**Introduction to thread in c++ (c++11)**

// QUESTIONS

// 1. What do you understand by thread and give one example in C++?

// ANSWER

// 0. In evey application there is a default thread which is main(), in side this we create other threads.

// 1. A thread is also known as lightweight process. Idea is achieve parallelism by dividing a process into multiple threads.

// For example:

// (a) The browser has multiple tabs that can be different threads.

// (b) MS Word must be using multiple threads, one thread to format the text, another thread to process inputs (spell checker)

// (c) Visual Studio code editor would be using threading for auto completing the code. (Intellicence)

// WAYS TO CREATE THREADS IN C++11

// 1. Function Pointers

// 2. Lambda Functions

// 3. Functors

// 4. Member Functions

// 5. Static Member functions

// REQUIREMENT

// Find the addition of all odd number from 1 to 1900000000 and all even number from 1 to 1900000000

**Use Of join(), detach() and joinable() In Thread In C++ (C++11)**

// JOIN NOTES

// 0. Once a thread is started we wait for this thread to finish by calling join() function on thread object.

// 1. Double join will result into program termination.

// 2. If needed we should check thread is joinable before joining. ( using joinable() function)

// DETACH NOTES

// 0. This is used to detach newly created thread from the parent thread.

// 1. Always check before detaching a thread that it is joinable otherwise we may end up double detaching and

// double detach() will result into program termination.

// 2. If we have detached thread and main function is returning then the detached thread execution is suspended.

// NOTES:

// Either join() or detach() should be called on thred object, otherwise during thread object�s destructor it will

// terminate the program. Because inside destructor it checks if thread is still joinable? if yes then it terminates the program.

**Mutex In C++ Threading | Why Use Mutex | What Is Race Condition And How To Solve It? | What Is Critical Section**

// Mutex: Mutual Exclusion

// RACE CONDITION:

// 0. Race condition is a situation where two or more threads/process happened to change a common data at the same time.

// 1. If there is a race condition then we have to protect it and the protected section is called critical section/region.

// MUTEX:

// 0. Mutex is used to avoid race condition.

// 1. We use lock() , unlock() on mutex to avoid race condition

**std::mutex::try\_lock() On Mutex In C++11 Threading**

// 0. try\_lock() Tries to lock the mutex. Returns immediately. On successful lock acquisition returns true otherwise returns false.

// 1. If try\_lock() is not able to lock mutex, then it doesn't get blocked that's why it is called non-blocking.

// 2. If try\_lock is called again by the same thread which owns the mutex, the behavior is undefined.

// It is a dead lock situation with undefined behaviour. (if you want to be able to lock the same mutex by same thread

// more than one time the go for recursive\_mutex)

There are so many try\_lock function:

1. std::try\_lock

2. std::mutex::try\_lock

3. std::shared\_lock::try\_lock

4. std::timed\_mutex::try\_lock

5. std::unique\_lock::try\_lock

6. std::shared\_mutex::try\_lock

7. std::recursive\_mutex::try\_lock

8. std::shared\_timed\_mutex::try\_lock

9. std::recursive\_timed\_mutex::try\_lock

**// TOPIC: std::try\_lock() On Mutex In C++11 Threading**

// 0. std::try\_lock() tries to lock all the mutexes passed in it one by one in given order.

// 1. On success this function returns -1 otherwise it will return 0-based mutex index number which it could not lock.

// 2. If it fails to lock any of the mutex then it will release all the mutex it locked before.

// 3. If a call to try\_lock results in an exception, unlock is called for any locked objects before rethrowing.

**// TOPIC: Timed Mutex In C++ Threading (std::timed\_mutex)**

// We have learned Mutex, Race Condition, Critical Section in previous video.

// NOTES:

// 0. std::timed\_mutex is blocked till timeout\_time or the lock is aquired and returns true if success

// otherwise false.

// 1. Member Function:

// a. lock

// b. try\_lock

// c. try\_lock\_for ---\ These two functions makes it different from mutex.

// d. try\_lock\_until ---/

// e. unlock.

**// TOPIC: Recursive Mutex In C++ (std::recursive\_mutex)**

// NOTES:

// 0. It is same as mutex but, Same thread can lock one mutex multiple times using recursive\_mutex.

// 1. If thread T1 first call lock/try\_lock on recursive mutex m1, then m1 is locked by T1, now

// as T1 is running in recursion T1 can call lock/try\_lock any number of times there is no issue.

// 2. But if T1 have aquired 10 times lock/try\_lock on mutex m1 then thread T1 will have to unlock

// it 10 times otherwise no other thread will be able to lock mutex m1.

// It means recursive\_mutex keeps count how many times it was locked so that many times it should be unlocked.

// 3. How many time we can lock recursive\_mutex is not defined but when that number reaches and if we were calling

// lock() it will return std::system\_error OR if we were calling try\_lock() then it will return false.

