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

ANSWER 0. In every 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

In C++, a functor, also known as a function object, is an object that behaves like a function. Functors are instances of classes or structs that implement the function call operator **operator()**, allowing objects of that class to be used in a way that resembles calling a function. Functors are often used for custom comparisons, sorting, and other operations where you need more control or custom behavior than what standard functions or operators provide.

2. Lambda Functions

3. Functors

4. Member Functions

5. Static Member functions

1 functon pointers: eg:-

void fun(int x){

while(x-- > 0){ //(x-- >0)

cout<<x<<" ";

}

}

int main(){

thread t1(fun,10);

thread t2(fun,12);

t1.join(); // to join the thread with main thread

return 0;}

2) lambda expression:--

C++ 11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows.

Lambda: <https://www.programiz.com/cpp-programming/lambda-expression>

Lambda is just a way for the programmers to define a function without literally having to define a function.

Trivk to remember :

[] () {}

Capture list fn arguments fn body

3) Functor objects:

class base

{

public:

void operator() (int x)

{

while(x-- >0)

{

cout<<x<<" ";

}

}

};

int main()

{

thread t1((base()),10);

t1.join(); // to join the thread with main thread

return 0;

}

4) Non static member functions :

class base{

public:

int ope(int x) {

while(x-->0){

cout<<x<<" ";}

}

};

int main(){

base b;

thread t(&base::ope, &b ,10);

t.join();

}

5) static member functions::

class base{

public:

static void ope(int x) {

while(x-->0){

cout<<x<<" ";}

}

};

int main(){

thread t(&base::ope,10);

t.join(); //else gives an exception:terminate called without an exception

}

Notes on join(), detactch(), joinable()

**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)

Eg:join() and joinable()

#include <iostream>

#include <chrono>

#include <thread>

using namespace std;

void run(int x)

{

while(x>0)

{

cout<<x<<" "<<"thread"<<endl;

x--;

}

cout<<"thread complete"<<endl;

this\_thread::sleep\_for(chrono::seconds(5));

}

int main()

{

cout<<"first"<<endl;

std::thread t1(run,10);

t1.join();

if(t1.joinable())

t1.join();

cout<<"done"<<endl;

return 0;

}

**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 thread 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.

Eg of detach();

#include <iostream>

#include<thread>

#include<chrono>

using namespace std;

void run(int x){

while(x-->0)

cout<<"heloo"<<endl;

cout<<"Thread finished:";

this\_thread::sleep\_for(chrono::seconds(5));

}

int main(){

thread t(run,15);

cout<<"main()"<<endl;

t.detach();

if(t.joinable())

{

t.detach();

}

cout<<"aftetr main():"<<endl;

return 0;}

Always write detach or join for thread object iut will result in termination of the program.

Because the momentb u create thread that thread is marked as joinable and then later either u apply

Detach or join()on that thread then it will automatically become non-joinable:

Mutex: Mutual Exclusion RACE CONDITION:

0. Race condition is a situation where two or more threads/process happens to change a common data at the same time. (not access)

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.

Eg:’

#include <iostream>

#include<thread>

#include<chrono>

#include<mutex>

using namespace std;

int x=0;

mutex m;

void run(){

m.lock();

++x;

m.unlock();

}

int main(){

thread t(run);

thread t1(run);

t.join();

t1.join();

cout<<x<<endl;}

TOPIC: 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 behavior. (if you want to be able to lock the same mutex by same thread more than one time the go for recursive\_mutex)

Eg:-> #include <iostream>

#include<thread>

#include<chrono>

#include<mutex>

using namespace std;

int x=0;

mutex m;

void run(){

for(int i=0;i<100000;i++){

if ( m.try\_lock()){

++x;

m.unlock();

}

}

}

int main(){

thread t1(run);

thread t2(run);

t1.join();

t2.join();

cout<<x<<endl;}

o/p will always be less then 1lakh, because bpth threds will be trying for it.

If we want 2 lakh then we need to use just lock instead of try\_lock()

define mutex:

The mutex class is a synchronization primitive that can be used to protect

shared data from being simultaneously accessed by multiple threads.

----------------------------------------------------------------------------------------------------------

how to identify CS:

when there is a commom object between two threads , that is CS and if we are only reading that then it is not a CS

only if we write it is a CS.

