# MQF 633: C++ For Financial Engineering

# Lecture 8 Async, Multi-threading and More

### Part I: Asynchronous Programming

#### Introduction

- from Wikipedia

**Asynchrony**, in computer programming, refers to the **occurrence of events independent of the main program flow** and ways to deal with such events. These may be "outside" events such as the arrival of signals, or actions instigated by a program that take place concurrently with program execution, without the program hanging to wait for results. Asynchronous input/output is an example of the latter case of asynchrony, and lets programs issue commands to storage or network devices that service these requests while the processor continues executing the program. **Doing so provides a degree of parallelism**.[

A common way for dealing with asynchrony in a programming interface is to provide subroutines that return a **future or promise** that represents the ongoing operation, and a **synchronizing operation that blocks until the future or promise is completed**.

Examples of asynchrony include the following:

* Asynchronous procedure call, a method to run a procedure concurrently, a lightweight alternative to threads.
* Ajax is a set of client-side web technologies used by the client to create asynchronous I/O web applications.
* Asynchronous method dispatch (AMD), a data communication method used when there is a need for the server side to handle a large number of long lasting client requests.Using synchronous method dispatch (SMD), this scenario may turn the server into an unavailable busy state resulting in a connection failure response caused by a network connection request timeout. The servicing of a client request is immediately dispatched to an available thread from a pool of threads and the client is put in a blocking state. Upon the completion of the task, the server is notified by a callback. The server unblocks the client and transmits the response back to the client. In case of thread starvation, clients are blocked waiting for threads to become available.

#### Some basic concept

Thread vs Process

**Process**: Processes are basically the programs that are dispatched from the ready state and are scheduled in the CPU for execution. PCB (Process Control Block) holds the concept of process. A process can create other processes which are known as Child Processes. The process takes more time to terminate and it is **isolated** means it does not share the memory with any other process. The process can have the following states **new, ready, running, waiting, terminated, and suspended.**

**Thread**: **Thread is the segment of a process** which means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked. The thread takes less time to terminate as compared to the process but unlike the process, **threads do not isolate**.

Thread Safety

- from Wikipedia

In multi-threaded computer programming, a function is thread-safe when it can be invoked or accessed concurrently by multiple threads without causing unexpected behavior, race conditions, or data corruption. As in the multi-threaded context where a program executes several threads simultaneously in a shared address space and each of those threads has access to all every other thread's memory, thread-safe functions need to ensures all those threads behave properly and fulfill their design specifications without unintended interaction.

The most commonly use thread-safety terminology are:

1. **Not thread safe**: Data structures should not be accessed simultaneously by different threads.
2. **Thread safe, serialization**: Use a single mutex for all resources to guarantee the thread to be free of race conditions when those resources are accessed by multiple threads simultaneously.
3. **Thread safe, MT-saf**e: Use a mutex for every single resource to guarantee the thread to be free of race conditions when those resources are accessed by multiple threads simultaneously.

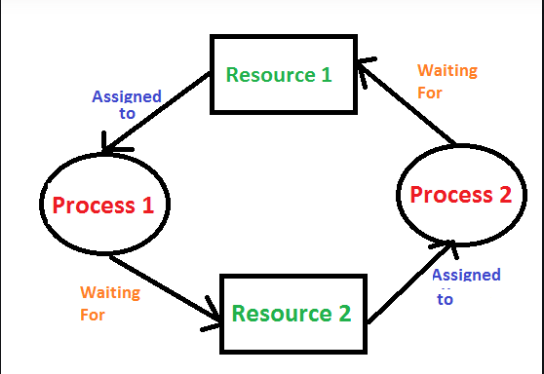
Thread safety guarantees usually also include design steps to prevent or limit the risk of different forms of **deadlocks**, as well as optimizations to maximize concurrent performance. However, deadlock-free guarantees cannot always be given, since deadlocks can be caused by callbacks and violation of architectural layering independent of the library itself.

Dead lock

A process in operating system uses resources in the following way.

* Requests a resource
* Use the resource
* Releases the resource

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1. 

