Quiz #5, on Wednesday, October 31st, will consist of questions taken from or inspired Parts I and II of this homework.

Part I — Questions

1. Briefly define what a race condition is.

2. Name a benefit of having a nonpreemptive kernel.

3. Briefly define what a critical section is.

4. What is starvation? How is it different from deadlock?

5. What is memory ordering? What is its purpose?

6. What is an atomic instruction? When is it useful?

7. Name one difference between `test_and_set` and `compare_and_swap`.

8. Discuss a situation where it is actually desirable to have a process "busy waiting" for a resource to be accessible.

9. What are the three operations one can perform on a semaphore?

10. What is the purpose of the `preempt_disable();` instruction?

11. What is priority inheritance? Why is it useful?

Part II – Problems

As usual, I'll assume that you will have successfully completed the following problems by Wednesday, October 31st, so don't wait and let me know if you had difficulties solving them.

Problem 1

Why do we say that a "good protocol" for synchronization should enforce mutual exclusion, progress, and busy waiting? Define each notion, and give an example of a protocol that enforces only two out of the three conditions, an example of protocol that violates all of them, and an example of a protocol that enforces them all. You can re-use the "Single Professor in a classroom full of students with questions at the same time" metaphor to define those notions and the examples of protocol.

Problem 2

This long problem is divided into four parts.

1. Examine the code displayed in Listing 1 carefully, and answer the following:
   (a) How many threads are created in the `main` function? What function do they execute?
   (b) Does the `main` function wait for the threads to terminate to exit?
   (c) What is the datatype of the `lock` variable?

(d) Try to guess what the `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, and `pthread_mutex_destroy` functions do. Compare your answers with what's indicated at `https://manpages.debian.org/jessie/glibc-doc/pthread_mutex_lock.3.en.html`.

(e) Try to guess what the `pthread_self` function do. Compare with `http://man7.org/linux/man-pages/man3/pthread_self.3.html`.

(f) 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).

2. Start your virtual machine, create a "5" folder in your "Desktop/HW/" folder, and download and extract the code located at `http://spots.augusta.edu/caubert/teaching/2018/fall/csci3271/code/hw5.zip`, or copy it from Listing 1. Compile the program, using `gcc -l pthread lock_pb.c`, and execute it, using `./a.out`. Make sure the output is consistent with your understanding of the code.

3. 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.

   (a) 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/2018/fall/csci3271/code/2018_10_06_threads.zip`, should help you.

   (b) In `task1`, replace

   ```
   pthread_mutex_lock(&lock);
   ```

   with

   ```
   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.

   (c) 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`.

   (d) This is the most challenging task: create a second mutex lock, named `lock2` and create a deadlock: make `task1` lock the first lock and then the second lock, perform its critical section, and then release the second lock and the first lock. Make `task2` lock the first lock and then the second, perform its critical section, and release the second lock and then the first lock. You may want to add a `sleep(0.5);` instruction between the locking of the two locks to increase your chance of a deadlock occurring.

   Once you successfully created a deadlock, use `Ctrl + C` to exit your program. The solution to *this particular part of the problem* is in the archive you downloaded, at `lock_sol.c`.

4. Find the formal definition of a deadlock and make sure that what we implemented at the previous step is actually a deadlock. Consider the code we originally started with, and assume that we had just "forgotten" to release the lock in `task1`. Would that be considered a deadlock?

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

```c
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\t  Thread %08x performs critical section.\n", ttid );
        sleep(1);
        printf("\n\t  Thread %08x finished 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: lock_pb.c