Lesson 15 - Multi-Threading
Programming Multiple Threads of Execution

Summary

This lesson provides demonstrates how to adapt a C++ program to incorporate Multi-Threading in the Windows NT environment.

New Concepts

Multi-Threading, Mutexes, Wrapper Classes, Abstract Classes, Pure Virtual Methods.

Multi-Threading

This lesson is intended to provide an introductory demonstration of multi-threading; the practise of using multiple, concurrently executing threads of execution within the same program. Multi-threading falls under the subject of Concurrency Control which is in itself a broad and complex topic within Computing Science. We do not attempt to provide a full demonstration of multi-threading capabilities, rather we provide enough information to get started with program some rudimentary applications of threads.

Multi-Threading was not formally a feature of the C++ Language as it was in some other languages like Java or C#. All that has changed, however, in the new C++11 standard. This now gives you the option of writing your own interface to the Operating System's threads or using the threading capabilities afforded by the compiler (if it supports the new standard). In this lesson we will demonstrate both approaches; first we will create our own thread class and then we will have a play with the new threading features. It's worth knowing how to write your own thread class, as you may find yourself with a compiler which does not support the new standard, or you may wish to customise your threads in a way which is not supported by the standard (to force a thread to run on a sub-set of processors, for example).

When writing our own thread class, we will need to invoke library functions provided by the Operating System and for this lesson we shall be using Windows NT. The library function headers we need can be obtained by including the `windows.h` header file so let's begin by including it (see line 1).

```
1 #include <windows.h>
2 #include <iostream>
3 #include <vector>
4 #include <string>
5
6 using std::vector;
7 using std::string;
```

threads.cpp

Next what we want to do is provide an Abstract Thread Class, to act as a 'wrapper' around the threading library function calls of the host Operating System. The aim of a 'wrapper' class is to provide a generic class interface rather than exposing the application programmer to the specifics of the Operating System. Lines 8 to 24 provide the declaration of our Abstract Thread Class.

When we say this is an 'Abstract' class what do we mean? Notice the syntax of the run method declaration on line 17. You'll note that we assign this method to zero as if it was a variable, i.e. `run() = 0`. When we declare class methods like `run` we are saying that no concrete implementation for that method exists for this class. The effect is that the class is abstract and cannot be instantiated. What you can do however is extend that class, but you must provide an implementation of the `run` method. This may sound rather counter-intuitive now, but this is actually pretty common. For instance, in the **run** method we intend to define the work that the Thread will undertake. While the rest of the methods provide common functionality to every Thread we create, the **run** method will differ as we wish to create different Threads for different tasks.

We could have simply provided a **run** method implementation in the `Thread` class and overridden this method in any sub-class, but sometimes you simply don't know what that method should do at the super-class level. In addition, if you implement the **run** method there's no guarantee that an application programmer will override it. Using Abstract Classes is a stronger requirement that forces the application programmer to provide an implementation.

Note that the Abstract Thread class uses some types that you may not have seen before like `DWORD` (lines 15 and 20), `LPVOID` (18) and `HANDLE` (19). These are simply basic types that have been redefined in the **windows.h** file to provide a consistent set of types when programming in a Windows environment. For example, a `DWORD` is a type definition for a 'double word'; typically a 32-bit value on a 32-bit processor. A `HANDLE` is a 'void pointer', useful for allowing us to point to any type. You can find a list of these Windows type definitions at the following URL:

`http://msdn.microsoft.com/en-us/library/aa383751(VS.85).aspx`.

Returning to the Thread Class however we see that there is a public **start** method which we shall see, initiates the thread function declared on line 18. There is a **join** method which our main thread will call before terminating, and a **get_id** method for returning a unique identifier for the Thread, assigned by the Operating System.