Racing Condition

A race condition occurs when **two or more threads can access shared data and they try to change it at the same time**. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

**if (x == 5) // The "Check" {**

**y = x \* 2; // The "Act"**

**// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above, y will not be equal to 10.**

**}**

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing. In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time.

**// Obtain lock for x**

**if (x == 5) {**

**y = x \* 2; // Now, nothing can change x until the lock is released. Therefore y = 10**

**}**

**// release lock for x**

In later section, we will show example how to add or release lock for above code in a mult-threading program.

Lvalue and Rvalue

“L-value” refers to a memory location that identifies an object. “R-value” refers to the data value that is stored at some address in memory. References in C++ are nothing but the alternative to the already existing variable. They are declared using the ‘&’ before the name of the variable.

In C++03, an expression is either an rvalue or an lvalue. In C++11, an expression can be an:

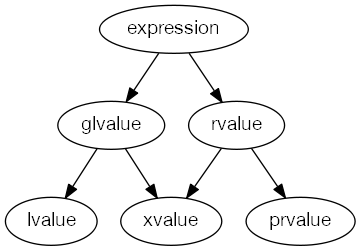
* rvalue

An rvalue (so-called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object or subobject thereof, or a value that is not associated with an object.

* lvalue

An lvalue (so-called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then \*E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue.]

* xvalue

An xvalue (an “eXpiring” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references. [Example: The result of calling a function whose return type is an rvalue reference is an xvalue.]

* glvalue

A glvalue (“generalized” lvalue) is an lvalue or an xvalue.

* prvalue

A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue]

Example: basic use of lvalue and rvalue

// C++ program to illustrate the lvalue and rvalue

#include <iostream>

using namespace std;

// Driver Code

int main()

{

int a = 10;

// Declaring lvalue reference (i.e variable a)

int& lref = a;

// Declaring rvalue reference

int&& rref = 20;

// Print the values

cout << "lref = " << lref << endl;

cout << "rref = " << rref << endl;

// Value of both a and lref is changed

lref = 30;

// Value of rref is changed

rref = 40;

cout << "lref = " << lref << endl;

cout << "rref = " << rref << endl;

// This line will generate an error as l-value cannot be assigned to the r-value references

// int &&ref = a;

return 0;

}

Example 2: use of the lvalue references:

1. lvalue references can be used to alias an existing object.
2. They can also be used to implement pass-by-reference semantics.

// C++ program to illustrate lvalue

#include <iostream>

using namespace std;

// Creating the references of the parameter passed to the function

void swap(int& x, int& y)

{

int temp = x;

x = y;

y = temp;

}

// Driver Code

int main()

{

// Given values

int a{ 10 }, b{ 20 };

cout << "a = " << a << " b = " << b << endl;

// Call by Reference

swap(a, b);

// Print the value

cout << "a = " << a << " b = " << b << endl;

return 0;

}

Uses of rvalue references:

1. They are used in working with the move constructor and move assignment.
2. Cannot bind non-const lvalue reference of type ‘int&‘ to an rvalue of type ‘int’.
3. Cannot bind rvalue references of type ‘int&&‘ to lvalue of type ‘int’.

// C++ program to illustrate rvalue

#include <iostream>

using namespace std;

// lvalue reference to the lvalue passed as the parameter

void printReferenceValue(int& x)

{

cout << x << endl;

}

// rvalue reference to the rvalue passed as the parameter

void printReferenceValue2(int&& x)

{

cout << x << endl;

}

int main()

{

// Given value

int a{ 10 };

// Function call is made lvalue & can be assigned to lvalue reference

printReferenceValue(a);

// Works fine as the function is called with rvalue

printReferenceValue2(100);

return 0;

}

#### Part II: Async in C++

As the name indicates, C++ **std::async** is a **function template**, which takes functions or function objects as arguments (basically called callbacks) and runs them asynchronously. **It returns the std:: the future object** which is used to keep the result of the above function. **The result is stored in the shared state.** In order to exact the value from it, its member future:: get needs to be called by the programmer.

In C++, async functions are used in 2 ways, i.e. **with or without specifying the policies** in the function arguments. When specifying the launch policy, the first argument is the policy itself which defines the asynchronous behavior of the function.

