10

Multi-threading in Games Development

In chapter, the following recipes will be covered:

* Concurrency in games: Creating a thread
* Joining and Detaching a thread
* Passing arguments to thread
* Avoiding deadlocks
* Data race and locks
* Writing a thread safe class

# Introduction

To first understand multi-threading, let us first understand the meaning of threads. A Thread is a concurrent unit of execution. It has its own call stack for methods being invoked, their arguments and local variables. Each application has at least one thread running when it is started, the main thread. When we talk about multi-threading, it means one process has many threads running independently and concurrently, however having a shared memory. Often multi-threading is confused as multi-processing. A multi-processor has multiple processes running each having its own thread.

Although multi-threaded applications may be complex to write, they are light weight. However multi-threaded architecture is not well suited for a distributed application. In games we may have one or more threads running. The golden question is why and when should we use multi-threading. Although these are quite subjective, you would use multi-threading if you want multiple tasks to happen concurrently. So if you do not want your physics code in the game or the audio code in the game to wait for the main loop to finish processing, you would multi-thread the physics and the audio loop.

# Concurrency in games: Creating a thread

1. The first step to start writing multi-threaded code is to spawn a thread. At this point we must note that the application is already running an active thread or the main thread. So when we spawn a thread, there will be 2 active threads in the application.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio. No other prerequisites are required.

## How to do it...

In this recipe we will see how easy it is to spawn a thread.

* Add a source file called Source.cpp

int ThreadOne()

{

std::cout << "I am thread 1" << std::endl;

return 0;

}

int main()

{

std::thread T1(ThreadOne);

if (T1.joinable()) // Check if can be joined to the main thread

T1.join(); // Main thread waits for this to finish

\_getch();

return 0;

}

## How it works...

The first step is to include the header file, thread.h. This gives us access to all the inbuilt libraries that we may need to create our multi-threaded application. Next step is to create the task or the function that we need to thread. In this example we have created a function called ThreadOne. This function represents any function that we can use to multi-thread. This could be a physics function or audio or anything that we may desire. For simplicity we have used a function that prints a message. The next step is to spawn a thread. We simply need to write the keyword thread, assign a name to the thread(T1) and then write the function/task that we want to thread. In this case it is ThreadOne.

This spawns a thread and will not execute independently to the main thread.

# Joining and Detaching

After a thread is spawned, it starts executing as a new task apart from the main thread. However, there may be situations in which we want the task to rejoin with the main thread. This is possible. Also we may want that the thread always stays apart from the main thread. That is also possible. There are few precautions that we must take when attaching and detaching to the main thread

## Getting ready

1. You need to have a working Windows machine and Visual Studio.

## How to do it...

In this recipe we will see how easy it is to join and detach threads.Add a source file called Source.cpp

Code Snippet

int ThreadOne()

{

std::cout << "I am thread 1" << std::endl;

return 0;

}

int ThreadTwo()

{

std::cout << "I am thread 2" << std::endl;

return 0;

}

int main()

{

std::thread T1(ThreadOne);

std::thread T2(ThreadTwo);

if (T1.joinable()) // Check if can be joined to the main thread

T1.join(); // Main thread waits for this to finish

T2.detach(); //Detached from main thread

\_getch();

return 0;

}

## How it works...

In the above example, at first two threads are spawned. The two threads are T1 and T2. At the point when the threads are spawned, they act independently and concurrently. However when there is a need for any thread to be joined back to the main thread, we can do that as well. First we need to check if the thread can be joined to the main thread. We accomplish by the joinable function. If that function returns true, the thread can join back to the main thread. We can join to the main thread by the join function. If we directly join without checking if the thread can join to the main thread, it may cause issues with the main thread failing to accept the thread. After the thread joins with the main thread, the main thread waits for that thread to finish.

If we want to detach from the main thread, we can use the detach function. However after we detach from the main thread, we are forever detached from the thread.

# Passing arguments to the thread

Like in functions, we may also want to sent parameters and arguments to the thread. As the threads as just tasks, and tasks are just a collection of functions, it is necessary to understand how to send arguments to the thread. If we can send arguments to the thread at runtime, then it can perform all the operations dynamically. Most cases we would thread the physics, AI or audio sections of the code. All these sections would require functions that takes in arguments.

## Getting ready

You need a Windows machine and a working copy of Visual Studio. No other pre-requisite is needed.

## How to do it...

In this recipe we will find out how easy it is to add a heuristic function to our game for path finding.

* Add a source file called Source.cpp

Code Snippet

class Wrapper

{

public:

void operator()(std::string& msg)

{

msg = " I am from T1";

std::cout << "T1 thread initiated" << msg << std::endl;

}

};

int main()

{

std::string s = "This is a message";

std::cout << std::this\_thread::get\_id() << std::endl;

std::thread T1((Wrapper()), std::move(s));

std::cout << T1.get\_id() << std::endl;

std::thread T2 = std::move(T1);

T2.join();

\_getch();

}

## How it works...

The best way to do this is to write a wrapper class and overload the () operator. After we overload the () operator we can now send arguments to the thread. For doing this, we create a string and store the string in a variable. Then we need to spawn a thread as usual, however, now instead of just passing in the function name, we pass in the classname and the string. In threads, we need to pass the arguments by reference. So we could use the ref function. However, a better way to do it is by using the move function in which we are noting the memory location itself and passing it to the argument. The operator function accepts the string and prints the message.

If we want to create a new thread and make it same as the first thread, we can again use the move function to do this. In addition to this we can get the thread id by using the get\_id function.

