1. First, define these four functions as the trivial functions (do nothing).

(This has already been done for you in the file acquire_release_case1.c)
Then do: make case1

a. Observe that you hit an assert statement as readers and writers conflict.
Explain why.

There are multiple readers and writers, trying to update the same variable and in the process update the num_readers and num_writers flag. Since the locking scheme is not implemented (Blank Code), the reader process that updates the flag does not undo it and therefore, when the assert test is performed *num_reader* still has value =1 even after completing the read process, hence the assert test before the write fails.

Output

```
read_write_locks: read_write_locks.c:130: begin_write: Assertion `num_writers == 1
&& num_readers == 0' failed.
Makefile:37: recipe for target 'check' failed
make[1]: *** [check] Aborted (core dumped)
make[1]: Leaving directory '/home/manjit/Downloads/HW5-locks'
Makefile:23: recipe for target 'case1' failed
make: *** [case1] Error 2
```

b. Now let's look at the running time, if we ignore the assert statement. Do:
make clean
make NOASSERT=1 case1
make clean
Observe that now the code completes with a time of about 100 milliseconds.
Explain why.

(HINT: If you look closely at read_write_locks.c, you'll see that there
are 2 readers and 2 writers. Each reader and each writer does 10 tasks,
and for each task, it sleeps 10 milliseconds.)

when we run with NOASSERT=1, this an option that disables the assert option in the program therefore the program does not check for the assert and runs successfully. In this case there is no guarantee of consistency of the update of the data.

Output

```
*** 20 reads and 20 writes in 111 milliseconds
```

2. Next, let's fix the bug that the assert statement showed to us.

Define these four functions so that each acquire_*_lock() calls:
pthread_mutex_lock(&acquire_release_lock);
and so that each release_*_lock() function calls:
pthread_mutex_unlock(&acquire_release_lock);
(You will also need to define 'acquire_release_lock', similarly to the way
that 'task_stats_lock' is defined in read_write_locks.c, and also
include the 'pthread.h' file.)
Write this in acquire_release_case2.c.
Then do: make case2

This caused each reader and writer to wait to acquire the
"acquire_release_lock".
Observe the number of reads and writes, as well as the total time taken.
There is a clear relationship between the number of reads/writes and
the total time taken. Describe the relationship, and explain why.

Here, each process is waiting to acquire the lock.

There are 2 threads and 10 tasks for each thread, therefore 20 read threads and 20 write threads
each reach thread waits for 10ms for the lock, that's 100ms for 10 tasks for 1 thread
2 thread in total may take 2x100ms = 200ms

So in total, 200ms for read and 200ms for write, approximately around 400ms

As we increase the threads the total time increases by N*100 ms , where N is the number of threads. The linear relationship exists because each thread has to wait before acquiring and there is no other way around.

Output

```
*** 20 reads and 20 writes in 430 milliseconds
```

3. Next, let's see if we can produce a more efficient version of this solution.

In acquire_release_case3.c, define global variables:

```
int num_with_shared_lock = 0;
int num_with_excl_lock = 0;
```

For acquire_shared_lock(), do:

```
while (1) {
  pthread_mutex_lock(&acquire_release_lock);
  if (num_with_excl_lock > 0) {
    pthread_mutex_unlock(&acquire_release_lock);
    struct timespec ten_milliseconds = {0, 10000000};
    nanosleep(&ten_milliseconds, NULL);
    continue;
  }
  num_with_shared_lock++;
  if (num_with_excl_lock == 0 && num_with_shared_lock == 1) {
    pthread_mutex_lock(&read_lock);
  }
  pthread_mutex_unlock(&acquire_release_lock);
  break;  // We now have the shared lock.
}
```

For release_shared_lock(), do:

```
pthread_mutex_lock(&acquire_release_lock);
num_with_shared_lock--;
if (num_with_shared_lock == 0) {
  pthread_mutex_unlock(&read_lock);
}
pthread_mutex_unlock(&acquire_release_lock);
```

Then define *_exclusive_lock() appropriately to test that
"num_with_shared_lock == 0" before acquiring the exclusive lock.
It should sleep and try again later if it finds that num_reader > 0.
Define *_exclusive_lock() compatibly with the above design.
In particular, acquire_exclusive_lock() should follow the pattern, below.

```
while (1) {
  pthread_mutex_lock(&acquire_release_lock);
    if (num_with_excl_lock == 0 && num_with_shared_lock == 0) {
      num_with_excl_lock++;
      pthread_mutex_unlock(&acquire_release_lock);
      break; // We now have the exclusive lock
    } else {
      struct timespec ten_milliseconds = {0, 10000000};
      nanosleep(&ten_milliseconds, NULL);
    }
```

```
    pthread_mutex_unlock(&acquire_release_lock);
  }
```

a. With the above design, do: make case3
You will likely see the code continue in an infinite loop.
Explain what you see, and why.

