from mine. The order will also be different each time you run the program!

Constant values used for this output:
NUM_READERS = 3
NUM_WRITERS = 2
NUM_READS = 3 (number of reads for each reader)
NUM_WRITES = 3 (number of writes for each writer)

```
writer 2 wrote 136 0 reader(s)
writer 1 wrote 120 0 reader(s)
reader 3 read  120 1 reader(s)
reader 2 read  120 1 reader(s)
writer 2 wrote 127 0 reader(s)
reader 1 read  127 1 reader(s)
writer 1 wrote  67 0 reader(s)
writer 2 wrote 172 0 reader(s)
reader 3 read  172 1 reader(s)
reader 2 read  172 1 reader(s)
reader 3 read  172 2 reader(s)
reader 1 read  172 1 reader(s)
writer 1 wrote  82 0 reader(s)
reader 1 read   82 1 reader(s)
reader 2 read   82 1 reader(s)
Press ENTER to continue...
```

Version 2

In most cases, we would like updates to take place as soon as possible. (Think of the airline reservation system example discussed in the video.) This version will favor writers. Your implementation must allow readers to enter the critical section until a writer arrives. When a reader arrives and a writer is waiting, the reader should be suspended behind the writer instead of being allowed to enter the critical region immediately. A writer will need to wait until all readers that are active in the critical region are finished but must not wait for readers that arrived after its arrival. If more than one writer is waiting, all writers must be allowed to write before any readers are allowed to read again.

Implementation Notes for Version 2

- All of the implementation notes for version 1 apply to version 2.
- You must use condition variables to put to sleep and wake up readers and writers. Your solution must not use busy waiting. You will not receive credit for version 2 if you use busy waiting. See the producer-consumer code on page 138 for an example of using condition variables to synchronize process events. This code is explained in the video *PThreads: Mutex and Condition Variables*.
- In addition to printing the thread type, thread number, value read, and the number of readers present in the critical region when the value is read, readers must print:
    - The number of readers waiting
    - The number of writers waiting
- Sample output for readers:

```
reader 1 read 71  1 reader(s), 1 reader(s) waiting,  1 writer(s) waiting
```

- In addition to printing the thread type, thread number, value written, and the number of readers present in the critical region when the value is read, writers must print:
    - The number of readers waiting
    - The number of writers waiting
- Sample output for writers:

```
writer 2 wrote 192  0 reader(s), 1 reader(s) waiting,  0 writer(s) waiting
```

Sample Output for Version 2

Remember that the order in which your threads execute will be different from mine. The order will also be different each time you run the program!

Constant values used for this output:
NUM_READERS = 5
NUM_WRITERS = 2
NUM_READS = 3 (number of reads for each reader)
NUM_WRITES = 3 (number of writes for each writer)

```
reader 2 read    0 3 reader(s), 1 reader(s) waiting,  1 writer(s) waiting
reader 5 read    0 2 reader(s), 2 reader(s) waiting,  2 writer(s) waiting
reader 4 read    0 1 reader(s), 2 reader(s) waiting,  2 writer(s) waiting
writer 2 wrote 179 0 reader(s), 2 reader(s) waiting,  1 writer(s) waiting
writer 1 wrote  69 0 reader(s), 3 reader(s) waiting,  0 writer(s) waiting
reader 1 read   69 2 reader(s), 2 reader(s) waiting,  0 writer(s) waiting
reader 2 read   69 2 reader(s), 1 reader(s) waiting,  0 writer(s) waiting
reader 5 read   69 1 reader(s), 1 reader(s) waiting,  0 writer(s) waiting
reader 3 read   69 1 reader(s), 0 reader(s) waiting,  0 writer(s) waiting
reader 1 read   69 2 reader(s), 2 reader(s) waiting,  2 writer(s) waiting
reader 2 read   69 1 reader(s), 2 reader(s) waiting,  2 writer(s) waiting
writer 1 wrote  27 0 reader(s), 2 reader(s) waiting,  1 writer(s) waiting
writer 2 wrote 170 0 reader(s), 3 reader(s) waiting,  0 writer(s) waiting
reader 3 read  170 1 reader(s), 3 reader(s) waiting,  0 writer(s) waiting
reader 4 read  170 3 reader(s), 0 reader(s) waiting,  1 writer(s) waiting
reader 5 read  170 2 reader(s), 0 reader(s) waiting,  1 writer(s) waiting
reader 1 read  170 1 reader(s), 2 reader(s) waiting,  2 writer(s) waiting
writer 1 wrote 157 0 reader(s), 2 reader(s) waiting,  1 writer(s) waiting
writer 2 wrote 192 0 reader(s), 2 reader(s) waiting,  0 writer(s) waiting
reader 4 read  192 1 reader(s), 1 reader(s) waiting,  0 writer(s) waiting
reader 3 read  192 1 reader(s), 0 reader(s) waiting,  0 writer(s) waiting
Press ENTER to continue...
```

Grading Criteria (20 points possible)
Note: Your program must compile in CodeLite to receive credit for this assignment!

Points	Criteria
0 - 3 points	**Main Method (both versions):** Are constants used to represent the number of reader threads, the number of writer threads, and the number of reads or writes? Are mutex and condition variables properly created and freed? Does the main method correctly create and join the correct number of reader threads and writer threads?
0 - 4 point	**Readers - Version 1:** Does the reader start routine have one parameter, the thread number? Does each reader thread read an integer from the shared integer variable the appropriate number of times? Is the usleep function used to randomly cause a thread to wait before it attempts to access the critical region? Is the readers function implemented as shown in the textbook? Is output printed for each read as described above?
0 - 4 points	**Writers- Version 1:** Does the writer start routine have one parameter, the thread number? Does each writer thread write a random integer to the shared integer variable the appropriate number of times? Is the usleep function used to randomly cause a thread to wait before it attempts to access the critical region? Is the writers function implemented as shown in the textbook? Is output printed for each write as described above?
0 - 4 point	**Readers - Version 2:** Is Version 1 modified so that writers now have preference? Are readers prevented from entering the critical region after a writer has arrived? Are condition variables used to put readers to sleep and to and wake up a writer? Your solution must use condition variables. It must **not** use busy waiting! You will not receive credit for version 2 if your solution uses busy waiting.
0 - 4 points	**Writers - Version 2:** Is Version 1 modified so that writers now have preference? Are writers allowed to enter the critical region before any readers that arrive after the writer. If another writer is waiting when a writer finishes, is it allowed to write before any readers are allowed to enter the critical region? Are condition variables used to put a writer to sleep and to and wake up either a writer if writers are waiting or all readers if a writer is not waiting? Your solution must use condition variables. It must **not** use busy waiting! You will not receive credit for version 2 if your solution uses busy waiting.
0-1 point	**Style:** Is the code easy to read? Does it follow class style guidelines (See Program Style page in Canvas.)