C++ 的多執行序程式開發 Thread：基本使用

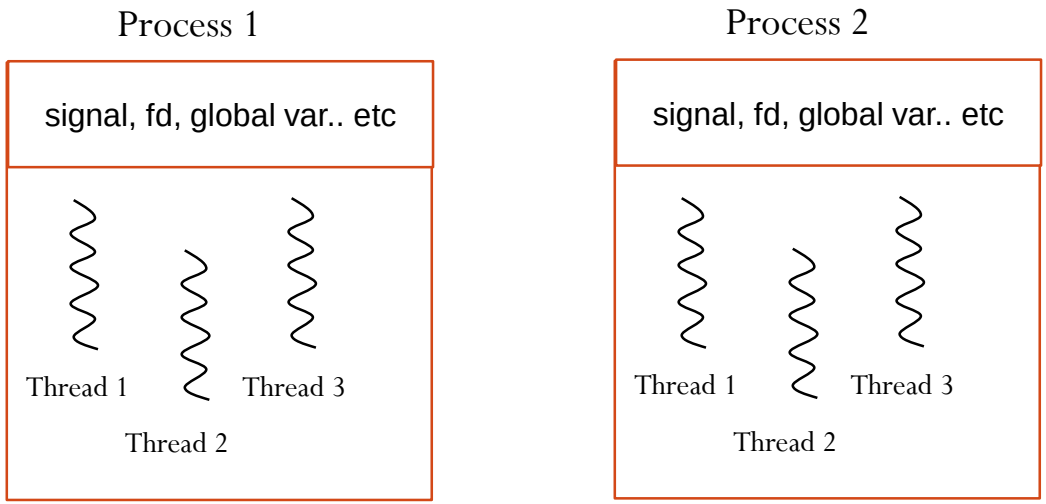
上課老師：莊啟宏

前言

- **程式 (Program)**
  - 儲存於硬碟中的可執行檔稱為 Program
- **行程 (Process)**
  - 載入記憶體中的可執行檔稱為 Process
- **執行緒 (Thread)**
  - Process中一段程式碼的執行序列稱為Thread，是作業系統能夠進行運算與排程的最小單位。

執行緒（thread）

- 簡介
  - 執行緒（thread）是被包含在行程（Process）中實際運作的單位。
  - 一個行程中可以並行多個執行緒，而這些執行緒共用同一份行程的系統資源、名稱位址等等。



- 而要來介紹的是在 C++11的 STL 新加入的「Thread」（以下稱為「STL Thread」，官方文件、MSDN）！
- 不過，雖然 STL Thread 是 C++11 標準函式庫的一部分，但是要注意的是，當時由於 C++11 算是一個很新的標準，並非所有編譯器都有支援；像是 Visual C++ 2010 就還不支援、要等到下一代的 Visual Studio 2012 才有支援。

- 基本使用
- 如果只是要產生一個新的執行序來執行額外的程式的話，STL Thread 的基本使用其實相當簡單，大致上如下：

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

using namespace std;

void test_func()
{
  // do something
  double dSum = 0;
  for( int i = 0; i < 10000; ++ i )
    for( int j = 0; j < 10000; ++ j )
      dSum += i*j;

  cout << "Thread: " << dSum << endl;
}
```

```cpp
int main( int argc, char** argv )
{
  // execute thread
  thread mThread( test_func );

  // do somthing
  cout << "main thread" << endl;

  // wait the thread stop
  mThread.join();

  return 0;
}
```

執行結果：

```
main thread
Thread: 2.4995e+015
```

- 首先，STL Thread 的 header file 是 <thread>，在使用前必須要先 include 這個檔案。
- 而要產生新的 thread，基本上就是取去建立一個新的 std::thread 的物件，在這邊就是 mThread；而在建立 std::thread 的物件的時候，可以直接把一個可以呼叫的物件（callable object、一般是現成的 function 或是 function object）當作參數傳進去，這樣在 mThread 這個物件被建立出來的時候，系統就會產稱一個新的執行序、去執行所指定的 function object 了～而在這邊，就是 test_func() 這個函式。
- 而當新的執行序開始執行後，雖然電腦會開始執行 test_func() 裡面的計算，但是同時，他也會繼續執行下面的指令，在這邊就是透過 cout 輸出「main thread」這個字串。

- 最後面去呼叫 mThread 的 join() 這個函式，則是用來告訴編譯器，在這邊要等 mThread 的計算工作完成後、才能繼續做下去；如此一來，可以避免 mThread 明明還在進行計算，但是主程式卻已經結束的問題。而如果之後的程式有要用到其他執行序的計算結果的話，也是要記得加上 join()，才能確定所需要的計算已經結束了。

- 而如果要執行的 function object 是需要參數的話，也可以直接在建立 std::thread物件的時候，直接把參數附加在建構子裡。下面就是一個例子：

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

using namespace std;

void test_func2( int i )
{
  cout << i << endl;
}

int main( int argc, char** argv )
{
  thread mThread( test_func2, 10 );
  mThread.join();

  return 0;
}
```