Syntax

Below given is the basic syntax of how the async function is used in C++ programs:

1. Without specifying the policy

**template <class function (Fn), class... Args>**

**std::future<typename std::result\_of<Fn(Args...)>::type>**

**async (Function && fn, Args&&... args);**

In the above syntax, the launch policy is not specified in the function arguments. Launch policy is automatically selected, which is launch:: async | launch:: deferred.

1. Specifying the policy

**template <class function(Fn), class... Args>**

**std::future<typename std::result\_of<Fn(Args...)>::type>**

**async (launch policy, Function && fn, Args&&... args);**

In the above syntax, launch policy is mentioned in the function arguments in order to specify before in which policy a function should execute.

**Fn**: It is the callable object or the function object. The return value of this function ‘fn’ is stored in the shared state, which is accessed by the object ‘future’. In case of exceptions also, the value is set in the shared state, which the future object can also access.

**args**: It is the arguments or the parameters which are passed in the async function ‘fn’.

One important point to note in the async function is that both the function ‘Fn’ and argument ‘args’ should be move constructible, and the function uses the decay copies of both the Fn and args.

**Policy**: Policy in C++ async plays an important role in the one which defines the asynchronous behaviour followed using the function async. There are basically 3 ways in which create async using different policies:

|  |  |  |
| --- | --- | --- |
| S.No | Policy name | Behavior |
| 1. | launch::async | This launch policy assures the async behavior of the function, which means that the callable function will be executed in a new thread. It follows the easy evaluation policy in which calculation will be evaluated immediately in the work package in the new thread. In case of exception, it is stored in the shared state, which is accessed by std::future. |
| 2. | launch::deferred | In this launch policy, a callable function is not executed in the new thread; instead, it follows the non-async behavior. It follows the lazy evaluation policy in which the call to the function is deferred (postponed) till the previous thread calls the get on the future object, which makes the shared state again accessible. The function then will no longer be deferred. In this case, the function will run its work package in the same thread. In case of exceptions, it is placed in the shared state of the future; then, it is made ready for the required function. |
| 3. | launch::async | launch::deferred | This is an automatic launch policy. In this policy, the behavior is not defined. The system can choose either asynchronous or deferred depending on the implementation according to the optimized availability of the system. Programmers have no control over it on anything. |

**Return Value:** The return value of async is the std:: future which is the shared state which is created by the function call of std::async. We can get the values from it using member future::get returned by the function.

// Example of checking the number is even or not using async

#include <iostream> // library used for std::cout

#include <future> // library used for std::async and std::futur

// function for checking the number is even or not

bool check\_even (int num) {

std::cout << "Hello I am inside the function!! \n";

//checking the divisibility of number by 2 and returning a bool value

if(num%2==0) {

return true;

}

return false;

}

int main () {

// calling the above function check\_even asynchronously and storing the result in future object

std::future<bool> ft = std::async (check\_even,10);

std::cout << "Checking whether the number 10 is even or not.\n";

// retrieving the exact value from future object and waiting for check\_even to return

bool rs = ft.get();

if (rs) {

std::cout << "Number mentioned by you is even \n";

}

else {

std::cout << "Sorry the number is odd \n";

}

return 0;

}

Quiz for part I & II

1. What does std::async provide in C++?

A) A mechanism for creating threads.

B) Asynchronous execution of a function.

C) Synchronous execution of a function.

D) A tool for memory management.

1. How is std::async different from std::thread?

A) std::async can only execute functions asynchronously.

B) std::async returns a future to obtain the result of the function execution.

C) std::async cannot execute member functions of classes.

D) std::async is more lightweight than std::thread.

1. Which of the following is true about std::async?

A) The function is executed immediately upon calling std::async.

B) std::async returns a std::thread object.

C) The default launch policy is std::launch::async.

D) std::async blocks the calling thread until the function completes.

1. How can you specify the launch policy for std::async?

A) By passing a launch policy parameter to std::async.

B) By setting a global launch policy variable.

C) Launch policy cannot be specified for std::async.

D) Launch policy is always std::launch::async.

1. What happens if an exception is thrown in the function executed by std::async?

A) The exception is propagated to the caller when calling .get() on the future.

B) The exception is silently ignored.

