Assignment #2
Threads, Race Conditions & Mutual Exclusion

Grade Weight: 100pts

Assignment Overview:

In this assignment, you will edit two programs. One user-level application and a second program that is a loadable kernel module (LKM). The code in the kernel represents a basic device driver. The end goal is to have a user-level application that runs multiple threads in parallel. Each thread accesses the same device driver. If they all access the driver at the same time, they will overwrite each other's data and the code will not function properly. The device driver (Loadable Kernel Module) needs to be edited to be thread-safe, such that only one program can use the driver resource at any one time.

Assignment Description:

The following sample code contains two programs: hw2_sample_code.tar
1. **userspace.c** → as the name suggests, contains code that runs in the user-space
2. **kernelDriver.c** → as the name suggests, contains code that runs in the kernel. This is an initial step to creating a kernel driver which will built in Assignment #3.

Run the Sample Code

Test the current sample code. You must load the kernel driver as a Linux Kernel Module first. The sample code opens a device driver. Sends the drivers some information from user space., then reads that information back from the driver. Then the userspace program closes the driver. The output should look similar to the following. Pink areas indicate commands or user input.

Edit the Code

In this assignment, you will edit the userspace.c code to do the following:
- userspace.c → create 2 threads in parallel. Each thread will access the kernel driver, hold it until the user presses "ENTER" and then will release the driver resource.
- kernelDriver.c → needs to be edited such that only one thread or process can access the driver at any one time.

Part #1 - Edit the user space code

Change the user space code to have multiple threads request the driver resource in parallel.
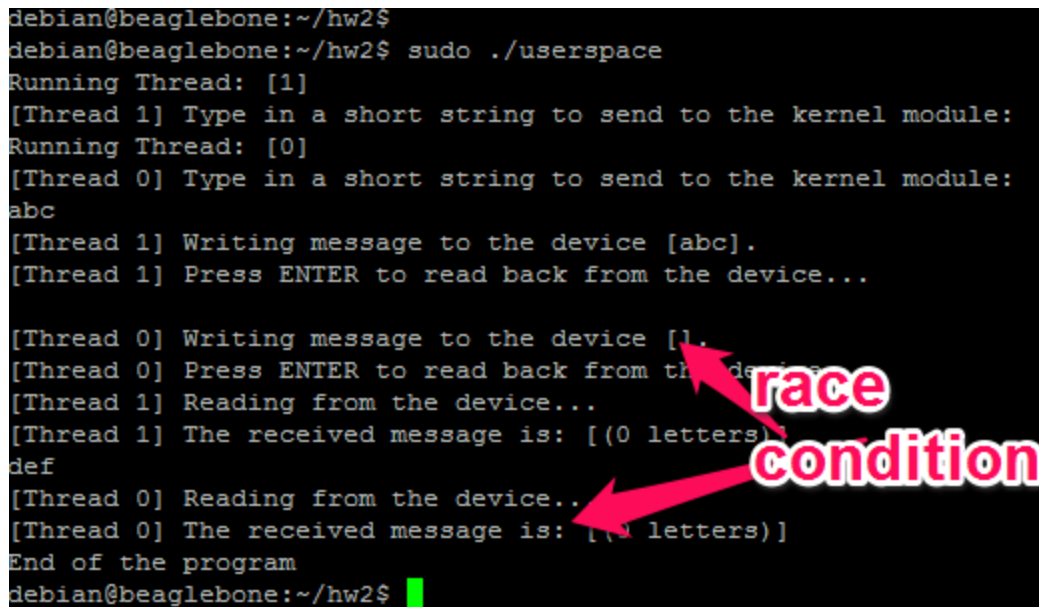
Use the threaded hello world source code to assist you in editing userspace.c:
Threaded Hello World Source Code

You will need to use at least the following three methods:
- pthread_create: (Man)ual Page   Creates a thread
- pthread_exit: (Man)ual Page   Terminates a thread
- Pthread_join: (Man)ual Page   Causes a thread to wait for another thread to terminate before proceeding.

If you did this part correctly, your output should look similar to the following. Notice the race condition! The threads are accesses the driver simultaneously and corrupting each other's data! We will fix this race condition in the kernel code.

Part #2 - Edit the kernel driver

Add a mutex lock to enable mutual exclusion and avoid the race condition mentioned above.

You will need the linux/mutex library to use mutexes in the kernel.
    #include <linux/mutex.h>

Declare a mutex called "driver_mutex" using the following statement.
    static DEFINE_MUTEX(driver_mutex);

You will need to implement the following rules in the appropriate locations..

- `mutex_init(&driver_mutex);`
  - Initializes the mutex lock dynamically at runtime
  - This should occur when the driver (Linux Kernel Module) is first loaded.
- `mutex_lock(&driver_mutex);`
  - Locks the mutex and begins access to the critical region. If the lock is being held by another process/thread, the process will sleep
  - In this assignment, you should lock the mutex when the driver is opened by a process
- `mutex_unlock(&driver_mutex);`
  - Unlocks the mutex and allows another process to enter the critical region.
  - In this assignment, you should unlock the mutex when the driver is released by a process
- `mutex_destroy(&driver_mutex);`
  - Destroys the dynamically allocated memory used for the mutex.
  - You should destroy the mutex when the driver (Linux Kernel Module) is unloaded

At this point, if you fixed the race condition, you should have output similar to the following. This code is considered thread-safe.

```
debian@beaglebone:~/hw2$ sudo insmod kernelDriver.ko
debian@beaglebone:~/hw2$ sudo ./userspace
Running Thread: [1]
[Thread 1] Type in a short string to send to the kernel module:
Running Thread: [0]
abc
[Thread 1] Writing message to the device [abc].
[Thread 1] Press ENTER to read back from the device...

[Thread 1] Reading from the device...
[Thread 1] The received message is: [abc(3 letters)]
[Thread 0] Type in a short string to send to the kernel module:
def
[Thread 0] Writing message to the device [def].
[Thread 0] Press ENTER to read back from the device...

[Thread 0] Reading from the device...
[Thread 0] The received message is: [def(3 letters)]
End of the program
debian@beaglebone:~/hw2$
```

Submission Instructions

- Submit a single tar ball (.tar file) that contains all of your source code and a working Makefile.
- Include a screenshot that shows your code output (from the console)

Grading Rubric

Component	Points
Properly Submitted: Submitted a Single tar ball containing source code and a single working Makefile. Submission also includes a screenshot of the final console output.	**10 points**
Formatting & Error Checking: Code is reasonably well structured, readable, properly indented and contains helpful comments. Code contains error checking, particularly for system calls. If an error occurs, displays a useful error message.	**10 points**
User Space Code Functionality: Code generates two threads. Each thread opens the device, sends and reads information from the device, closes the device and subsequently quits. The first thread to finish should wait for the second thread to complete before the code terminates.	**40 points**
Linux Kernel Module Functionality: Code has been edited to prevent race conditions using mutexes. Mutexes for init, lock, unlock and destroy are all used in the proper locations.	**40 points**
TOTAL	**100 points**

***NOTE: Programs that do not compile using the submitted Makefile will not receive credit