// BOTTOM LINE:

// 0. It is similar to mutex but have extra facitility that it can be locked multiple time by same thread.

// 1. If we can avoid recursive\_mutex then we should becuase it brings overhead to the system.

// 2. It can be used in loops also.

**// TOPIC: lock\_guard In C++ (std::lock\_guard<mutex> lock(m1))**

// NOTES:

// 0. It is very light weight wrapper for owning mutex on scoped basis.

// 1. It aquires mutex lock the moment you create the object of lock\_guard.

// 2. It automatically removes the lock while goes out of scope.

// 3. You can not explicitly unlock the lock\_guard.

// 4. You can not copy lock\_guard.

**// TOPIC: unique\_lock In C++ (std::unique\_lock<mutex> lock(m1))**

// NOTES:

// 1. The class unique\_lock is a mutex ownership wrapper.

// 2. It Allows:

// a. Can Have Different Locking Strategies

// b. time-constrained attempts at locking (try\_lock\_for, try\_lock\_until)

// c. recursive locking

// d. transfer of lock ownership (move not copy)

// e. condition variables. (See this in coming videos)

// Locking Strategies

// TYPE EFFECTS(S)

// 1. defer\_lock do not acquire ownership of the mutex.

// 2. try\_to\_lock try to acquire ownership of the mutex without blocking.

// 3. adopt\_lock assume the calling thread already has ownership of the mutex.

**TOPIC: Condition Variable In C++ Threading**

// NOTES:

// 1. Condition variables allow us to synchronize threads via notifications.

// a. notify\_one();

// b. notify\_all();

// 2. You need mutex to use condition variable

// 3. Condition variable is used to synchronize two or more threads.

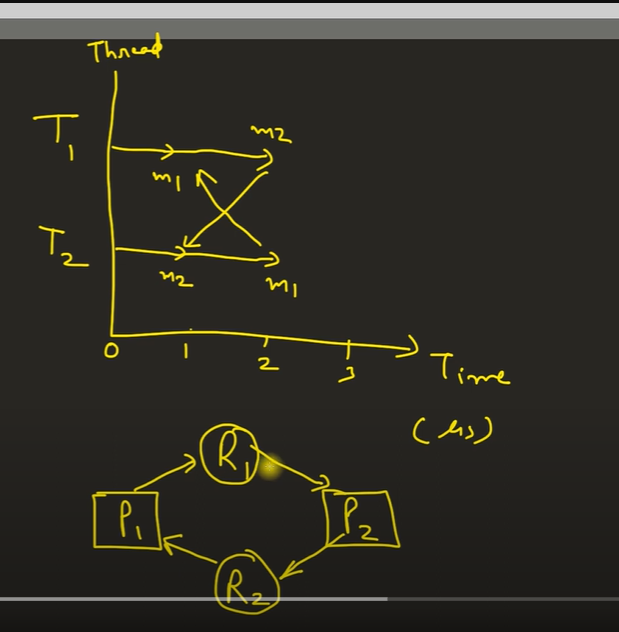
// 4. Best use case of condition variable is Producer/Consumer problem.

// 5. Condition variables can be used for two purposes:

// a. Notify other threads

// b. Wait for some condition

**Deadlock**

****

**// TOPIC: Thread OR Process Synchronization**

// NOTE: we will use only thread examples to explain this topic.

// POINTS:

// 1.0 Thread Or Process synchronize to access critical section.

// 2.0 Critical section is one or collection of program statements which should be executed by only one thread or process

// at a time.

**// TOPIC: std::lock() In C++11**

// It is used to lock multiple mutex at the same time.

// IMPORTANT:

// syntax---> std::lock(m1, m2, m3, m4);

// 1. All arguments are locked via a sequence of calls to lock(), try\_lock(), or unlock() on each argument.

// 2. Order of locking is not defined (it will try to lock provided mutex in any order and ensure that

// there is no dead lock).

// 3. It is a blocking call.

// [Example:0] --> No deadlock.

// Thread 1 Thread 2

// std::lock(m1,m2); std::lock(m1,m2);

// [Example:1] --> No deadlock.

// Thread 1 Thread 2

// std::lock(m1, m2); std::lock(m2, m1);

// [Example:2] --> No deadlock.

// Thread 1 Thread 2

// std::lock(m1, m2, m3, m4); std::lock(m3, m4);

// std::lock(m1, m2);

// [Example:3] --> Yes, the below can deadlock.

// Thread 1 Thread 2

// std::lock(m1,m2); std::lock(m3,m4);

// std::lock(m3,m4); std::lock(m1,m2);

**// TOPIC: std::future and std::promise in threading.**

// NOTES:

// 1. std::promise

// a. Used to set values or exceptions.

// 2. std::future

// a. Used to get values from promise.