C) The program crashes.

D) The exception is handled internally by std::async.

1. Which of the following launch policies for std::async ensures deferred execution?

A) std::launch::async

B) std::launch::deferred

C) std::launch::sync

D) std::launch::parallel

1. What does deferred execution mean in the context of std::async?

A) The function is executed immediately upon calling std::async.

B) The function is executed asynchronously.

C) The function is executed synchronously.

D) The function is executed only when .get() is called on the future.

1. Which of the following is true about the return value of std::async?

A) It always returns a value.

B) It returns a std::future to obtain the result of the function execution.

C) It returns a std::shared\_ptr to the result of the function execution.

D) It returns void.

### Part III: C++ thread and multi-threading

#### Std::thread

Multithreading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. Each part of such a program is called a thread. So, threads are lightweight processes within a process. Multithreading support was introduced in C++11. Prior to C++11, we had to use POSIX threads or <pthreads> library. While this library did the job the lack of any standard language-provided feature set caused serious portability issues. C++ 11 did away with all that and gave us std::thread. The thread classes and related functions are defined in the **<thread>** header file.

**Syntax: std::thread thread\_object (callable);**

**std::thread** is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e, a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable. A callable can be any of the five:

* A Function Pointer
* A Lambda Expression
* A Function Object
* Non-Static Member Function
* Static Member Function

Example of launching thread on function pointer

**void foo(param) {**

**Statements;**

**}**

**// The parameters to the function are put after the comma**

**std::thread thread\_obj(foo, params);**

Example of launching thread on lambda expression

**// Define a lambda expression**

**auto f = [] (params) {**

**Statements;**

**};**

**// Pass f and its parameters to thread object constructor as**

**std::thread thread\_object(f, params);**

Example of launching thread on function object

**// Define the class of function object**

**class fn\_object\_class {**

**// Overload () operator**

**void operator() (params) {**

**Statements;**

**}**

**}**

**// Create thread object**

**std::thread thread\_object(fn\_object\_class(), params)**

Example of launching thread on function object

**// defining class**

**class Base {**

**public:**

**// non-static member function**

**void foo(param) { Statements; }**

**}**

**// object of Base Class**

**Base b;**

**// first parameter is the reference to the function and second paramter is reference of the object at last we have arguments**

**std::thread thread\_obj(&Base::foo, &b, params);**

Wait for thread to finish

Once a thread has started we may need to wait for the thread to finish before we can take some action. For instance, if we allocate the task of initializing the GUI of an application to a thread, we need to wait for the thread to finish to ensure that the GUI has loaded properly. To wait for a thread, use the std::thread::join() function. This function makes the current thread wait until the thread identified by \*this has finished executing.

**int main()**

**{**

**// Start thread t1**

**std::thread t1(callable);**

**// Wait for t1 to finish**

**t1.join();**

**}**

**// t1 has finished do other stuff;**

**return 0;**

**}**

// C++ program to demonstrate multithreading using three different callables.

#include <iostream>

#include <thread>

using namespace std;

// A dummy function

void foo(int Z) {

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

cout << "Thread using function pointer as callable\n";

}

}

// A callable object

class thread\_obj {

public:

void operator()(int x) {

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

cout << "Thread using function object as callable\n";

}

};

// class definition

class Base {

public:

// non-static member function

void foo(){

cout << "Thread using non-static member function "

"as callable"

<< endl;

}

// static member function

static void foo1() {

cout << "Thread using static member function as callable" << endl;

}

};

// Driver code

int main()

{

cout << "Threads 1 and 2 and 3 operating independently" << endl;

// This thread is launched by using function pointer as callable

thread th1(foo, 3);

// This thread is launched by using function object as callable

thread th2(thread\_obj(), 3);

// Define a Lambda Expression

auto f = [](int x) {

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

cout << "Thread using lambda expression as callable\n";

};

// This thread is launched by using lambda expression as callable

thread th3(f, 3);

// object of Base Class

Base b;

thread th4(&Base::foo, &b);

thread th5(&Base::foo1);

// Wait for thread t1 to finish

th1.join();

// Wait for thread t2 to finish

th2.join();

// Wait for thread t3 to finish

th3.join();

// Wait for thread t4 to finish

th4.join();

// Wait for thread t5 to finish

th5.join();

return 0;

}

C++ std::mutex

A C++ mutex, short for "mutual exclusion," is a synchronization primitive used in multithreading to protect shared resources from being accessed concurrently by multiple threads. Mutexes ensure that only one thread can access the protected resource at a time, thus preventing race conditions and ensuring data consistency.