------------------------------------------------------------------------------------------------

it is like 1/2 rule

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

----------------------------------------------------------------------------------------------------------

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

Few things to remember about the try\_lock is as follows:

0. std::try\_lock() tries to lock all the mutex 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 re-throwing.

The actual use of try::lock is it can try to lock multiple mutex objects at the same time

big pgm:

eg:bucket x y and add x+y in third thread;

7) Timed Mutex:

few points to remember about timed mutex is as follows:

0. std::timed\_mutex is blocked till timeout or the lock is acquired and returns true if success otherwise false.

1. Member Function:

a. lock

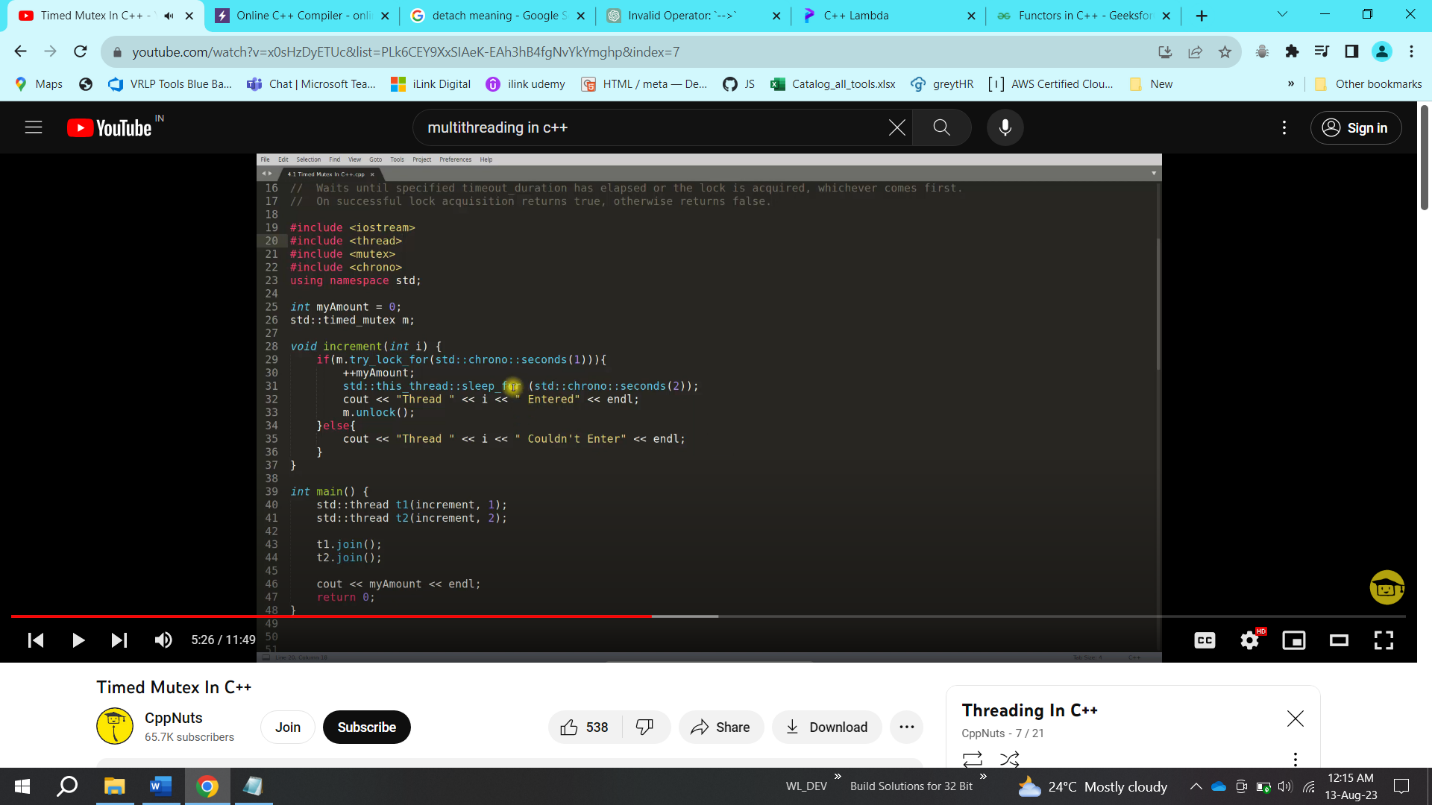
b. try\_lock

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

d. try\_lock\_until ---/

e. unlock

EXAMPLE: try\_lock\_for(); Waits until specified timeout\_duration has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.