- 不過要注意的一點是，在把 callable object 傳遞給 STL Thread 開啟一個新的 thread 的時候，他會是採用複製的方法，把傳入的物件複製一份來用；所以如果在裡面有修改道本身的資料的話，就需要使用 std::ref() 來產生物件的參考、然後再傳進去。下面就是一個例子：

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

using namespace std;

class funcObj
{
public:
  int iData;

  funcObj()
  {
    iData = 0;
  }

  void operator()()
  {
    ++iData;
  }
};
```

```
int main( int argc, char** argv )
{
  funcObj co;

  // copy
  thread mThread1( co );
  mThread1.join();
  cout << co.iData << endl;

  // reference
  thread mThread2( ref( co ) );
  mThread2.join();
  cout << co.iData << endl;

  return 0;
}
```

- 以基本的操作方法來說，要使用 STL Thread，就是：
  - 透過建立一個新的 std::thread 物件、產生一個新的執行序
  - 在必要時呼叫 std::thread 物件的 join() 函式，確保該執行序已結束

- 這邊的 funcObj 就是一個有 call operator(operator()())(注意名字裡帶()，所以會有兩個())的類別，他被呼叫的時候，會把內部的計數器（iData）的值加 1。而在主程式裡面，第一次使用 STL Thread 執行的時候，是直接把 funcObj 的物件（co）傳進去；這時候他會在內部複製一份來執行，所以
當 mThread1 執行結束後，co 裡的 iData 的值並不會改變。

- 而當第二次執行的時候，由於傳進到 STL thread 建構子的物件是 ref(co)，所以實際上 mThread2 所執行的會是 co 這個 funcObj 物件的參考；也因此，
co.iData 就會在 mThread2 裡被修改到，等到結束後，他的值就會變成 1 了 ~

- 每個thread有自己的id 可以call this_thread::get_id()取得自己id, 或是call get_id()來得到相對應thread的id

```
#include <thread>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void func(int i, string s)
{
    cout << i << ", " << this_thread::get_id() << endl;
}

int main()
{
    vector<thread> threads;
    for(int i = 0; i < 10; i++){
        threads.push_back(thread(func, i, "test"));
    }
    for(int i = 0; i < threads.size(); i++){
        cout << threads[i].get_id() << endl;
        threads[i].join();
    }
    return 0;
}
```

問題：

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

using namespace std;

void OutputValue( int n )
{
  cout << "Number:";
  for( int i = 0; i < n; ++ i )
  {
    this_thread::sleep_for( chrono::duration<int, std::milli>( 5 ) );
    cout << " " << i;
  }
  cout << endl;
}

int main( int argc, char** argv )
{
  cout << "Normal function call" << endl;
  OutputValue( 3 );
  OutputValue( 4 );

  cout << "\nCall function with thread" << endl;
  thread mThread1( OutputValue, 3 );
  thread mThread2( OutputValue, 4 );
  mThread1.join();
  mThread2.join();
  cout << endl;

  return 0;
}
```

- 在這個例子裡，主要是透過 OutputValue() 這個函式，透過 standard output stream、cout 來輸出 0 – n 的數值；不過這邊為了拉長函式執行的時間間隔，所以有刻意使用 this_thread::sleep_for() 來在每次輸出間、停頓 5 毫秒（ms）。而在主函式裡，一開始則是先用一般的函式呼叫方法來做呼叫兩次，接下來則是用 STL Thread 建立兩個執行續、個別執行 OutputValue()。而程式執行的結果，應該會是像這樣：

```
Normal function call
Number: 0 1 2
Number: 0 1 2 3

Call function with thread
Number:Number: 0 0 1 1 2
 2 3
```

- 可以看到，一般呼叫兩次的話，會很正常地、輸出成兩行。但是如果是建立兩個執行序各自執行的話，則會因為都是透過 cout 來做輸出，所以結果都會混在一起、失去本來希望呈現的格式。
- 如果遇到這種共用資源，但是又想獨佔他的時候，該怎麼辦呢？在 STL Thread 裡，有提供一系列特別的類別、Mutual exclusion（縮寫為 mutex），就是用來處理這種問題的。在 STL Thread 的 <mutex> 這個 header 檔裡，總共提供了四種 mutex 可以視不同的需求來使用，包括了 mutex、timed_mutex、recursive_mutex、recursive_timed_mutex；而如果要以基本的 mutex 來修改上面的程式的話，大致上就像下面這樣：