In C++, mutexes are typically implemented using the std::mutex class provided by the C++ standard library (<mutex> header). Here's a brief overview of how mutexes work in C++:

* **Locking**: A thread that wants to access a shared resource locks the mutex associated with that resource using the lock() method of the std::mutex object. If the mutex is already locked by another thread, the calling thread will be blocked until the mutex becomes available.
* **Unlocking**: After completing its work with the shared resource, the thread unlocks the mutex using the unlock() method. This allows other threads waiting to access the resource to proceed.
* **Scoped Locking**: To simplify mutex usage and ensure proper unlocking even in case of exceptions, C++ provides the std::lock\_guard class. std::lock\_guard is a RAII (Resource Acquisition Is Initialization) wrapper around a mutex. It locks the mutex upon construction and unlocks it automatically when it goes out of scope.

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mtx; // Declare a mutex

void sharedResourceAccess() {

std::lock\_guard<std::mutex> lock(mtx); // Lock the mutex

// Access the shared resource here

std::cout << "Thread " << std::this\_thread::get\_id() << " is accessing the shared resource.\n";

// The mutex will be automatically unlocked when 'lock' goes out of scope

}

int main() {

std::thread t1(sharedResourceAccess);

std::thread t2(sharedResourceAccess);

t1.join();

t2.join();

return 0;

}

C++ Thread Pool

The Thread Pool in C++ is used to manage and efficiently resort to a group (or pool) of threads. Instead of creating threads again and again for each task and then later destroying them, what a thread pool does is it maintains a set of pre-created threads now these threads can be reused again to do many tasks concurrently. By using this approach we can minimize the overhead that costs us due to the creation and destruction of threads. This makes our application more efficient. There is no in-built library in C++ that provides the thread pool, so we need to create the thread pool manually according to our needs.

**What is Thread Pool?**

A group of worker threads that are established at the program start and stored in a pool to be used at a later time are called thread pools. The Thread Pool effectively maintains and allocates existing threads to do several tasks concurrently, saving time compared to starting a new thread for each activity.

**Need of Thread Pool in C++**

* When several activities must be completed simultaneously, like in server applications, parallel processing, and parallelizing loops, thread pools are frequently utilized.
* Thread Pools enhance overall performance by lowering the overhead of thread generation and destruction through thread reuse.

**Key components of thread pool implementation**

* there is an array of threads
* there is a FIFO queue of tasks (a Task should be a wrapper for a Callable object).
* when a client creates a task he enqueu() it into the task queue
* all threads are started up front in the pool and they never quit running; if there's no task in the queue they wait
* each incoming task notifies() a single thread to handle it
* if a Task returns a result the client can send a query for its value later
* the thread pool has a switch On/Off flag
* Thread synchronization: a condition variable along with a mutex variable perform thread synchronization
* mutex is used to synchronize access to shared data (the containers mostly)
* condition notifies threads upon task arrival and puts them to sleep when there are no tasks
* Try to reduce the usage of standard raw pointers as much as possible in your programs.

// C++ Program to demonstrate thread pooling

#include <condition\_variable>

#include <functional>

#include <iostream>

#include <mutex>

#include <queue>

#include <thread>

using namespace std;

// Class that represents a simple thread pool

class ThreadPool {

public:

// Constructor to creates a thread pool with given number of threads

ThreadPool(size\_t num\_threads= thread::hardware\_concurrency()) {

// Creating worker threads

for (size\_t i = 0; i < num\_threads; ++i) {

threads\_.emplace\_back([this] {

while (true) {

function<void()> task;

// The reason for putting the below here is to unlock the queue before executing the task so that other

// threads can perform enqueue tasks

{

// Locking the queue so that data can be shared safely

unique\_lock<mutex> lock(queue\_mutex\_);

