**1) Run this program (multiple times) and note down observations from the code.**

(how many number of threads, execution order) 1\_scheduling.cpp

A. there are 2 user threads and a main thread, all are executing parallelly so execution order is different every time we execute the program

**2) What is the final value of x?**

On a single core system

On a multiple core system

When optimization is disabled / enabled

When the variable is declared as volatile

2\_global\_counter.cpp

A. single core: multithreading is possible, will take more time,

multicore : random values printed as critical section problem is not handles(the global counter)

optimization on (release mode) always 0

optimization off (debug mode) random answer

when using volatile int: random answer always (The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.)

**3) Reading: and write Summary.**

First 21 pages from http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

A. Summary

Chapter 1: - concept of synchronisation, serialisation, mutual exclusion

- models with complications - parallel model(multiprocessor)

- multithread

- reasons for complications - Non- determinism of sequence of execution

- shared variables (concurrent writes, concurrents updates, )

- solutions - serialisation with messages

- mutual excludion with messages

Chapter 2: - introduction to semaphore

- how we can implement a decided amount of concurrency through semaphores

- what the values of semaphore means (positive, 0, negative)

- a semaphore is created with a non- zero value signifying the amount of concurrency allowed

- threads are allowed only to increment or decrement the semaphore

- at any given time, the program should be independent of the actual value of the semaphore

- naming convensions of semaphore operations (signal and wait)

Chapter 3: Basic synchronization patterns

- signaling - way of communication between threads

- solves the serialisation problem

- Rendezvous - Ascenario where all threads have to reach a certain point after that they may proceed their own path

- Deadlock - A brief introduction

- mutex = mutual exclusion

- concept : only one thread holding the "mutex" can execute, reat have to wait

- implementation : semaphore initialised with value 1

- all threads before entering critical section, wait for mutex, and signal it to represent the release of critical section

- multiplex = a fixed amount of maximum no. of threads can access critical section simultaneously

- implemented by initialising semaphore with said no.

- all threads before entering critical section, wait for mutex, and signal it to represent the release of critical section

**4) What does the value of a semaphore means (+ve, -ve, 0)?**

A. positive: represents the number of threads that can decrement without blocking.

negative: represents the number of threads that have blocked and are waiting.

zero: represents there are no threads waiting, but if a thread tries to decrement, it will block.

**5) Rendezvous**

**Problem: Generalize the signal pattern so that it works both ways. Thread A has to wait for Thread B and vice versa. Given the below code, guarantee that a1 happens before b2 and b1 happens before a2 (we don’t care about the order of a1 and b1.)**

**Thread A Thread B**

**statement a1 statement b1**

**statement a2 statement b2**

A. aArrived = semaphore(0)

bArrived = semaphore(0)

Thread A Thread B

statement a1 statement b1

aArrived.signal() bArrived.signal()

bArrived.wait() aArrived.wait()

statement a2 statement b2

**6) Run both these programs and understand the difference between them.**

**2\_global\_counter\_mutex.cpp**

**2\_global\_counter\_atomic.cpp**

A. global\_counter\_atomic:

- output is 0

- increment and decrement operations are done by InterlockedIncrement and InterlockedDecrement respectively

- these functions take the parameter as volatile, so the compiler restricts doing any kind of optimisation (e.g copying them in registers and use it later on) and we always have the latest vale stored in the memory

- these functions increase or decreace the specified 32-bit variable, pointed by pointer taken as parameter, as atomic operation, i.e, full memory barriers and locking mechanisms are used to prevent any other thread to access this memory location.

- here both threads are running parallely, with each increment and decrement involves acquiring and releasing locks.

- hence critical section problem is solved

global\_counter\_mutex

- output is 0

- here increment and decrement are done using unary operators ++ and --

- mutual exclusion is explicitly provided by functions WaitForSingleObject() and ReleaseMutex() which impplement "wait and signal".

- here a given thread executes completely before other thread starts executing.

- here each thread aquire a lock before the loop and releases the lock after the loop.

- hence critical section problem is solved

**7) Understand the program, run it and note down observations**

**3\_ProducerConsumer.cpp**

A. ProducerThreadProc()

- thread which infinitely keeps on producing new products and adds them is a global queue

- just before adding products ro queue

- it acquires a lock on the queue via EnterCriticalSection(). This lock is declared critical section so other producers cant aquire this lock when a producer is adding to the global queue.

- it checks if queue is full, if so, it waits in an infinite loop for at least one consumer to consume a product and decrement the queue size

ConsumerThreadProc()

- thread which infinitely keeps on consuming products decrements the global queue

- just before decrementing the queue

- it acquires a lock on the queue via EnterCriticalSection().

- it checks if queue is empty, if so, it waits in an infinite loop for at least one producer to produce a product and increment the queue size

this is a critical section problem where a gobal buffer of products is there where producers add products and while doing so, consumers are not allowed to consume, and vise versa.