// b. Ask promise if the value is available.

// c. Wait for the promise.

**TOPIC: std::async**

NOTES:

1. It runs a function asynchronously (potentially in a new thread) and returns a std::future that

will hold the result.

2. There are three launch policies for creating task:

a. std::launch::async **-> creates threading**

b. std::launch::deffered **-> does not create a thread**

c. std::launch::async | std::launch::deffered **-> let the computer decide**

HOW IT WORKS:

1. It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.

2. Then passes the std::promise object to thread function and returns the associated std::future object.

3. When our passed argument function exits then its value will be set in this promise object,

so eventually return value will be available in std::future object.

SIDE NOTES:

1. We can send functor and lambda function as callback to std::async, it will work the same.

**// TOPIC: Producer/Consumer OR Bounded Buffer Problem In C++11 Threading**

// THE PROBLEM STATEMENT:

// 1. Producer will produce and consumer will consume with synchronization of a common buffer.

// 2. Until producer thread produces any data consumer thread can't consume.

// 3. Threads will use condition\_variable to notify each other.

// 4. We need mutex if we use condition\_variable because CV waits on mutex.

// 5. This is one of the example of producer consumer there are many.

// PRODUCER thread steps:

// 1. lock mutex, if success then go ahead otherwise wait for mutex to get free.

// 2. check if buffer is full and if it is full then unlock mutex and sleep, if not then go ahead and produce.

// 3. insert item in buffer.

// 4. unlock mutex.

// 5. notify consumer.

// CONSUMER thread steps:

// 1. lock mutex, if success then go ahead and consume otherwise wait for mutex to get free.

// 2. check if buffer is empty and if it is, then unlock the mutex and sleep, if not thean go ahead and consume.

// 3. consume item from buffer.

// 4. unlock mutex.

// 5. notify producer.

// IMP:

// If you are talking about producer consumer then they have to notify each other.

**Sleep vs Wait**

What Is SLEEP :

0. “I’m done with my time-slice, and please don’t give me another one for at least n milliseconds.” The OS doesn’t even try to schedule the sleeping thread until requested time has passed.

1. It will keep the lock and sleep.

2. Sleep is directly to thread, it is a thread function.

What Is WAIT :

0. “I’m done with my time-slice. Don’t give me another time-slice until someone calls notify().” As with sleep(), the OS won’t even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs).

1. It releases the lock and wait.

2. Wait is on condition variable, it is like there is a condition variable in a thread and wait is applied to that CV but it ends up putting thread in waiting state.

**Thread Synchronization**

**Binary Semaphore**

**// TOPIC: Semaphore In C++**

// POINTS:

// 1.0 Semaphore is a signaling mechanism (T1 saying i am done T2 please carry on).

// 2.0 There are two types of semaphores

// a. Binary Semaphore

// b. Counting Semaphore

// BINARY SEMAPHORE:

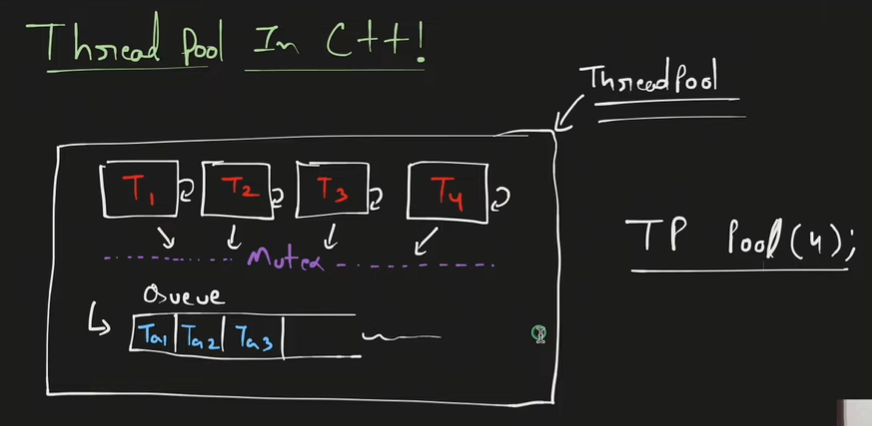
// 1.0 Binary Semaphore means there are two state 0 or 1.

// 2.0

**Mutex vs Semaphore**

|  |  |
| --- | --- |
| **MUtex** | **Semaphore** |
| Allows only one thread in critical section | Allows collection of thread in critical section with shared resources |
| Locked/ unlocked function | Count type locking, since aloows multiple thread so maintains a count |
| Locking mechanism | Signalling mechanism |
| Primarly used for mutual exclusion, allowing only one thread to access the critical section | Pool of thread and n resources(buffers etc.), now semaphores allows n threads to get allocated at a time , uses counter to track. |

**Thread pool**

****

**Atomic variables**