Note that we make the Copy Constructor and Assignment Operator private. This is sometimes necessary because we want to ensure that an object cannot be copied or reassigned. We follow this style here because we want our Threads to be initiated in a strictly defined way to avoid inconsistencies arising with shared data. We'll see exactly what this means soon.

```
 8 class Thread
 9 {
10 public:
11    Thread(){}
12    virtual ~Thread(){ CloseHandle(thread_handle);}
13    virtual void start();
14    virtual void join();
15    virtual DWORD get_id() const {return tid;}
16 protected:
17    virtual void run() = 0;
18    friend DWORD thread_ftn(LPVOID T);
19    HANDLE thread_handle;
20    DWORD tid;
21 private:
22    Thread(const Thread& src);
23    Thread& operator=(const Thread& rhs);
24 };
```

threads.cpp

From line 25 to 29 we provide the definition for the `thread_ftn`. Notice that this is simply a global function declared in the abstract Thread class which being a 'friend' function, can access protected/private data. Note that `thread_ftn` performs a 'static cast' of the function parameter `T` to a Thread pointer, before calling its **run** method and returning `NULL`.

On lines 31 to 39 we define the **start** method. Notice the use of the library function `CreateThread` here. Windows provides this function which we use to create a new Thread. The important parameters to this function are the `thread_ftn` function, the `this` pointer and the `tid` address. These represent the function our thread will execute, the function parameter and the address of the variable that will hold the unique ID of this Thread, respectively.

Note that `CreateThread` requires a function as its 3rd parameter. This is why we had to make `thread_ftn` a global function as opposed to a class method. The `CreateThread` library call is defined in the C Programming Language and doesn't know how to handle class methods.

```
25 DWORD thread_ftn(LPVOID T) {
26    Thread* t = static_cast<Thread*>(T);
27    t->run();
28    return NULL;
29 }
30
31 void Thread::start() {
32    thread_handle = CreateThread(
33       NULL, // default security
34       0, // default stack size
35       (LPTHREAD_START_ROUTINE)&thread_ftn, // thread function name
36       (LPVOID)this, // argument to thread function
```

```
37        0, // use default creation flags
38        &tid);
39 }
```

threads.cpp

On examination you'll note that the `join` method simply calls another library function, `WaitForSingleObject`. When the calling thread invokes this method it will be suspended until the thread identified by `thread_handle` has finished its work. The second parameter to this function is the amount of time the calling thread should wait. We've set this to `INFINITE` so that the thread will wait indefinitely, however we could have specified a time-period here.

```
40 void Thread::join() {
41     WaitForSingleObject(thread_handle, INFINITE);
42 }
```

threads.cpp

Now that we have our abstract thread class, it's time to extend it. On lines 43 and 49 we declare two sub-classes of Thread to perform various tasks. We have a Producer on line 43 that will 'produce' a message, and a Consumer (line 49) that will 'consume' it. In a moment we'll create a shared 'buffer' to hold the message, but for now we declare that we are implementing the `run` method in our `Producer` and `Consumer` classes (lines 46 and 52).

```
43 class Producer : public Thread
44 {
45 protected:
46     virtual void run();
47 };
48
49 class Consumer : public Thread
50 {
51 protected:
52     virtual void run();
53 };
```

threads.cpp

For our shared message buffer we simply create a vector of strings (see line 58). We declare our buffer at global scope so that both our `Producer` and `Consumer` have access to it.

```
55 /**
56 * shared buffer
57 */
58 vector<string> buffer;
```

threads.cpp

Next we define the `run` methods for our `Producer` and `Consumer` classes. First take a look at the `Producer`'s `run` method on line 55. On line 57 we simply call the `push_back` method of the buffer to add a new message. On lines 56 and 58 we 'lock' and 'unlock' a 'mutex' object. We'll shortly explain the purpose behind these statements so ignore them for now. Now take a look at the `run` method for the `Consumer` thread on line 65. Here we are checking to see if there is a message in the buffer by examining its size (line 69). If there is, we display the message to screen (line 70) and set a 'flag' called `done` to be true. This indicates that the `Consumer` thread is finished and it subsequently exits the loop. Our simple example only expects one message. Again you'll note the calls to 'lock' and 'unlock' a 'mutex' object on lines 68 and 73.

```
59 void Producer::run() {
60     mut.lock_mutex();
61     buffer.push_back("Hello from Producer\n");
62     mut.unlock_mutex();
63 }
64
65 void Consumer::run() {
66     BOOL done = FALSE;
67     while(!done) {
68         mut.lock_mutex();
69         if(buffer.size() > 0) {
70             std::cout << "got msg: " << buffer.front() << "\n";
71             done = TRUE;
72         }
73         mut.unlock_mutex();
74     }
75 }
```

threads.cpp

Now what was the purpose behind those calls to 'lock' and 'unlock' a 'mutex'? A 'mutex' (which is short for mutual exclusion), is a way of granting exclusive access to an object when there exist multiple threads who can interact with that object. The idea is that each thread attempts to 'lock' the mutex before it trys to access a shared object. Only one thread can 'hold the lock' at any one time, so while one thread will succeed in acquiring it, the remaining threads are forced to wait. The remaining threads typically wait until the owning-thread relinquishes the lock. When the lock is released, the waiting threads can try again.