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

using namespace std;

mutex gMutex;

void OutputValue( int n )
{
  gMutex.lock();
  cout << "Number:";
  for( int i = 0; i < n; ++ i )
  {
    this_thread::sleep_for( chrono::duration<int, std::milli>( 5 ) );
    cout << " " << i;
  }
  cout << endl;
  gMutex.unlock();
}

int main( int argc, char** argv )
{
  thread mThread1( OutputValue, 3 );
  thread mThread2( OutputValue, 4 );
  mThread1.join();
  mThread2.join();

  return 0;
}
```

- 這邊的重點，就是透過一個全域的mutex變數gMutex來做控制，他主要就是透過lock()和unlock() 這兩個函式，來設定變數的狀態是否被鎖定。而當在OutputValue()裡面呼叫了gMutex的lock()這個函式時，他就會去檢查gMutex是否已經被鎖定，如果沒有被鎖住的話，他就會把gMutex設定成鎖定、然後繼續執行；而如果已經被鎖住的話，他則會停在這邊、等到鎖定被解除

- 不過實際上，上面直接使用mutex的lock()和unlock()，並不是一個好辦法。因為如果lock()和unlock()之間，不小心因為return而離開 OutputValue()，就有可能出現有 lock()、但是沒有對應的unlock() 的狀況！在這種狀況下，如果又有其他執行序在等著他被解鎖，那就會產生必須一直等下去、永遠不會結束的狀況了！

- 而要避免這樣的問題產生，最好是不要直接使用mutex的lock()/unlock()，而是透過 lock_guard 這個 template class、來做mutex的控制；它的使用方法，就是：

```
void OutputValue( int n )
{
  lock_guard<mutex> mLock( gMutex );
  cout << "Number:";
  for( int i = 0; i < n; ++ i )
  {
    this_thread::sleep_for( chrono::duration<int, std::milli>( 5 ) );
    cout << " " << i;
  }
  cout << endl;
}
```

- 基本上，這邊就是透過一個型別是lock_guad<mutex>的物件 mLock、來管理全域變數 gMutex；當mLock被建立的同時，gMutex 就會被自動鎖定，而當mLock因為生命週期結束而消失時，gMutex也會因此被自動解鎖 ~ 相較於前面手動使用lock()和unlock()，使用lock_guard算是一個比較方便、也比較安全的方法。

其他種類的 mutex

- recursive_mutex
  - 可以讓 mutex 認得鎖住自己的執行序，並且讓 mutex 在已經被鎖定的情況下，還是可以讓同一個執行序再去鎖定他、而不會被擋下來。下面就是一個簡單的例子：

```
class ClassA
{
public:
  void func1()
  {
    lock_guard<recursive_mutex> lock(mMutex);
  }

  void func2()
  {
    lock_guard<recursive_mutex> lock(mMutex);
    func1();
  }

private:
  recursive_mutex mMutex;
};
```

- 在ClassA裡，有func1()和func2()兩個函式，兩者都會去建立一個lock_guard的物件、來鎖定mMutex這個mutex。不過，由於func2()裡、會去呼叫func1()，所以如果去呼叫func2() 的話，實際上可以發現，程式會在執行func2() 的時候，透過lock_guard去鎖定mMutex，而當在func2() 內去呼叫 func1() 的時候，同樣的要求鎖定動作，又會再進行一次！
- 這時候如果是使用標準的mutex的話，在第二次試圖去鎖定mMutex的時候（ func1() ），會因為mMutex已經在func2() 裡被鎖定了，而就因此停在這邊，等待mMutex被解除鎖定再繼續；但是由於mMutex的鎖定狀態必須要等到func2()整個執行完成後才會解除，所以這邊就會變成永遠等不完的狀況，讓程式無法繼續執行。

- 但是因為這邊用的是 recursive_mutex，所以在第二次、也就是在 func1() 裡試著去鎖定 mMutex 的時候，系統會判斷出目前是由同一個執行序所鎖定的，所以就讓它繼續執行下去、不會出問題。

作業一~三

- 請建立兩執行緒，第一個執行緒累加二，第二個算五階層，並使兩執行敘同時執行將其印出。
- 試使用多執行緒設計兩人賽跑，每秒跑的公尺數是1-10之間的亂數，10秒後比賽結束並顯示結果。
- 利用多執行緒模擬餐廳中，主廚烹飪晚餐和服務生送餐的動作共十次，並將結果印出。