WEEK 4 – LAB 4 - UE15CS305 – Introduction to Operating System (IOS) Laboratory Problem Statements with Solutions

Problem Statement 4.1.

Write a C program to simulate Race Condition.

When two or more concurrently running threads/processes access a shared data item or resources and final results depends on the order of execution, we have race condition. Suppose we have two threads A and B as shown below. Thread A increase the share variable count by 1 and Thread B decrease the count by 1. If the current value of Count is 10, both execution orders yield 10 because Count is increased by 1 followed by decreased by 1 for the former case, and Count is decreased by 1 followed by increased by 1 for the latter. However, if Count is not protected by mutual exclusion, we might get difference results. Statements Count++ and Count-- may be translated to machine instructions as shown below: If both statements are not protected, the execution of the instructions may be interleaved due to context switching. More precisely, while thread A is in the middle of executing Count++, the system may switch A out and let thread B run.

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
LOAD Count	10			10
		LOAD Count	10	10
		SUB #1	9	10
		STORE Count	9	9
ADD #1	11			10
STORE Count	11			11

Similarly, while thread B is in the middle of executing Count--, the system may switch B out and let thread A run. Should this happen, we have a problem. The following table shows the execution details. For each thread, this table shows the instruction executed and the content of its register. Note that registers are part of a thread's environment and different threads have different environments. Consequently, the modification of a register by a thread only affects the thread itself and will not affect the registers of other threads. The last column shows the value of Count in memory. Suppose thread A is running. After thread A executes its LOAD instruction, a context switch switches thread A out and switches thread B in. Thus, thread B executes its three instructions, changing the value of Count to 9. Then, the execution is switched back to thread A, which continues with the remaining two instructions. Consequently, the value of Count is changed to 11!

The following shows the execution flow of executing thread ***B*** followed by thread ***A***, and the result is 9!

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

This example shows that without mutual exclusion, the access to a shared data item may generate different results if the execution order is altered. This is, of course, a ***race condition***. This race condition is due to no mutual exclusion.

Problem Statement 4.2.

Write a program to simulate race condition in Producer – Consumer problem.

Implement a main program that creates two threads: producer and consumer threads which executes producer and consumer functions respectively. The producer should produce an item and update the buffer. The consumer should consume an item and update the buffer. Consumer should wait if buffer is empty and producer should signal when the buffer has at least one item. You can assume it is bounded buffer so that producer needs to wait for buffer availability. Both the producer and consumer threads can be infinite loops and each should also randomly sleep to let the other proceed. The output should be the number of items in buffer along with the consumer/producer that is updating the buffer. This is an example that shows that without mutual exclusion, how race condition occurs between producer and consumer.

The structure of your program:

```
producer thread function()
        produce an item
        update the buffer
        consumer thread function()
                if buffer is empty, wait for producer
                        update the buffer
                :

        main function()
                initialize the semaphores

create consumer and producer threads
        // interprocess communication
        // gcc ipc.c -o ipc.exe –lpthread
```