We want to alleviate the details of this process from the user-programmer as much as possible. Calls to create, lock and unlock a mutex require the invocation of library functions, as with our Threads. Once again we will create a wrapper object around these library calls in the form of a `MutexClass`, which we can use to create mutex objects. See lines 76 to 85 for our `MutexClass` declaration. Note that it consists of a `HANDLE` to a mutex, in the form of a `protected` data field. In addition there are methods `lock_mutex` and `unlock_mutex` to 'lock' and 'unlock' the mutex, respectively.

```
76 class MutexClass
77 {
78 public:
79    MutexClass();
80    virtual ~MutexClass();
81    virtual void lock_mutex();
82    virtual void unlock_mutex();
83 protected:
84    HANDLE mutex;
85 };
```

threads.cpp

Our `MutexClass`' Constructor calls the library function `CreateMutex` and saves the return value in our `mutex` data field (see lines 86 to 90). We also implement a Destructor that calls the library function `CloseHandle` on the `mutex` field, so this `HANDLE` is removed after we're done with it.

```
86 MutexClass::MutexClass() {
87    mutex = CreateMutex(
88       NULL, // default security
89       FALSE, // initially not owned
90       NULL); // unamed mutex
91 }
92
93 MutexClass::~MutexClass() {
94    CloseHandle(mutex);
95 }
```

threads.cpp

In the `lock_mutex` method we call the library function `WaitForSingleObject`. When this function is invoked, the calling thread will attempt to acquire the lock on the mutex. If the lock is already held by another thread, it will wait until the other thread releases it. All this is taken care of by the library function, we simply have to provide a time limit for the wait. Note however that we pass the argument `INFINITE`, denoting that we're willing to wait indefinitely. In a real application we would perform some error checking on the return value of this function, but in the interests of succinctness, we have omitted those steps here. Note the `unlock_mutex` method simply calls the `ReleaseMutex` library function (line 103), thus relinquishing the lock.

```
96 void MutexClass::lock_mutex() {
97    WaitForSingleObject(
98       mutex, // handle to mutex
99       INFINITE); // no time-out interval
100 }
101
102 void MutexClass::unlock_mutex() {
103    ReleaseMutex(mutex);
104 }
```

threads.cpp

Now we create a global shared mutex object from the `MutexClass` (line 108). Take another look at the `run` methods of the `Producer` and `Consumer` threads on lines 59 and 65 respectively. In particular, note our use of the mutex to guard access to the shared buffer by calling the `lock_mutex` and `unlock_mutex` methods.

```
105 /**
106 * global mutex object
107 */
```

```
108 MutexClass mut;
```

threads.cpp

Finally, we implement the `main` function. We begin by creating a `Producer` and `Consumer` thread on the stack (lines 111 and 112). Next we start the `Consumer` thread on line 114. After this statement has executed there will exist a new thread of execution, scheduled to be executed by the Operating System. There will also be a "main" thread of course, that was created by the Operating System to execute the main function.

We then invoke another library function called `Sleep` on line 115. This function will tell the currently executing thread to sleep for the amount of milliseconds specified in the `Sleep` function's argument (1000). The currently executing thread is the 'main' thread, hence the 'main' thread will be suspended for 1000 milliseconds before it starts the `Producer` thread on line 116.

While 'main' is suspended it's very likely that the `Consumer` thread will execute its `run` method, now that it has been started. However, because the buffer size is still 0 (see the test on line 69), it will continue to loop until the `Producer` thread is started and deposits a message in the buffer. This doesn't take place until the 'main' thread awakes from the `Sleep` function and starts the `Producer` thread on line 116.

