Concurrent and Parallel Systems

Unit 1 -Multi-threading

1.1 -Starting a Thread

Starting a thread:
Creating a thread in C++ is simple. We need to include the thread header file

- #include <thread>

To create a thread, we then only need create a new thread object in our application, passing in the name of the function that we want the thread to run

- thread t(hello_world);

This will create a new thread that will run the function hello_world(). The thread will start executing the function while the main application continues operating on the rest of the code. Essentially the main application is a thread that is created and executed as soon as an application is launched.

Waiting for a thread to complete:
Generally we want to wait for a thread to complete its operation. We do this by using the join method on the thread

- t.join();

This will mean the currently executing code (whether it be the main application thread or another thread) will wait for the thread it joins to complete its operation.

1.2 -Multiple Tasks

Starting one thread is all well and good, but we really want to execute multiple tasks. Creating multiple threads is easy - we just create multiple thread objects. We can use the same function if we want (useful for parallelising a task), or we can run different functions (useful for tasks which block computation).
We will use a new operation during this next example;

- sleep_for(seconds(10));

This operation will allow us to put a thread to sleep for an amount of time.

To allow us to access the duration constructs (which contains seconds, etc) we use the chrono header

- #include <chrono>

1.3 -Passing Parameters to Threads

We have now seen how to create threads in C++ 11 and how to put them to sleep. The next piece of functionality we are going to use is how we pass parameters to a thread. This is actually fairly easy, and just requires us adding the passed parameters to the thread creation call. For example, given a function with the declaration:

- void task(int n, int val)

We can create a thread and pass in parameters to 'n' and 'val' as follows:

- thread t(task, 1, 20);

'n' will be assigned the value 1 and 'val' will be assigned the value 10. Our test application will use random number generation, which has also changed in C++11.

-Random Numbers in C++ 11

C++ 11 has added an entirely new random number generation mechanism which is very different, as well as having more functionality. To use random numbers we need to include the random header

- #include <random>

We then need to create a random number generation engine. There are a number of generation engines, but we will use the default one in C++11.

- default_random_engine e(seed);

'seed' is a value used to seed the random number engine (you should hopefully know by now that we cannot create truly random numbers so a seed defines the sequence the same seed will produce the same sequence of random numbers). We will use the system clock to get the current time in milliseconds to use as a seed. To get a random number from the engine we simply use the code;

- auto num = e ();

-Ranged For Loops in C++11

The other new functionality we will use in this example is a ranged for loop. You will probably be familiar with the foreach loop in C#. The ranged for loop in C++11 has the same functionality.

- for (auto &t : threads)

We can then use t as an object reference in our loop. The threads variable is a collection of some sort (here a vector, but we can use arrays, maps, etc.). This is much easier than the iterator approach in pre-C++11 code.

1.4 -Using Lambda (λ) Expressions

Lambda (λ) expressions are another new feature of C++11, and are becoming a popular feature in object-oriented languages in general (C# has them, Java 8 will have them). λ expressions come from functional style languages (for example, F#, Haskell, etc). They allow us to create function objects which we can apply parameters to and get a result.

-What is a λ Expression?

A λ expression is essentially a function. However, it has some properties that allow us to manipulate that function. For example, if we define a function as follows

- add(x,y) = x + y

We can then use this function object to create a new function object that adds 3 to any parameter:

- add3(y) = add(3)

We can then use the add3 function as a new function. We can then get the results from function calls as follows

- add(10,5) = 15
- add3(10) = 13
- add(2,0) = 2
- add4(y) = add(4)
- add4(6) = 10

-λ Expressions in C++11

One advantage of C++ λ expressions is that they allow us to create functions using fewer lines of code. Traditionally we would create an entire function / method definition with parameters and return type. In C++11, we can simplify this. For example, we can create an add function shown below:

- auto add = [] (int x, int y) { return x + y; };

The [] allows us to pass parameters into the function (we will look at this capability later). We define the parameters we want to pass to the function. Finally, in the curly brackets we define the λ expression. We can then use the add function as a normal function:

- auto x = add(10, 12);

We are brushing the surface of λ expressions here, but it is enough for what we are doing in the module.