// Waiting until there is a task to execute or the pool is stopped

cv\_.wait(lock, [this] {

return !tasks\_.empty() || stop\_;

});

// exit the thread in case the pool is stopped and there are no tasks

if (stop\_ && tasks\_.empty()) {

return;

}

// Get the next task from the queue

task = move(tasks\_.front());

tasks\_.pop();

}

task();

}

});

}

}

Static casting on user defined types

In C++, static\_cast is a type of casting operator used for explicit type conversions. It allows you to convert one data type into another in a type-safe manner. static\_cast is generally used for conversions that are well-defined and safe at compile time, such as converting between related types, like integers and floating-point numbers, or performing upcasts in inheritance hierarchies.

Syntax: **result\_type result = static\_cast<result\_type>(expression);**

Where:

* result\_type is the type you want to convert to.
* expression is the value or expression you want to convert.

// Destructor to stop the thread pool

~ThreadPool() {

{

// Lock the queue to update the stop flag safely

unique\_lock<mutex> lock(queue\_mutex\_);

stop\_ = true;

}

// Notify all threads

cv\_.notify\_all();

// Joining all worker threads to ensure they have completed their tasks

for (auto& thread : threads\_) {

thread.join();

}

}

// Enqueue task for execution by the thread pool

void enqueue(function<void()> task) {

{

unique\_lock<std::mutex> lock(queue\_mutex\_);

tasks\_.emplace(move(task));

}

cv\_.notify\_one();

}

private:

// Vector to store worker threads

vector<thread> threads\_;

// Queue of tasks

queue<function<void()> > tasks\_;

// Mutex to synchronize access to shared data

mutex queue\_mutex\_;

// Condition variable to signal changes in the state of the tasks queue

condition\_variable cv\_;

// Flag to indicate whether the thread pool should stop or not

bool stop\_ = false;

};

**Example**

**int main()**

**{**

**// Create a thread pool with 4 threads**

**ThreadPool pool(4);**

**// Enqueue tasks for execution**

**for (int i = 0; i < 5; ++i) {**

**pool.enqueue([i] {**

**cout << "Task " << i << " is running on thread " << this\_thread::get\_id() << endl;**

**// Simulate some work**

**this\_thread::sleep\_for(**

**chrono::milliseconds(100));**

**});**

**}**

**return 0;**

**}**

Output

Task 0 is running on thread 140178994148928  
Task 1 is running on thread 140178985756224  
Task 2 is running on thread 140179010934336  
Task 3 is running on thread 140179002541632  
Task 4 is running on thread 140178994148928

**Explanation**

1. In the above code, we have used the following C++ features for the implementation of the thread pool:
2. A vector of worker threads, a task queue, a mutex for synchronization, a condition variable for signaling, and a boolean flag to indicate whether the pool should stop are all managed by the ThreadPool class.
3. The worker threads are initialized by the constructor, who then puts them in an endless loop while they wait for jobs to be enqueued. We use a wrapper class std::function over the given tasks.
4. A job is added to the queue and one of the worker threads is notified to begin executing it using the enqueue method.
5. To guarantee a clean shutdown, the destructor joins the worker threads, sets the stop flag, and informs all threads.
6. A ThreadPool with four threads is formed in the main function. Ten jobs are queued up, each of which prints a message including the task number and the thread ID that is currently carrying it out.
7. It should be noted that in a real-world situation, you would usually connect the threads or use some other kind of synchronization to make sure that all jobs are finished before the program ends.

**Advantages of Thread Pooling in C++**

The following are some main advantages of thread pooling in C++:

* Resource Management: Thread pools effectively manage resources by preventing resource depletion by restricting the number of threads that are executing concurrently.
* Enhanced Performance: By lowering the cost involved in establishing and terminating threads, reusing threads enhances performance.
* Scalability: Thread Pools are scalable in a variety of settings because they may dynamically modify the number of worker threads according to the capabilities of the system.

**Disadvantages of Thread Pooling in C++**

Thread pooling also have some limitations which are mentioned below:

* Thread pool adds complexity to the code hence managing threads and task queues might impact the performance of other lightweight tasks.
* The thread pool works on the assumption that each task is independent. So, Handling the dependencies between tasks is challenging when one task depends on the result of another task.
* The behavior of thread pools is platform-dependent. Hence, behavior may vary in different operating systems and C++ compilers.