# Deadlocks

When two or more tasks are wanting to use the same resource, we have a race condition. Till one task finish using the resource, the other task cannot have access to it. This is known as a deadlock and we must avoid deadlocks at all costs. For example, Resource Collision and resource Audion are used by process Locomotion and process Bullet

* Locomotion starts to use Collision.
* Locomotion and Bullet try to start using Audio
* Bullet 'wins' and gets Audio first
* Now Bullet needs to use Collision
* Collision is locked by Locomotion, which is waiting for Bullet

## Getting ready

For this recipe, you will need a Windows machine an installed copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to add source control.

#include <thread>

#include <string>

#include <iostream>

using namespace std;

void Physics()

{

for (int i = 0; i > -100; i--)

cout << "From Thread 1: " << i << endl;

}

int main()

{

std::thread t1(Physics);

for (int i = 0; i < 100; i++)

cout << "From main: " << i << endl;

t1.join();

int a;

cin >> a;

return 0;

1. }

## How it works...

In the above example, we have spawned a thread t1 which starts a function to print numbers from 0 to -100 decreasing by 1. There is also a main thread which starts printing numbers from 0 to 100 increasing by 1. Now again for simplicity in understanding, we have chosen these functions. Those could easily be replaced by an A\* algorithm and a search algorithm or anything we want.

If we look at the console output, we notice that it is quite messy. The reason for that is the object cout is being used by both main thread and t1. Hence there is a data race condition that is taking place. Whoever is winning at each turn, is getting to display the number. We must avoid this kind of programming structure at all costs. More often than not, it will cause deadlock and cause disruption.

# Data race and Mutex

Data race conditions are very common in multi-threaded applications. But we must avoid such a scenario so that deadlocks do not happen. Mutex help us to overcome deadlocks. A **mutex** is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a **mutex** is created with a unique name.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version of Visual Studio.

## How to do it...

In this recipe we will see how easy it is to understand data race and mutex

1. Add a source file called Source.cpp

Code Snippet

#include <thread>

#include <string>

#include <mutex>

#include <iostream>

using namespace std;

std::mutex MU;

void Locomotion(string msg, int id)

{

std::lock\_guard<std::mutex> guard(MU); //RAII

//MU.lock();

cout << msg << id << endl;

//MU.unlock();

}

void InterfaceFunction()

{

for (int i = 0; i > -100; i--)

Locomotion(string("From Thread 1: "), i);

}

int main()

{

std::thread FirstThread(InterfaceFunction);

for (int i = 0; i < 100; i++)

Locomotion(string("From Main: "), i);

FirstThread.join();

int a;

cin >> a;

return 0;

}

## How it works...

In this example, both the main thread and t1 wants to display some numbers. However as both of them want to use the cout object, hence it creates a data race situation. To avoid this, one approach is to use mutex locks. So before executing the print statement, we have a mutex.lock and after the print statement we have a mutex.unlock. This will work and prevent the data race condition as mutex will allow one thread to use the resource and make the other thread wait for it. However, this program is not yet thread safe. This is because, if the cout statement throws an error or exception, the mutex will never get unlocked and the other threads will always be in a wait state.

To prevent this, we will use the Resouce Acquisition is Initialisation technique(RAII) of C++. We add an inbuilt lock\_guard to the function. This code is exception-safe because C++ guarantees that all stack objects are destroyed at the end of the enclosing scope, known as stack unwinding. The destructors of both the *lock* and *file* objects are therefore guaranteed to be called when returning from the function, whether an exception has been thrown or not. Hence it will not stop other threads from waiting eternally if an exception has occurred. Inspite of doing this, this application is not thread safe. This is because the cout object is a global object. Hence other parts of the program can access it as well. Hence we need to encapsulate this even further. This we will see later.

# Thread safe class

When dealing with multiple threads, writing a thread safe class becomes an absolute must. If we do not write classes which are thread safe, there are many complications which may arise like deadlocks. We must also keep in mind that when we write the thread safe class, there is not potential danger for data race and mutex.

## Getting ready

1. For this recipe, you will need a Windows machine and an installed version Visual Studio.

## How to do it...

In this recipe we will see how easy it is to write a thread safe class in C++.

1. Add a source file called Source.cpp

#include <thread>

#include <string>

#include <mutex>

#include <iostream>

#include <fstream>

using namespace std;

class DebugLogger

{

std::mutex MU;

ofstream f;

public:

DebugLogger()

{

f.open("log.txt");

}

void ResourceSharingFunction(string id, int value)

{

std::lock\_guard<std::mutex> guard(MU); //RAII

f << "From" << id << ":" << value << endl;

}

};

void InterfaceFunction(DebugLogger& log)

{

for (int i = 0; i > -100; i--)

log.ResourceSharingFunction(string("Thread 1: "), i);

}

int main()

{

DebugLogger log;

std::thread FirstThread(InterfaceFunction,std::ref(log));

for (int i = 0; i < 100; i++)

log.ResourceSharingFunction(string("Main: "), i);

FirstThread.join();

int a;

cin >> a;

return 0;

1. }

## How it works...

In the previous recipe, we have seen how in spite of wring mutex and locks, our code is not thread safe. This is because we were using a global object cout. This could have been accessed from other parts of the code as well. Hence it was not thread safe. So we have avoided to do this by adding one more layer of abstraction and outputting the result to a log file.

We have created a class called Logfile. Inside the class we have created a lock guard and mutex. On top of that we have also create a stream object called f. Using that we output the contents to a text file. The threads which need access to this functionality will need to create an object of the LogFile and then use the function appropriately. We are using the lock guard in RAII system. Because of this layer of abstraction, there is no chance that the functionality can be used externally and it is quite safe.

However even in this program, we need to take certain precautions. The first precaution that we should take is that we should not return f from any function. Also we have to be careful that f should not be directed available from any other class or external functions.