1.5 -λ Expressions and Threads

We can use λ expressions to create threads, making our code more compact (but not necessarily easier to read).

For the rest of the module, it is up to you whether you want to use λ expressions or not. We will work interchangeably, and there is no effect on your grades.

1.6 -Gathering Data

Throughout this module, we will be gathering profiling data to allow us to analyse performance of our applications, and in particular the speed up. To do this, we will be outputting data into what is called a comma separated value (.csv) file. This is a file that we can load easily into Excel and generate a chart from. In this first application, we will just get the average of 100 iterations, and our work will be simple (it will just spin the processor and do nothing).

-Creating a File

Creating a file in C++ is easy, and you should know this by now. As a reminder, we create an output file as shown below:

- ofstream data(“data.csv”, ofstream::out);

This will create an output file called data.csv which we can write to. We can treat the file just as any output stream (e.g. cout).

-Capturing Times

To capture times in C++11, we can use the new system_clock object. We have to gather a start time, and end time, and then calculate the total time by subtracting the first from the second. Typically, we will also want to cast the output to milliseconds (ms).

-Getting the Data

Once the application is complete, you should have a .csv file. Open this with Excel. Then get the mean of the 100 stored values (there is a good chance that a number of the recorded values are 0).
You also want to document the specification of your machine. This is very important. You can find this in the device manager of your computer. For example, the result from my test is 0.27ms. The hardware I ran the test on is:

- CPU - Intel i5-2500 @ 3.3GHz
- Memory - 8GB
- OS - Windows 7 64bit

This is the only pertinent information at the moment. As we progress through the module, we will find that the other pieces of information about your machine will become useful.

1.7 -Monte Carlo π

We now move onto our first case study - the approximation of π using a Monte Carlo method. A Monte Carlo method is one where we use random number generation to compute a value. Because we can just increase the number of random values we test, the problem we use can be scaled to however many computations we wish. This will allow us to actually perform a test that we can push our CPU with.

Theory

The theory behind why we can calculate π using a Monte Carlo method can best be described by figure 1.5. The radius of the circle is refined as R. We know that the area of a circle can be calculated by the following equation:

- πR^2

We can also calculate the area of a square as follows:

- 4R^2

Now let us imagine that R=1, or in other words we have a unit circle. If we pick a random point within the square, we can determine whether it is in the circle by calculating the length of the vector equivalent of the point. If the length is 1 or less, the point is within the circle. If it is greater than 1, then it is in the square but not the circle. The ratio of radom points tested to ones in the circle allows us to approximate π. This is because:

- AreaC = πR^2
- AreaS = 4R^2
- Ratio = AreaC / AreaS = π / 4

We can then create an algorithm to approximate π as shown:

Algorithm 1 Monte Carlo π

```
begin
        attempts : = N
        in_circle := M
        ratio := M/N
        π/4 ~= M/N => π ~= 4 * M/N
end
```

We can therefore approximate by generating a collection of random points (we will work in the range [0.0, 1.0] for each coordinate), checking the length and if it is less than or equal to 1 we count this as a hit in the circle.

Distributions for Random Numbers

Before looking at our C++11 implementation of the Monte Carlo π algorithm, we need to introduce the concept of distributions for random numbers. A distribution allows us to define the type of values we get from our random engine and the range of the values. There are a number of distribution types. We will be using a real distribution (i.e. for double values) and we want the numbers to be uniformly distributed (i.e. they will not tend towards any particular value). We also define the range from 0.0 to 1.0. Our call to do this is shown:

- //Create a distribution
- uniform_real_distribution<double> distribution(0.0, 1.0);
- //Use this to get a random value from our random engine e
- auto x = distribution(e);

Our algorithm implementation in C++ is shown in Listing 1.27.

We have used all the pieces of this algorithm by now, so you should understand it. Notice that we are not actually getting the return value for π at the moment you can print it out if you want to test the accuracy. We will look at how we can get the result from a task in later tutorials.

Main Application

Unit 2 -Controlling Multi-Threaded Applications

Control of multi-threaded applications is really important due to the problems encountered with shared memory. However, controlling multi-threaded applications also leads to problems. The control of multi-threaded applications is really where the concept of concurrency comes in.
The second half of this tutorial will look at some more control concepts;

- Atomics
- Futures

-Atomics are a simple method of controlling individual values which are shared between threads.
-Futures allow us to spin off some work and carry on doing other things, getting the result later.