o/p

thread 2 couldn’t enter

thread 1 entered,

1

-----------------------------------------------------------------------------------------------

8) Recursive mutex:

Recursive Mutex In C++ (std::recursive\_mutex). it is very helpful when we have to put lock in recursive function calls. Few points to remember about recursive mutex is as follows:

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 acquired 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 facility that it can be locked multiple time by same thread. 1. If we can avoid recursive\_mutex then we should because it brings overhead to the system. 2. It can be used in loops also

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

std::recursive\_mutex m1;

int a = 0;

void fn(char c, int n)

{

if(n<=0)

return;

m1.lock();

cout<<a++<<" ";

fn(c,n-1);

m1.unlock();

cout<<endl;

cout<<"unlock by thread: "<<c; //it unlocks same no of times it had locked:

}

int main()

{

thread t1(fn,'f',10);

t1.join();

}

-----------------------------------------------------------------------------------------------

9)Few points to remember about the lock\_guard is as follows:

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

1. It acquires 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.

std::lock\_guard<mutex> lock(m1) basically to avoid wrtiing

m1.lock() and m1.unlock()

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

int num = 10;

mutex m;

void run(int x)

{

Std:: lock\_guard<mutex> lock(m);

// here m is mutex and d is objectof class lock\_guard

cout<<num+x<<endl;

}

int main()

{

thread t1(run,10);

thread t2(run,100);

t1.join();

t2.join();

return 0; }

we use lock\_guard, when we want to essure that the mutex for which lock has been acquired should be locked until it goes out of scope

10) UNIQUE LOCK

Few important points to remember about the unique\_lock is as follows:

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)

3.No need to unlock()

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.

Eg:std::unique\_lock<mutex> obj(m1,defer\_lock);

eg: only using unique lock

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

int num = 10;

mutex m;

void run(int x)

{

std::unique\_lock<mutex>lock(m1); //automay cllas lock for m1

for(int i=0;i<10;i++)

{

Cout<<m++;

}

//lock.unlock() is not needed as it will be unlocked in destructor of //unique\_lock();

int main()

{

thread t1(run,10);

thread t2(run,100);

t1.join();

t2.join();

return 0;

}

eg:

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

int num = 10;

mutex m;

void run(int x)

{

std::unique\_lock<mutex>lock(m, std::defer\_lock); //do not lock automatically needs to do it manually, also we can add some code here,this is the main difference beweteen lock\_guard and unique lock, the first auromatically locks when it is called, but this one needs manual lock declaration.

lock.lock();

cout<<num+x<<endl;

}

int main()

{

thread t1(run,10);

thread t2(run,100);

t1.join();

t2.join();

return 0;

}

In summary, choose **std::lock\_guard** when you need simple, automatic lock management for the entire scope of a block or function and don't need to unlock the mutex prematurely. Use **std::unique\_lock** when you require more flexibility, such as conditional locking/unlocking, manual locking/unlocking, or transferring ownership of a lock between scopes or functions. The choice depends on the specific requirements of your multithreaded code.

1. **std::lock\_guard**:
   * **Use Case**: **std::lock\_guard** is suitable when you want to lock a mutex for a specific scope and release the lock automatically when the scope exits.
   * **Ownership**: **std::lock\_guard** takes ownership of the lock immediately upon construction and releases it automatically upon destruction, ensuring that the lock is always released when the **std::lock\_guard** goes out of scope.
   * **Ease of Use**: It's straightforward to use and ensures that you don't forget to release the lock, making it a good choice for most cases where you simply need to protect a critical section of code.
   * **Example**:

cppCopy code

std::mutex myMutex; { std::lock\_guard<std::mutex> lock(myMutex); // Critical section } // Lock is automatically released when 'lock' goes out of scope