Dead lock program example

**#include <iostream>**

**#include <boost/thread/thread.hpp>**

**#include <boost/thread/mutex.hpp>**

**boost::mutex mutex1, mutex2;**

**void ThreadA()**

**{**

**// Creates deadlock problem**

**mutex2.lock();**

**std::cout << "Thread A" << std::endl;**

**mutex1.lock();**

**mutex2.unlock();**

**mutex1.unlock();**

**}**

**void ThreadB()**

**{**

**// Creates deadlock problem**

**mutex1.lock();**

**std::cout << "Thread B" << std::endl;**

**mutex2.lock();**

**mutex1.unlock();**

**mutex2.unlock();**

**}**

**void ExecuteThreads()**

**{**

**boost::thread t1( ThreadA );**

**boost::thread t2( ThreadB );**

**t1.join();**

**t2.join();**

**std::cout << "Finished" << std::endl;**

**}**

**int main()**

**{**

**ExecuteThreads();**

**return 0;**

**}**

When running this notice that the program hangs and is unable to proceed beyond the second mutex lock acquisition.



This is because If thread A is executing and isn’t holding mutex lock 1 yet and thread B acquires mutex lock 2, neither of the threads can continue past the second lock acquisition.

Solve this issue we need to

**#include <iostream>**

**#include <boost/thread/thread.hpp>**

**#include <boost/thread/mutex.hpp>**

**boost::mutex mutex1, mutex2;**

**void ThreadA()**

**{**

**// Creates deadlock problem**

**Mutex1.lock();**

**std::cout << "Thread A" << std::endl;**

**mutex2.lock();**

**mutex2.unlock();**

**mutex1.unlock();**

**}**

**void ThreadB()**

**{**

**// Creates deadlock problem**

**mutex1.lock();**

**std::cout << "Thread B" << std::endl;**

**mutex2.lock();**

**mutex2.unlock();**

**mutex1.unlock();**

**}**

**void ExecuteThreads()**

**{**

**boost::thread t1( ThreadA );**

**boost::thread t2( ThreadB );**

**t1.join();**

**t2.join();**

**std::cout << "Finished" << std::endl;**

**}**

**int main()**

**{**

**ExecuteThreads();**

**return 0;**

**}**



Quiz for part II

1. What does multithreading refer to in C++?

A) Execution of multiple processes simultaneously

B) Execution of multiple threads within a single process concurrently

C) Execution of multiple functions sequentially

D) Execution of multiple classes simultaneously

1. How can you create a thread in C++ using the standard library?

A) Using std::create\_thread()

B) Using std::thread()

C) Using std::start\_thread()

D) Using std::run\_thread()

1. What is the purpose of synchronization in multithreading?

A) To increase the number of threads.

B) To decrease the number of threads.

C) To coordinate access to shared resources among multiple threads.

D) To terminate threads gracefully.

1. What is a mutex in C++ multithreading?

A) A tool for memory management.

B) A tool for thread scheduling.

C) A mutual exclusion primitive used to protect shared resources.

D) A function to create threads.

1. What is a race condition in multithreading?

A) A situation where two threads are waiting for each other to complete.

B) A situation where the execution of threads is sequential.

C) A situation where multiple threads access shared data concurrently, leading to unpredictable results.

D) A situation where a thread is terminated unexpectedly.

1. How can a deadlock be avoided in multithreading?

A) By using sleep statements.

B) By avoiding the use of synchronization primitives.

C) By ensuring that locks are always acquired in the same order.

D) By terminating all threads when a deadlock occurs.

1. What is a condition variable in C++ multithreading?

A) A variable used to store thread IDs.

B) A variable used to check the condition of a thread.

C) A synchronization primitive used to signal changes in shared data.

D) A variable used to store thread priorities.

1. What is a benefit of multithreading in C++?

A) Increased memory usage.

B) Reduced CPU utilization.

C) Improved responsiveness and performance in applications.

D) Simplified debugging process.