Please read 5.1 to 5.7 of the textbook and then answer the following, trying not to look at your notes or at the textbook. Quiz #6, on Thursday 15th November, will consist exclusively of questions taken from the Part I of this homework.

**Part I — Short Questions**

**Question 1**
Briefly define what a critical section is.

**Question 2**
Briefly define what a race condition is.

**Question 3**
What is starvation? How is it different from deadlock?

**Question 4**
What is memory ordering? What is its purpose?

**Question 5**
What is an atomic instruction? When is it useful?

**Question 6**
Name one difference between `test_and_set` and `compare_and_swap`.

**Question 7**
Discuss a situation where it is actually desirable to have a process “busy waiting” for a resource to be accessible.

**Question 8**
What are the three operations one can perform on a semaphore?

**Question 9**
What is the purpose of the `preempt_disable()` instruction?

**Question 10**
What is priority inheritance? Why is it useful?

**Question 11**
What are the benefits of nonpreemptive kernels?
Part II — Problem

There is only one problem this time: it requires a computer and it is rather lengthy. As usual, I'll assume that you will have successfully completed it by the time Homework #7 is released (Thursday 15th November), but you probably want to look at it before the exam to increase your chances of success.

```c
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
pthread_mutex_t lock;

void* task1(void *arg)
{
    pthread_t ttid = pthread_self();
    printf("\n Thread %08x started\n", ttid );
    pthread_mutex_lock(&lock);
    printf("\n Thread %08x performs critical section\n", ttid );
    sleep(1);
    printf("\n Thread %08x is about to be done with its critical section\n", ttid );
    pthread_mutex_unlock(&lock);
    printf("\n Thread %08x finished\n", ttid);
    pthread_exit(0);
}

int main(void)
{
    pthread_mutex_init(&lock, NULL);
    pthread_create(&tid[0], NULL, &task1, NULL);
    pthread_create(&tid[1], NULL, &task1, NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    exit(0);
}
```

Listing 1: mutex_pb1.c

Problem 1

This long problem is divided into four parts.

(a) Examine the code displayed in Listing 1 carefully, and answer the following:
i. How many threads are created in the main function? What function do they execute?

ii. Does the main function wait for the threads to terminate to exit?

iii. What is the datatype of the lock variable?


vi. We basically covered all the lines of this program: make sure you understand the remaining lines (that is, including libraries, prototypes, variable declarations, printing, exiting, sleeping).

(b) Run your virtual machine, created a “06” folder in your “Desktop/HW/” folder, and copy the code displayed in Listing 1, or download it from http://spots.augusta.edu/caubert/teaching/2017/fall/csci3271/hw/06/mutex_pb1.c. Compile the program, using `gcc -l pthread mutex_pb1.c`, and execute it, using `.a.out`.

(c) We will now modify this program progressively. Make a copy of the file before each step, to make sure that you can always return to a previous, “working” state of the program. Compile and execute before and after every step.

i. Modify the program so that a message is printed on the screen if `pthread_create` returns an error code. The code shared previously, available at http://spots.augusta.edu/caubert/teaching/2017/fall/csci3271/code/2017_10_17_thread.zip, should help you.

ii. In task1, replace

```c
pthread_mutex_lock(&lock);
with
```

```c
if(pthread_mutex_trylock(&lock)){
   printf("\n   Thread %08x is waiting\n", ttid );
   pthread_mutex_lock(&lock);
}
```

Find in the documentation what `pthread_mutex_trylock` does, and analyze this code before compiling and running it.

iii. Create a task2 function, whose code is exactly the same as the code for task1, except for the name. Modify your program so that, in the main function, the first thread created execute task1 while the second execute task2.

iv. Modify task1 and task2: instead of storing the value returned by `pthread_self` in the `ttid` variable, store it respectively in `tid[0]` and `tid[1]`. Modify the rest of those functions accordingly.

v. Finally, create a second mutex lock, named `lock2` and create a deadlock: make `task1` lock the first lock and then the second lock, and make `task2` lock the second lock and then the first lock. You may want to add a `sleep(0.3);` instruction between the locking of the two locks to increase your chance of a deadlock occurring. Use `Ctrl + C` to exit your program.

(d) Read 5.11.2 and make sure that what you coded at the previous step is actually an implementation of a deadlock. Consider the code we originally started with, and assume that we had just “forget” to release the lock in task1. Would that be considered a deadlock?