1. **std::unique\_lock**:
   * **Use Case**: **std::unique\_lock** is more versatile and is suitable when you need more fine-grained control over the locking and unlocking of a mutex.
   * **Ownership**: **std::unique\_lock** allows you to acquire and release the lock multiple times within its scope. You can explicitly release the lock and reacquire it as needed.
   * **Flexibility**: It provides additional features like condition variable support (**wait** and **notify**), deferred locking (you can acquire the lock later), and timed locking (lock with a timeout).
   * **Example**:

cppCopy code

std::mutex myMutex; std::unique\_lock<std::mutex> lock(myMutex); // Critical section lock.unlock(); // Explicitly release the lock // Other non-critical work lock.lock(); // Reacquire the lock // More critical section

**Choosing Between std::lock\_guard and std::unique\_lock**:

* Use **std::lock\_guard** when you want a simple and safe way to manage locks for a specific scope without the need for advanced features like condition variables or manually releasing and reacquiring the lock.
* Use **std::unique\_lock** when you need more control over locking and unlocking, want to use condition variables, or have complex scenarios where you need to release and reacquire the lock multiple times.

In general, if you're unsure which one to use, **std::lock\_guard** is often a good default choice because it's straightforward and ensures that you won't forget to release the lock when it goes out of scope. However, for more complex synchronization scenarios, **std::unique\_lock** offers greater flexibility.

Top of Form

11)condition variable:

VI if u want to synchronise some threads on some conditions

Few important points to remember while using "condition variables" are as follows:

1. Condition variables allow us to synchronise threads via notifications.

a. notify\_one();

b. notify\_all();

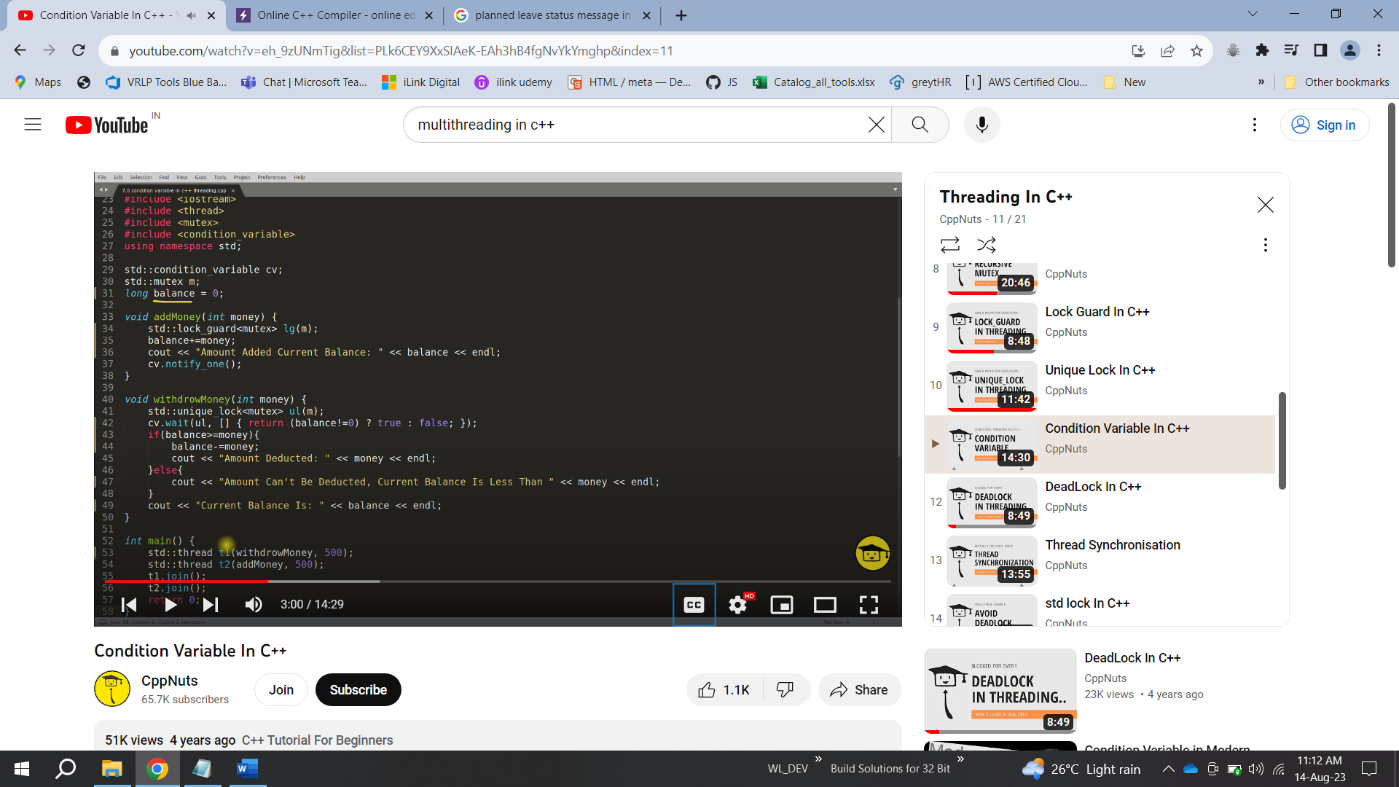
2. You need mutex to use condition variable

