Challenge 1: "Make a barrier using only one mutex lock() and unlock() call!"

"Impossible! Line 2 is a Critical Section, if a thread has locked the mutex..."

But here is an awful solution. (Why is this a 'poor' solution?)

```
01  void barrier() {

02    count ++

03    while( count != N) ?
04

05  }
```

2. When is disabling interrupts a solution to the Critical Section Problem?

```
pthread_mutex_lock    => { disable interrupts on the CPU }
pthread_mutex_unlock  => {enable interrupts on the CPU }
```

Are there limitations to this approach?

3. Challenge II: Create a barrier using each of the following lines once. All 5 threads must call barrier before they all continue.

```
int remain =5;   earlier... sem_init(&s,0,___?)
void barrier() { ... Rearrange the following!
   sem_wait(&s);
   sem_post(&s);
   remain --;
   pthread_mutex_lock(&m);
   pthread_mutex_unlock(&m);
   if(remain)
}
```

4. Is there a Race condition?

```
pleaseStop = 1          while(!pleaseStop)
p_cond_broadcast(&cv)      p_cond_wait(&cv,&m)
```

5. Deadlock: " _____ "

Use two mutex locks and two threads to create an example of deadlock

Thread1:	Thread 2:

Use three counting semaphores and three threads to deadlock 3 threads

thread #1:	thread #2:	thread #3:

- Must deadlock involve threads? What about single-threaded processes?

6. What is the Resource Allocation Graph for deadlock detection?

7. The Reader Writer problem

A common problem in many different system applications

read_database(table, query) {...}	update_row(table, id, value) {...}

cache_lookup(id) {...}	cache_modify(id, value) {...}

8. ReaderWriter locks are useful primitives & included in the pthread library!

```
01  pthread_rwlock_t lock;        01  cache_lookup(id) {
                                  02      p...rdlock(...)
02  p_rwlock_init                 03      read from resource
03  p_rwlock_wrlock               04      p...unlock(...)
04  p_rwlock_rdlock               05      return result
05  p_rwlock_unlock               06  }
```

CS241: synch. skills and the ability to *build* these! Along the way, also learn to reason about, develop and fix multi-threaded code

9.   ~~ Welcome to the *Reader Writer* Game Show! ~~

Contestant #1

```
p_mutex_t *readlock,*writelock        write() {
readlock=malloc(sizeof p_mutex_t)     lock(writelock)
writelock=malloc(sizeof p_mutex_t)    lock(readlock)
p_m_init(readlock,NULL)               // do writing
P_m_init(writelock,NULL)              unlock(readlock)
                                      unlock(writelock)
read() {                              }
lock(readlock)
// do read
unlock(readlock)
}
```

Is #1 a Solution? Problems?

Contestant #2

```
bool reading=0, writing=0
read() {                      write() {
  while(writing) {}             while(reading||writing) {}

  reading = true               writing = true
  // do reading here           // do writing here
  reading = false              writing = false
}                             }
```

Is #2 a Solution? Problems?

Contestant #3

```
read(){                       write(){
 lock(&m)                      lock(&m)
 while (writing)               while (reading||writing)
    cond_wait(cv,m)               cond_wait(cv,m)

 reading++                     writing++

/* Read here! */              /* Write here! */

 reading--                     writing--;
 cond_signal(cv)               cond_signal(cv)
 unlock(&m)                    unlock(&m)
```

Is #3 a Solution? Problems?