Depending on the machine and the OS, the code may or may not encounter an Infinite loop. In this case, the code runs much longer and the run time is sporadic. Here, as and when a thread starts, the next thread will check for lock and when it does not get it , it will sleep for 10 milliseconds, this is true for both shared lock and the exclusive lock, and then try again in a while loop until it gets a lock.

Output

```
*** 20 reads and 20 writes in 3213 milliseconds
```

b. Now, modify acquire_exclusive_lock() by moving the final
'pthread_mutex_lock()' to before the 'nanosleep()' instead
of after it.
Then do: make case3
You will likely see the bug go away.
Explain what happens, and why

By moving the unlock before the sleep, we are avoiding the time wastage due to the wait. When we do not get the exclusive lock,we try again quickly than wait for 10 ms. Therefore, consistently we see the threads running faster.

Output

```
*** 20 reads and 20 writes in 331 milliseconds
```

4. We saw that there was a bug in Case 3, when NUM_READERS is set to 2.

In acquire_release_case4.c, define global variables (and needed 'include's):

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>

pthread_mutex_t acquire_release_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t write_lock = PTHREAD_MUTEX_INITIALIZER;

int num_with_shared_lock = 0;
int num_with_excl_lock = 0;
```

For acquire_exclusive_lock(), define the function as follows:

```
int has_lock = 0;
while (! has_lock) {
  pthread_mutex_lock(&acquire_release_lock);
  if (num_with_shared_lock == 0 && num_with_excl_lock == 0) {
    num_with_excl_lock++;
    pthread_mutex_lock(&write_lock);
    has_lock = 1;
  }
  pthread_mutex_unlock(&acquire_release_lock);
  if (! has_lock) {  // delay before trying again for the lock
    struct timespec ten_milliseconds = {0, 10000000};
    nanosleep(&ten_milliseconds, NULL);
  }
}
```

Write the appropriate definitions for:

* void release_exclusive_lock();
* void acquire_shared_lock();
* void release_shared_lock();

There must be no calls to sleep or nanosleep in the remaining definitions.
HINT: For the reader function, think about testing if the write_lock is held.

In the following, run your program on a computer with at least 5 CPU cores and preferably 10 or more. The machine login.ccs.neu.edu has 48 CPU cores, but use the command 'top' to check if too many users are on it at one time.

a. Then do: make case4
Explain what happens, and why.

Previous scenario we had sleep both for the shared lock and the exclusive lock. In this case we have eliminated one of the sleep in reader as the read is generally more common than the write. Hence, overall for many cases this approach will perform better. As we can see the overall run time is faster than the case 3.

Output

```
*** 20 reads and 20 writes in 331 milliseconds
```

b. Next, modify read_write_locks.c to set NUM_READERS to 5.
Do: make case4
Explain how the printed statistics change and why.

The output is similar to previous case, therefore the hypothesis that the read occurs more frequently than the write and not having the sleep in the read part if validated. There is not waiting time to acquire the shared lock and more than one process can acquire the shared lock as per the rules. Therefore having more threads does not increase the run time.

Output

```
*** 50 reads and 20 writes in 331 milliseconds
```

c. Next, also modify read_write_locks.c to set NUM_WRITERS to 5.
Do: make case4
Explain how the printed statistics change and why.

The process takes longer now, since there are more writers and the multiple writer threads have to wait for the lock, since write operation is exclusive and that there can be only one lock. Here the increase in the run time is purely attributed to the increase in the writers. The number of readers do not have effect on the run time.

Output

```
*** 50 reads and 50 writes in 670 milliseconds
```

d. Develop a formula in terms of NUM_READERS and NUM_WRITERS
that will predict the time to complete the program read_write_locks,
when this program is executed on a computer with 10 CPU cores available
to the program. (This formula will only be approximate, but see how
well you can predict the times for a large number of readers, writers,
or both.)

The formula mainly depends on the timing of the writer. Each additional writer takes approximately 120 ms more than the previous operation.

Single writer takes 229 ms and Two writer takes approximately 350 ms.
Therefore 100 + 120*N , is the formula for the program time, where N is the number of writers

```
*** 50 reads and 10 writes in 229 millisecond
*** 50 reads and 20 writes in 358 milliseconds
*** 50 reads and 30 writes in 444 milliseconds
*** 50 reads and 40 writes in 549 milliseconds
*** 50 reads and 50 writes in 668 milliseconds
```