3. Condition variable is used to synchronise 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

#include <iostream>

#include <mutex>

#include <thread>

#include <condition\_variable>

using namespace std;

std::mutex m1;

int balance = 0;

std::condition\_variable cv;

void addmoney(int money)

{

lock\_guard<mutex> lock(m1);

balance+=money;

cout<<"the balkance is:"<<balance<<endl;

cv.notify\_one();

}

void withdrawmoney(int money)

{

unique\_lock<mutex> ul(m1);

cv.wait(ul,[] {return (balance!=0) ? true: false; });

if(balance>money)

{

cout<<"the remianing balance is:" <<balance-money<<endl;

}

else

cout<<"balance-money is less then requested:"<<endl;

}

int main()

{

thread t1(withdrawmoney, 1000);

thread t2(addmoney, 2000);

t1.join();

t2.join();

}

Print odd even sequence:

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <condition\_variable>**

**using namespace std;**

**mutex mtx;**

**condition\_variable cv;**

**bool isOdd = false;**

**void odd()**

**{**

**for(int i=1; i<=9; i+=2)**

**{**

**unique\_lock<mutex> lock(mtx);**

**cv.wait(lock, [] { return !isOdd; });**

**cout << i << " ";**

**isOdd = true;**

**lock.unlock();**

**cv.notify\_all();**

**}**

**}**

**void even()**

**{**

**for(int i=2; i<=8; i+=2)**

**{**

**unique\_lock<mutex> lock(mtx);**

**cv.wait(lock, [] { return isOdd; });**

**cout << i << " ";**

**isOdd = false;**

**lock.unlock();**

**cv.notify\_all();**

**}**

**}**

**int main()**

**{**

**thread t1(even);**

**thread t2(odd);**

**t1.join();**

**t2.join();**

**cout << endl;**

**}**

A **mutex** is a synchronization object. You acquire a lock on a mutex at the beginning of a section of code, and release it at the end, in order to ensure that no other thread is accessing the same data at the same time. A mutex typically has a lifetime equal to that of the data it is protecting, and that one mutex is accessed by multiple threads.

A **lock object** is an object that encapsulates that lock. When the object is constructed it acquires the lock on the mutex. When it is destructed the lock is released. You typically create a new lock object for every access to the shared data

A mutex is an object which can be locked. A lock is the object which maintains the lock. To create a lock, you need to pass it a mutex.

12

------------------------------------------------------------------------------------------------13)

Thread Synchronisation

"Thread OR Process Synchronisation" is as follows:

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.

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

int num = 10;

mutex m;

void run(int x)

{

m.lock();

cout<<num+x<<endl;

m.unlock();

}

int main()

{

thread t1(run,10);

thread t2(run,100);

t1.join();

t2.join();

return 0;

}

-------------------------------------------------------------------------------------------------

14) std\_lock

std::lock() In C++11 It is used to lock multiple mutex at the same time.