2.1 -Shared Memory Problems

First we will see why shared memory is a problem when dealing with multi-threaded applications. We are going to look at a simple application which increments a value, using a pointer to share the value between all the threads involved. The example is somewhat contrived, but illustrates the problem.

-Shared Pointers in C++11

Smart pointers have made the world of the C++ programmer much easier in that you no longer need to worry about memory management (well, not as much anyway). In this example, we will use a shared_ptr. A shared_ptr is a pointer that is reference counted. That is, every time you create a copy of the pointer (i.e. provide access to a part of the program), a counter is incremented. Every time the pointer is not required by a part of the program, the counter is decremented. Whenever the counter hits 0, the memory allocated is automatically freed. This does have a (very slight) overhead, but will make our life easier.
For this example, we are going to use the make_shared function. make_shared will call the relevant constructor for the defined type based on the parameters passed. For example:

- auto value = make_shared<int>(0);

The type of value (automatically determined by the compiler) is shared_prt<int>. We can now happily share this value without our program

(with multi-threaded risks) by passing the value around.
To use smart pointers, we need to include the memory header:

- #include <memory>

-Application

Our application is quite simple - it will increment our shared value. Listing 2.3 illustrates.

Notice what we do on line 6. We dereference the pointer (this still works for shared_ptr) to get the value stored in the shared_ptr and add one to the value, setting the shared_ptr value with this new incremented value. Our main application is shown in listing 2.4.

Notice on line 7 we use a call to thread::hardware_concurrency. This value is the number of threads that your hardware can handle natively - a useful value if you understand our results from the last tutorial.
As increment adds 1 million to the shared value, depending on your hardware configuration you might expect a printed value of 1 million, 2 million, 4 million, maybe even 8 million. However, running this application on my hardware I get the result shown in figure 2.1. Somehow I lost almost 2 million increments. So where did they go?

This is the problem of shared resources and multi-threading. We do not actually know which order the threads are accessing the resource. For example, if we consider a 4 thread application, something like Figure 2.2 might happen.

Even though each thread has incremented the value twice (you can check) and the value should be 8, the actual value is 2. Each thread is competing for the resource. This is what we call a RACE CONDITION (or race hazard). This is an important concept to grasp and you need to understand that sharing resources in multi-threaded applications is inherently dangerous if you do not add some control (control does bring its own problems).

2.2 -Mutex

The first approach to controlling multi-threaded applications is using what's called a mutex. A mutex (mutual expression) allows us to guard particular pieces of code so that only one thread can operate within it at a time. If we share the mutex in some manner (mutexes cannot be copied so have to be shared in some manner) then we can have different sections of code which we protect, ensuring that only one of these sections is running at a time.
To use a mutex, we need to include the mutex header.

- #include <mutex>

We will also need a mutex as a global variable for our program

- mutex mut;

We can then simply update our increment method from the previous application

to use the mutex. The two important methods of the mutex we will be using are lock (to lock the mutex thus not allowing any other thread to successfully lock) and unlock (freeing the mutex, allowing another thread to call lock). Any thread which cannot lock the mutex must wait until the mutex is unlocked and it successfully acquires the lock (there might be a queue of waiting threads).
With a mutex in place, we can update our increment function to that shown in Listing 2.6. If you run this application (and note that it takes a little longer) your result should be as expected (4000000).

2.3 -Lock Guards

Remembering to lock and unlock a mutex can cause problems (especially when you forget to lock and unlock) and does not necessarily make your code simple to follow. Other problems that can occur come from methods having multiple exit points (including exceptions) which means you might miss when to unlock a mutex. Thankfully, thanks to C++s object deconstruction at the end of scopes, we can utilise what is known as a lock guard to automatically lock and unlock a mutex for us.
Modifying our application to use lock guards is very simple. A change to the increment function is required as shown in Listing 2.7.
Using mutexes and lock guards is the simplest method of protecting sections of code to ensure shared resources are protected. However, their simplicity does lead to other problems (such as dealock) which we need to overcome.

