**Part I: The Mutual Exclusion Pattern in Pthreads**

The *Mutual Exclusion* pattern is used to avoid race conditions when different threads need to access the same (shared) memory location. While multiple threads can simultaneously *read* from the same memory location without conflicting, any thread that tries to *write* to a memory location while another thread is reading or writing causes a conflict. In particular:

* Read-write conflicts occur when any thread tries to write to a location from which other threads are reading.
* Write-write conflicts occur when any two threads try to write to the same memory location at the same time.

Such conflicts tend to produce non-deterministic behavior, which is usually a Bad Thing.

Sections of code where these conflicts occur are called *critical sections*. The *Mutual Exclusion* pattern is to prevent two threads from entering the critical section at the same time. The basic idea is to declare a shared variable that acts as a lock, and surround the critical section with calls to lock and unlock the lock:

...

*declare a "lock variable" shared by the threads*

...

*lock the "lock variable"*

*critical section*

*unlock the "lock variable"*

...

With this pattern, before the first thread to reach the critical section can enter, it must first lock the lock. Until that thread leaves the critical section and unlocks the lock, any other threads that try to enter the critical section will find that the lock is locked and be forced to wait.

To support the *Mutual Exclusion* pattern, the pthread library provides:

* The pthread\_mutex\_t type, which can be used to create "lock" variables.
* The pthread\_mutex\_lock() function, which a thread can use to lock a "lock variable". Any threads that perform this function when the lock is locked will block. you need to pass pthread\_mutex\_t type object to the function as:

pthread\_mutex\_t variable;

pthread\_mutex\_lock(&variable);

* The pthread\_mutex\_unlock() function, which can be used to unlock a "lock variable". If there are blocked threads waiting at the lock, exactly one of them will be unblocked and allowed to proceed. You need to pass pthread\_mutex\_t type object to the function as:

pthread\_mutex\_t variable;

pthread\_mutex\_unlock(&variable);

* Other functions: pthread\_mutex\_trylock( ), pthread\_mutex\_init, pthread\_mutex\_destroy – Refer to the Internet about these after you complete the exercises.

The first part of today's exercise is to download and explore the tank account simulation (mutualExclusion.c) from your instructors ftp server. The simulation contains a race condition that you must fix using the *Mutual Exclusion* pattern.

When you have fixed the race condition, you may you can continue to part two.

**Part II: (Please do this lab in LINUX)**

This week you will write a program that creates 3 threads. These threads will access a shared resource, an integer called buffer, one at a time. The buffer will initially be set to 0. They will each print their thread ID, process ID and the buffer's current value in one statement, then increment the buffer by one. Use a mutex to ensure this whole process is not interrupted.

Have the threads modify the buffer a total of 24 times. When each thread is done, it should return the number of times it modified the buffer to the main thread. The total number of modifications should be exactly 24.   
(Hint: the buffer may increase in value while you are waiting to access it. Make sure that you have code that checks the buffer's value after you get access, but before you work with it.)

The threads do not need to access the buffer in any particular order, but over multiple runs it should be obvious that all three are getting a fair chance. If one buffer seems to dominate over multiple runs, try introducing a sleep to simulate work between the mutex lock and unlock, just before the lock, or just after the unlock.   
(Tip: if sleep is too slow for you, try nanosleep - it's what I used for my sample runs.)

Did you need to use a sleep to get evenly distributed results? Where did it help the most? Speculate as to why.

Is it possible to control the order of access to the buffer with just a mutex? Justify your answer.

**Sample runs:**

$ ./exercise

TID: 3077897072 PID: 30656 Buffer: 0

TID: 3069504368 PID: 30656 Buffer: 1

TID: 3059014512 PID: 30656 Buffer: 2

TID: 3077897072 PID: 30656 Buffer: 3

TID: 3069504368 PID: 30656 Buffer: 4

TID: 3077897072 PID: 30656 Buffer: 5

TID: 3059014512 PID: 30656 Buffer: 6

TID: 3069504368 PID: 30656 Buffer: 7

TID: 3077897072 PID: 30656 Buffer: 8

TID: 3059014512 PID: 30656 Buffer: 9

TID: 3069504368 PID: 30656 Buffer: 10

TID: 3077897072 PID: 30656 Buffer: 11

TID: 3069504368 PID: 30656 Buffer: 12

TID: 3059014512 PID: 30656 Buffer: 13

TID: 3069504368 PID: 30656 Buffer: 14

TID: 3077897072 PID: 30656 Buffer: 15

TID: 3059014512 PID: 30656 Buffer: 16

TID: 3077897072 PID: 30656 Buffer: 17

TID: 3069504368 PID: 30656 Buffer: 18

TID: 3059014512 PID: 30656 Buffer: 19

TID: 3077897072 PID: 30656 Buffer: 20

TID: 3069504368 PID: 30656 Buffer: 21

TID: 3059014512 PID: 30656 Buffer: 22

TID: 3077897072 PID: 30656 Buffer: 23

TID 3077897072 worked on the buffer 9 times

TID 3069504368 worked on the buffer 8 times

TID 3059014512 worked on the buffer 7 times

Total buffer accesses: 24

$ ./exercise

TID: 3077978992 PID: 30660 Buffer: 0

TID: 3069586288 PID: 30660 Buffer: 1

TID: 3059096432 PID: 30660 Buffer: 2

TID: 3077978992 PID: 30660 Buffer: 3

TID: 3069586288 PID: 30660 Buffer: 4

TID: 3077978992 PID: 30660 Buffer: 5

TID: 3059096432 PID: 30660 Buffer: 6

TID: 3077978992 PID: 30660 Buffer: 7

TID: 3069586288 PID: 30660 Buffer: 8

TID: 3077978992 PID: 30660 Buffer: 9

TID: 3059096432 PID: 30660 Buffer: 10

TID: 3077978992 PID: 30660 Buffer: 11

TID: 3069586288 PID: 30660 Buffer: 12

TID: 3077978992 PID: 30660 Buffer: 13

TID: 3059096432 PID: 30660 Buffer: 14

TID: 3077978992 PID: 30660 Buffer: 15

TID: 3069586288 PID: 30660 Buffer: 16

TID: 3077978992 PID: 30660 Buffer: 17

TID: 3059096432 PID: 30660 Buffer: 18

TID: 3077978992 PID: 30660 Buffer: 19

TID: 3069586288 PID: 30660 Buffer: 20

TID: 3077978992 PID: 30660 Buffer: 21

TID: 3059096432 PID: 30660 Buffer: 22

TID: 3077978992 PID: 30660 Buffer: 23

TID 3077978992 worked on the buffer 12 times

TID 3069586288 worked on the buffer 6 times

TID 3059096432 worked on the buffer 6 times

Total buffer accesses: 24

**Deliverables for this exercise:**

* The code
* A few sample runs
* Your answers to the two questions