And the syntax is as follow:

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.

i.e. std\_lock will change the order in which mutexes will be locked, and unlock the other mutexes

[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);

in general if u dont want dedalock then call the locks in similar order

std::lock(m1, m2 )

std:lock(m2,m1)

works fine for std::lock

but for m.lock() ->it will be a deadlock

**std::try\_lock** and **std::lock** are both functions used for managing locks and synchronization in multi-threaded C++ programs. However, they serve different purposes and have distinct behaviors. Here are the key differences between **std::try\_lock** and **std::lock**:

1. **Purpose**:
   * **std::try\_lock**: This function attempts to acquire locks on multiple mutexes simultaneously. It returns immediately, indicating whether it successfully acquired all the locks or not. It is non-blocking.
   * **std::lock**: This function is used to acquire locks on multiple mutexes in a way that guarantees deadlock avoidance. It may block the calling thread until all the locks can be obtained.
2. **Blocking Behavior**:
   * **std::try\_lock**: Does not block the calling thread. If it cannot acquire all the locks, it returns immediately without waiting.
   * **std::lock**: May block the calling thread if it cannot obtain all the locks immediately. It will wait until it can acquire all the locks without causing a deadlock.
3. **Return Value**:
   * **std::try\_lock**: Returns an **int** representing the number of locks successfully acquired. If it successfully acquires all locks, it returns the number of locks. If it cannot acquire all locks, it returns -1.
   * **std::lock**: Returns void. It either successfully acquires all locks or blocks until it can, without returning any value.
4. **Usage**:
   * **std::try\_lock** is often used when you want to attempt to acquire multiple locks without blocking to check if they are available. It's useful when you want to avoid potential deadlock situations.
   * **std::lock** is used when you need to acquire multiple locks in a way that guarantees that all locks are acquired together to avoid deadlocks. It's typically used when you need exclusive access to multiple resources.

-------------------------------------------------------------------------------------------------

15) Promise and Future In c++:

we will learn about how std::promise and std::future and how they work together to synchronise threads. Basically std::promise is sent to the called thread and once the value is ready we set that value in promise object, now at calling thread side we get that value using std::future object which was created using std::promise object before sending it to the called thread. And this is how we receive value from one thread to another in synchronisation.

The main objective of creatig promise and future is get the value from thread function in main function:

We do this by,

1. Create the promise object and crating future from that promise object

A screen shot of a computer

Description automatically generated

In C++, `std::promise` and `std::future` are synchronization primitives that allow communication between threads by enabling one thread to produce a value or an exception and another thread to retrieve that value or exception asynchronously. These classes are part of the C++ Standard Library and are used for implementing asynchronous and concurrent programming patterns.

Here's an explanation of `std::promise` and `std::future`:

1. \*\*std::promise\*\*:

- `std::promise` is a class that represents a promise of a value (or an exception) that will be provided in the future.

- It is typically used by the producer thread to set a value or an exception that will be made available for retrieval by another thread.

- A promise is associated with a `std::future` object, which allows the consumer thread to retrieve the promised value.

- Promises are often used when you want to perform a task asynchronously and provide a result to another thread.

2. \*\*std::future\*\*:

- `std::future` is a class that represents a value (or an exception) that may be available in the future.

- It is typically used by the consumer thread to retrieve the value (or exception) that was promised by the producer thread.

- Futures provide a mechanism for waiting asynchronously for the result of a task to become available.

- Futures can be used to synchronize threads and retrieve the result of an asynchronous operation without blocking.

Here's a basic example of how `std::promise` and `std::future` can be used:

```cpp

#include <iostream>

#include <thread>

#include <future>

void producer(std::promise<int>& p) {

// Simulate some time-consuming operation

std::this\_thread::sleep\_for(std::chrono::seconds(2));

int result = 42;

p.set\_value(result); // Set the value to be retrieved

}

int main() {

std::promise<int> p;

std::future<int> f = p.get\_future(); // Associate the future with the promise

std::thread producer\_thread(producer, std::ref(p)); // Start a producer thread

// Do some work in the main thread

std::cout << "Main thread is doing some work..." << std::endl;

// Wait for the result from the producer thread

int result = f.get(); // Blocking call until the value is available

std::cout << "Result received from the producer: " << result << std::endl;

producer\_thread.join(); // Wait for the producer thread to finish

return 0;

}

```

In this example, the `std::promise` is used by the producer thread (`producer` function) to promise a value (42) that will be set in the future. The main thread associates a `std::future` with the promise and then waits for the result using `f.get()`. This allows the main thread to continue doing other work while waiting for the producer thread to complete.