Once the 'main' thread has completed step 116, it waits for the `Producer` and `Consumer` to complete by calling their `join` methods. If 'main' didn't carry out these steps it may reach the end of the `main` function before the `Producer` and `Consumer` had a chance to finish their `run` methods. This is bad from the perspective of `Producer` and `Consumer` because when the `main` function completes, the program is terminated.

```
109 int main() {
110 
111    Producer prod;
112    Consumer cons;
113 
114    cons.start();
115    Sleep(1000);
116    prod.start();
117 
118    prod.join();
119    cons.join();
120 
121    return 0;
122 }
```

threads.cpp

New Feature

We've seen that writing our own thread class has not required too much code. Creating and launching a new thread in C++11, however, is even simpler, so let's do just that. We begin with including the `thread` and the `mutex` header files on lines 1 and 2 respectively. We then create a `Counter` class, to encapsulate thread-safe operations on a shared counter (lines 5 to 22). The `Counter` has methods to `increment` (lines 11-14) and `decrement` (lines 16-19) a shared counter, called `mCount`, which we have declared on line 6.

```
1  #include <thread>
2  #include <mutex>
3  #include <iostream>
4 
5  class Counter {
6     long mCount;
7     std::mutex mMutex;
8  public:
9     Counter(): mCount(0), mMutex() {}
10 
11    void increment() {
12       std::lock_guard<std::mutex> guard(mMutex);
13       mCount++;
14    }
15 
16    void decrement() {
17       std::lock_guard<std::mutex> guard(mMutex);
18       mCount--;
19    }
20 
21    long count() { return mCount; }
```

```
22 };
```

cpp11threads.cpp

We make the `Counter` class thread safe by using a `std::mutex` provided in the new C++11 standard (see line 7). Before we increment `mCount` we lock the `mutex` via a `std::lock_guard` object (see line 12). We could have called the lock method of `mMutex` directly, but we would have use a `try\catch` block to ensure that we unlocked the mutex in the event of an exception being thrown. By using the `std::lock_guard` approach instead, we let the guard ensure the mutex is released for us.

```
23 void increment(Counter* counter) {
24    for(int i = 0; i < 100000; ++i) {
25       counter->increment();
26    }
27 }
28
29 void decrement(Counter* counter) {
30    for(int i = 0; i < 100000; ++i) {
31       counter->decrement();
32    }
33 }
34
35 Counter counter;
36
37 int main(int argc, char* argv[]) {
38
39    std::thread producer(increment, &counter);
40    std::thread consumer(decrement, &counter);
41
42    std::cout << "Launch" << std::endl;
43    std::chrono::milliseconds duration(1000);
44    std::this_thread::sleep_for(duration);
45
46    producer.join();
47    consumer.join();
48
49    if(counter.count() != 0) {
50       std::cout << "race condition: " << counter.count() << std::endl;
51    }
52    return 0;
53 }
```

cpp11threads.cpp

We now turn to our threads. As C++11 already provides a thread class (`std::thread`), we simply create a `producer` and `consumer` thread (see lines 39 and 40). We pass to their constructors a function to execute and a pointer to a global `Counter` object. The ‘producer’ thread is passed the `increment` function (lines 23-27) and the ‘consumer’ thread is passed the `decrement` function (lines 29-33) and each thread begins working as soon as it is created.

Meanwhile in the `main` thread, we create a single second time duration with a `chrono::milliseconds` object (line 43). We then send the `main` thread to sleep using the `this_thread::sleep_for` method. Before `main` can exit, we have it join with both the `producer` and `consumer` (line 46 and 47). Finally, we check the result of the `counter` to ensure a race condition did not occur. As both the `producer` and `consumer` modify the counter an equal number of times, the counter should end up with a value of zero.

Exercises

1. Create a ‘hello world’ thread which executes a loop, printing to screen the message ‘Hello from Thread (handle)’ 10 times. Extend the Thread class provided.
2. Create a thread-safe Binary Search Tree class which uses a single lock to ensure only one thread at a time can insert comparable objects (this is coarse-grained locking).
3. Create a thread-safe Binary Search Tree class which uses a lock-per-node so that threads can insert comparable objects concurrently (this is fine-grained locking).