MULTITHREADING IN C++

Multithreading is a technique where a program is divided into smaller units of execution called threads. Each thread runs independently but shares resources like memory, allowing tasks to be performed simultaneously. This helps improve performance by utilizing multiple CPU cores efficiently. Multithreading support was introduced in C++11 with the introduction of **<thread>** header file.

Importance of Multithreading

- Leverages multiple CPU cores to execute tasks in parallel, reducing overall execution time.
- Keeps applications responsive by running background operations without blocking the main thread. For example, in a word document, one thread does auto-formatting along with the main thread.
- Makes it easier to handle large workloads or multiple simultaneous operations, such as in servers or real-time systems.

Common Operations On Thread

The **<thread>** header in C++ provides a simple and powerful interface for managing threads. Below are some of the most common operations performed on threads.

Create a Thread

Syntax

```
thread thread_name(callable);
```

- **thread_name**: It is an object of thread class.
- **callable**: It is a callable object like function pointer, function object.

Implement and run the following in a new project, within a file titled **ExampleThread.cpp**

```
#include <iostream>
#include <thread>

using namespace std;

// Function to be run by the thread
void func() {
    cout << "Hello from the thread!" << endl;
}

int main() {

    // Create a thread that runs
    // the function func
    thread t(func);

    // Main thread waits for 't' to finish
    t.join();
    cout << "Main thread finished.";
    return 0;
}
```

In the above program (**ExampleThread.cpp**) we have created a thread **t** that prints "Hello from the thread!" and this thread is joined with the main thread so that the main thread waits for the completion of this thread and once the thread **t** is finished the main thread resumes its execution and prints " Main thread finished".

Joining a Thread

Before joining a thread it is preferred to check if the thread can be joined using the **joinable()** method. The joinable method checks whether the thread is in a valid state for those operations or not.

Syntax

```
thread_name.joinable()
```

The **joinable()** method returns true if the thread is joinable else returns false.

Joining two threads C++ blocks the current thread until the thread associated with the std::thread object finishes execution. To join two threads in C++ we can use join() function. Which is called inside the body of the thread to which the specified thread is to be joined.

Syntax

```
thread_name.join();
```

The thread.**join()** function throws **std::system_error** if the thread is not joinable.

Note: Joining two non-main threads is risky as it may lead to race condition or logic errors.

- A race condition occurs when two or more threads access shared resources at the same time, and at least one of them modifies the resource. Since the threads are competing to read and write the data, the final result depends on the order in which the threads execute, leading to unpredictable or incorrect results.

Detaching a thread

A joined thread can be detached from the calling thread using the **detach()** member function of the **std::thread** class. When a thread is detached, it runs independently in the background, and the other thread does not waits for it to finish.

Syntax

```
thread_name.detach()
```

Getting Thread ID

In C++ each thread has a unique ID which can be obtained by using the **get_id()** function.

Syntax

```
thread_name.get_id();
```

The **get_id()** function returns an object representing the thread's ID.

Implement and run the following in a new project, within a file titled **MultipleThreads.cpp**.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;

void task1() {
    cout << "Thread 1 is running. ID: " << this_thread::get_id() << "\n";
}

void task2() {
    cout << "Thread 2 is running. ID: " << this_thread::get_id() << "\n";
}

int main() {
    thread t1(task1);
    thread t2(task2);

    // Get thread IDs
    cout << "t1 ID: " << t1.get_id() << "\n";
    cout << "t2 ID: " << t2.get_id() << "\n";

    // Join t1 if joinable
    if (t1.joinable()) {
        t1.join();
        cout << "t1 joined\n";
    }

    // Detach t2
    if (t2.joinable()) {
        t2.detach();
        cout << "t2 detached\n";
    }

    cout << "Main thread sleeping for 1 second...\n";
    this_thread::sleep_for(chrono::seconds(1));
    cout << "Main thread awake.\n";

    return 0;
}
```

Callables in Multithreading

A callable (such as a function, lambda, or function object) is passed to a thread. The callable is executed in parallel by the thread when it starts. like, **thread t(func)**; creates a thread that runs the func function. We can also pass parameters along with callable, like this thread **t(func, param1, param2)**;

In C++, callable can be divided into 4 categories:

- Function
    - A function can be a callable object to pass to the thread constructor for initializing a thread.
- Lambda Expression
    - Thread objects can also use a lambda expression as a callable. Which can be passed directly inside the thread object.
- Function Object
    - Function Objects or Functors can also be used for a thread as callable. To make functors callable, we need to overload the operator parentheses operator ().
- Non-Static or static Member Function
    - Threads can also be used using the non-static or static member functions of a class. For non-static member functions, we need to create an object of a class but it is not necessary with static member functions.

Implement and run the following in a new project, within a file titled **FunctionPointer.cpp**.

```cpp
#include <iostream>
#include <thread>

using namespace std;

// Function to be run
// by the thread
void func(int n) {
    cout << n;
}

int main() {

    // Create a thread that runs
    // the function func
    thread t(func, 4);

    // Wait for thread to finish
    t.join();
    return 0;
}
```

Implement and run the following in a new project, within a file titled **LambdaExpression.cpp**.

```
#include <iostream>
#include <thread>

using namespace std;

int main() {
    int n = 3;

    // Create a thread that runs
    // a lambda expression
    thread t([](int n){
        cout << n;
    }, n);

    // Wait for the thread to complete
    t.join();
    return 0;
}
```

Function Objects

Implement and run the following in a new project do the following:

In a header file named SumFunctor declare a class name SumFunctor as follows

```
class SumFunctor {
public:
    SumFunctor(int a);

    // Overload the operator() to make it callable
    void operator()() const;

private:
   int n;
};
```

In a separate cpp file named SumFunctor, implement the class as follows

```
SumFunctor::SumFunctor(int a) {
      n = a;
}

void SumFunctor::operator ()() const {
      cout << n;
}
```

In **main.cpp** do the following (do not forget to include your class' header file)

```
#include <iostream>
#include <thread>
using namespace std;

int main() {

    // Create a thread using
    // the functor object
    thread t(SumFunctor(3));

    // Wait for the thread to
    // complete
    t.join();
    return 0;
}
```

Non-Static and Static Member Function

Implement and run the following in a new project do the following:

In a header file named MyClass declare a class name MyClass as follows

```
class MyClass {
public:
    // Non-static member function
    void f1(int num);

    // Static member function that takes one parameter
    static void f2(int num);
};
```

In a separate cpp file named MyClass, implement the class as follows

```
void MyClass::f1(int num){
      cout << num << endl;
}

void MyClass::f2(int num){
      cout << num;
}
```

In **main.cpp** do the following (do not forget to include your class' header file)

```
#include <iostream>
#include <thread>

using namespace std;

int main() {

    // Member functions
    // requires an object
    MyClass obj;

    // Passing object and parameter
    thread t1(&MyClass::f1, &obj, 3);

    t1.join();

    // Static member function can
    // be called without an object
    thread t2(&MyClass::f2, 7);

    // Wait for the thread to finish
    t2.join();

    return 0;
}
```

References

- https://www.geeksforgeeks.org/cpp/multithreading-in-cpp/