`std::promise` and `std::future` provide a powerful mechanism for asynchronous programming and inter-thread communication in C++. They are commonly used in scenarios where you need to perform tasks concurrently and retrieve results or handle exceptions asynchronously.

#include <iostream>

#include <mutex>

#include <thread>

#include <algorithm>

#include <future>

using namespace std;

void findodd(promise<int>&oddsumpromise, int start, int end)

{

int oddsum = 0;

for(int i=start;i<=end;i++)

{

if(i&1)

{

oddsum+=i;

}

}

oddsumpromise.set\_value(oddsum);

}

int main()

{

promise<int> oddsum;

future<int> oddfuture = oddsum.get\_future();

cout<<"thread done:"<<endl;

int start=0, end =10;

thread t1(findodd, ref(oddsum), start,end);

cout<<oddfuture.get();

t1.join();

return 0;

}

16) std::async in c++

This is the first first type of thread from which we actually return something

3 different launch policies:

1. std::launch::async // create a sperate thraed

2. std::launch::deffered // does not create a thtrad

3. std::launch::async | std::launch::deffered // upto machine, first try to create a thread, if not doesn’t create a thread

#include <iostream>

#include <mutex>

#include <thread>

#include <algorithm>

#include <future>

using namespace std;

int findodd(int start, int end)

{

int oddsum = 0;

for(int i=start;i<=end;i++)

{

if(i&1)

{

oddsum+=i;

}

}

return oddsum;

}

int main()

{

int start=0, end= 10;

cout<<"Thread id is:"<<std::this\_thread::get\_id()<<endl;

std::future<int> res = async(std::launch::async,findodd,start,end);

cout<<"the result is:"<<res.get();

return 0;

}

17) Producer and consumer problem:

Producer Consumer OR Bounded Buffer Problem THE PROBLEM STATEMENT:

1. Producer will produce and consumer will consume with synchronisation 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 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 then go ahead and consume.

3. consume item from buffer.

4. unlock mutex.

5. notify producer. IMP: Producer and Consumer have to notify each other upon completion of their job.

#include <iostream>

#include <mutex>

#include <condition\_variable>

#include <thread>

#include <deque>

using namespace std;

std::mutex m;

condition\_variable cv;

int max\_buffer = 50;

deque<int> buffer;

void producer(int val)

{

while(val>0)

{

std::unique\_lock<mutex> lock(m);

cv.wait(lock, []() {return buffer.size()<max\_buffer;});

buffer.push\_back(val);

cout<<"produced"<<val<<endl;

val--;

lock.unlock();

cv.notify\_one();

}

}

void consumer()

{

while(true)

{

std::unique\_lock<mutex> lock(m);

cv.wait(lock, []() {return buffer.size()>0;});

int val = buffer.back();

cout<<"consumed"<<val<<endl;

buffer.pop\_back();

lock.unlock();

cv.notify\_one();

}

}

int main()

{

thread t1(producer, 10);

thread t2(consumer);

t1.join();

t2.join();

return 0;

}

It is random that o/p depends on cpu which thread is able to acquire the lock agaoin and again.

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 example from class:

#include <iostream>

#include <thread>

#include <mutex>

class Counter {

private:

int count;

std::mutex mutex;

public:

Counter() : count(0) {}

void increment() {

std::lock\_guard<std::mutex> lock(mutex);

count++;

}

int getCount() {

return count;

}

};

int main() {

Counter counter;

// Create two threads that increment the counter

std::thread t1([&counter]() {

for (int i = 0; i < 100000; i++) {

counter.increment();

}

});

std::thread t2([&counter]() {

for (int i = 0; i < 100000; i++) {

counter.increment();

}

});

// Wait for both threads to finish

t1.join();

t2.join();

// Print the final counter value

std::cout << "Final Counter Value: " << counter.getCount() << std::endl;

return 0;

}

In C++, it is common to pass **this** as a parameter to a constructor when you want to provide the constructor access to the instance of the class that is being created. This can be useful in various situations:

1. **Member Initialization**: You might need to initialize member variables of the class based on some properties or methods of the object being constructed. By passing **this** as a parameter, you give the constructor access to the object's properties and methods.
2. **Chaining Constructors**: In some cases, you might have multiple constructors in a class, and one constructor delegates part of its work to another constructor. In such cases, you can pass **this** to the other constructor to avoid duplicating code.
3. **Passing the Object to Helper Functions**: If you have helper functions or methods that need access to the object being constructed, passing **this** allows those functions to work on the object's properties.

