Due: Mon, Mar 13, 2017 @ 4pm

1. Using the code for vector-deadlock.c, solve question 1 from chapter 32 of the textbook.

   Analysis of vector-deadlock:
   Vector deadlock as it is described in this question will initialize two vectors per the two threads (-n 2) that are created. The vectors are then added in two separate orders.

   The first thread that is created (i=0 in the 'for (i = 0; i < num_threads; i++' segment of main-common) will add index-corresponding (index corresponding in the sense that the 1st item of vector 1 will be added to the 1st item of vector 0, and the result will be stored in the 1st item of vector 0) values of vector 1 to vector 0 using the vector_add() function defined in vector-deadlock.c. Vector 0 corresponds to the 0th and vector 1 corresponds to the 1st index positions of v[] in this instance.

   The second thread (i=1 in the same segment) will add index-corresponding values of vector 1 to vector 0, corresponding to the 1st and 0th index positions of v[] respectively. This is due to the vector_add_order flag, since cause_deadlock is 0 in this case, the vector_add_order flag is always 0, which ensures that worker() will always add vector 1 to vector 0. This is also because enable_parallelism == 0, causing both threads to operate on the same vector.
   The output changes between the following two cases:

```
Tonys-MacBook-Pro:code rpg711$ ./vector-deadlock -n 2 -l 1 -v
->add(0, 1)
                ->add(0, 1)
<-add(0, 1)
                <-add(0, 1)
Tonys-MacBook-Pro:code rpg711$ ./vector-deadlock -n 2 -l 1 -v
                ->add(0, 1)
->add(0, 1)
                <-add(0, 1)
<-add(0, 1)
```

   This shows that thread 0 sometimes reaches the add method before thread 1, and thread 1 reaches the add method before thread 0. Since there is a mutex in print_info, only one thread may print the entry '->add()' text or returning '<-add()' text at the same time. It is not possible for one thread to finish before another, because they operate on the same vectors with pointers to the same mutex objects. One thread will always enter add() before the other, and upon returning from add will immediately lock the print mutex and print

When the '-d' flag is enabled, the two threads that are created are assigned different add orders. The even thread (0^{th} in this case) will attempt to add vector 0 to vector 1, just like before. The odd thread (1^{st} in this case) will attempt to add vector 1 to vector 0, which is opposite from before. For lower looping numbers, this does not seem to cause a deadlock very frequently. With '-l 1000000', I was able to produce a deadlock after a few tries. This is caused by the add_vector() function in vector-deadlock.c. Since the vector add orders are inverted between the two threads, the deadlock situation occurs when one thread will lock the vector 1 mutex, and attempt to acquire the vector 0 mutex. At the same time, the other thread locks the vector 0 mutex and attempts to acquire the vector 1 mutex. Since both mutexes are locked at this point, the threads wait forever, neither will enter the critical section and release either of the locks, as they are waiting on each other to unlock their respective locks that they are waiting on.

To reiterate, the code does not seem to always deadlock, likely due to the speed at which vector_add can perform its task allowing the threads to essentially perform actions in tandem without ever encountering a case where they both enter the vector_add function at about the same timing. With a big enough loop number, it is possible to produce a deadlock 100% of the time, as the code is intentionally flawed in that way.

I am assuming this question is a continuation of question 2 from the previous problem, with ‘-d’ flag set. Changing the amount of threads increases the amount of worker threads that operate on adding vector 0 to vector 1 (even case) and vector 1 to vector 0 (odd case). By doing this, it increases the probability that two threads will enter the deadlock situation that I answered in problem 2 above. In this way, the only thread argument that ensures no deadlock can possibly happen is ‘-n 1’ or ‘-n 0’. ‘-n 0’ is nonsensical, it creates 0 threads and has no side effects, only printing some runtime stats (if enabled) and subsequently returning, so ‘-n 1’ is the only argument that causes the program to be guaranteed to do useful work and terminate with the ‘-d’ flag enabled.

documentation at docs.oracle.com to answer this question.

Code snippet provided for clarity of answer:

```
Lock l = readwritelock.readLock();
l.lock();
try {
   // critical section
} finally {
   l.unlock():
}
```

Acquiring the lock object in read mode is done using readwritelock.readLock(). This merely returns the reference to the readlock managed by the parent readwritelock. The lock must then be locked and unlocked at the critical section of code where it is needed, with a finally block to ensure unlocking occurs despite any exceptions that may occur.

To acquire the read lock, the lock() method of the Lock interface is used.
To release the read lock, the unlock() method of the Lock interface is used.

I referred to both my native pthread.h on my mac at /usr/include/pthread.h as well as http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html for this question, in case of inconsistencies in either document. The methods appear to correspond one to one, so this should be an exhaustive list.

The read write lock methods for pthreads are the following:

```
int   pthread_rwlock_destroy(pthread_rwlock_t *);
int   pthread_rwlock_init(pthread_rwlock_t *,
        const pthread_rwlockattr_t *);
int   pthread_rwlock_rdlock(pthread_rwlock_t *);
int   pthread_rwlock_tryrdlock(pthread_rwlock_t *);
int   pthread_rwlock_trywrlock(pthread_rwlock_t *);
int   pthread_rwlock_unlock(pthread_rwlock_t *);
int   pthread_rwlock_wrlock(pthread_rwlock_t *);
int   pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
int   pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *,
        int *);
int   pthread_rwlockattr_init(pthread_rwlockattr_t *);
int   pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
```