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

using namespace std;

mutex mtx;

condition\_variable cv;

bool isOdd = false;

void odd()

{

for(int i=1; i<=9; i+=2)

{

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [] { return !isOdd; });

cout << i << " ";

isOdd = true;

lock.unlock();

cv.notify\_all();

}

}

void even()

{

for(int i=2; i<=8; i+=2)

{

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [] { return isOdd; });

cout << i << " ";

isOdd = false;

lock.unlock();

cv.notify\_all();

}

}

int main()

{

thread t1(even);

thread t2(odd);

t1.join();

t2.join();

cout << endl;

}

* **Async** is multi-thread, which means operations or programs can run in parallel.
* **Sync** is a single-thread, so only one operation or program will run at a time.
* **Async** is non-blocking, which means it will send multiple requests to a server.
* **Sync** is blocking — it will only send the server one request at a time and wait for that request to be answered by the server.
* **Async** increases throughput because multiple operations can run at the same time.
* **Sync** is slower and more methodical.

Different ways to protect critical section in c++:

1)mutex

2)gurad\_lock()

3)unique\_lock()

4)condition\_variable

5)atomic\_lock

**Atomic Types**: C++11 introduced atomic types like **std::atomic** that provide thread-safe operations for variables. You can use atomic types for static variables that require simple operations like increments or assignments.

In C++, **std::atomic** is a template class provided by the Standard Library (introduced in C++11 and later) that enables atomic operations on variables. It is primarily used for concurrent programming to ensure that certain operations on shared data are performed atomically, meaning they are guaranteed to execute as a single, uninterruptible operation without interference from other threads. This helps in preventing data races and ensuring thread safety in multi-threaded applications.

1. **Atomic Types**: **std::atomic** can be used with various fundamental data types (e.g., **int**, **bool**, **float**, **double**) and some user-defined types that meet certain requirements. For example, you can create **std::atomic<int>** for atomic operations on integers.
2. **Atomic Operations**: **std::atomic** provides atomic versions of common operations like load, store, exchange, compare-and-swap, and various arithmetic operations (addition, subtraction, etc.). These operations are designed to be thread-safe, and their usage ensures that concurrent threads can safely access and modify the shared data.

std::atomic<int> atomicInt(0);

atomicInt.store(42); // Atomic store

int value = atomicInt.load(); // Atomic load

atomicInt.fetch\_add(1); // Atomic addition

Memory ordering constraints in C++ atomic operations (std::memory\_order) are used to specify how memory operations are ordered and visible to different threads in a multi-threaded program. These constraints provide control over the visibility and synchronization of memory operations, ensuring that the behavior of a multi-threaded program is well-defined and predictable

#include <iostream>

#include <atomic>

#include <thread>

std::atomic<int> counter(0);

void incrementCounter() {

for (int i = 0; i < 100000; ++i) {

counter.fetch\_add(1, std::memory\_order\_relaxed);

}

}

int main() {

std::thread thread1(incrementCounter);

std::thread thread2(incrementCounter);

thread1.join();

thread2.join();

std::cout << "Counter: " << counter.load() << std::endl;

return 0;

}

7) **Read-Write Locks (std::shared\_mutex)**:

* For scenarios where multiple threads may read data simultaneously but only one thread can write data, you can use **std::shared\_mutex** (C++17 and later).
* Multiple threads can acquire shared locks for reading, but a thread with an exclusive lock is required for writing.
* Example:

#include <iostream>

#include <shared\_mutex>

#include <thread>

std::shared\_mutex rwMutex;

void readOperation() {

std::shared\_lock<std::shared\_mutex> lock(rwMutex);

// Read data

}

void writeOperation() {

std::unique\_lock<std::shared\_mutex> lock(rwMutex);

// Write data

}