2.4 -Condition Variables

One limitation when using mutexes is that we are only really controlling access to certain sections of the application to try and protect shared resources. You can think of it as having a gate. We let one person in through the gate at any one time, and do not let anyone else enter until the person has left. This is all well and good, but we have no control over what happens outside the gate (where arguments over who is next in line may occur).
Another approach we might want to take is waiting for a signal - a sign to state that we can perform some action. We could happily wait, and when the signal is activated we stop waiting and carry on doing some work. This is a technique originally defined using what was called a semaphore.
Consier what happens are a set of traffic lights at a crossroads. We have two streams of traffic that wish to use the intersection at the same time. By having a set of lights (think of red as meaning wait and green meaning signal), we control access to the crossroads. A semaphore (and in C++11 a condition variable) allows us to control access in a similar manner to a set of traffic lights at a crossroads.
To use condition variables, we need to include the condition_variable header

- #include <condition_variable>

We are going to look at three operations to use with a condition variable. First there is wait

- condition.wait(unique_lock<mutex>(mut));

This will enable us to wait until a signal has been received. We have to pass a

lock to the wait method, and as we are not using the lock anywhere else we use unique_lock. This ensures that we are waiting on the mutex.
The next method we are interested in is notify_one

- condition.notify_one();

This will notify one thread that is currently waiting on the condition variable. There is a similar method called notify_all which will signal all waiting threads.
The final method we are interested in is wait_for

- if (condition.wait_for(unique_lock<mutex>(mut), seconds(3)))

As you can see, wait_for returns a true or false value. If we are signalled (notified) before the time runs out, then wait_for returns true. Otherwise, it returns false. There is a similar method called wait_until which allows you to set the absolute time to stop waiting at.

-Application

We are going to create 2 threads which wait and signal each other, allowing interaction between the threads. Our first thread runs the code in Listing 2.12, and our second will run the code in listing 2.13. Finally, our main is given in listing 2.14.

Note the use of the ref function on lines 7 and 8. This is how we create a reference to pass into our thread functions. The interactions between these threads have been organised so that you will get output shown in Figure 2.4.

2.5 -Guarded Objects

Now that we know how to protect sections of code, and how to signal threads, let us consider how we go about doing thread safe(ish) code in an object-oriented manner. We are going to modify our increment example so that an object controls the counter. First of all we need to define a header file

- #include “guarded.h”

Listing 2.15 provides the contents.
Notice that we provide the object with its own mutex. This is to protect access to our value. We then use a lock_guard to control access to the increment method. This goes in our guarded.cpp file. The contents of which can be found in Listing 2.16.
The method is somewhat contrived to force multiple operations within the method. Finally, our main method is shown in Listing 2.17.
Your output window should state that value equals 4 million. The use of a lock_guard and an object level mutex is the best method to control access to an object. This will ensure that methods are only called when permitted by competing threads.

2.6 -Thread Safe Data Structures

To end the first part of this tutorial, we will look at how we can implement a thread safe stack. Data structures are the normal method of storing data, but are the most susceptible to multi-threading problems. This example is taken from C++ Concurrency in Action (slightly modified).

-Overview
A stack is one of the simplest data structures. We simply have a stack of values which we can add to the top of (push) or remove from the top of (pop). Our implementation will be very primitive, and will just wrap a standard stack in an object with thread safe operations.
You will need to create a new header file
- threadsafe_stack.h

First we declare our class and constructures as in Listing 2.18.

Notice that we have a copy constructor. When it is copying it must lock the other stack to ensure its copy is correct. Also note the use of the keyword mutable on line 15. This keyword indicates that the mutex can be modified in const methods (where normally we do not mutate an object's state). This is a convenience as our mutex does not affect other classes, but we still want to have const methods.

-Push
Our push method is quite trivial - all we need to do is lock the stack for usage. The code is in Listing 2.19.

-Pop
Pop is a little bit more involved. We still have to lock the object, but we must also check if the stack is empty before attempting to return a value. The code is provided in Listing 2.20.

On line 7, we check if the internal stack is empty, and if so throw an exception. There is a very good chance you will have never worked with exceptions in C++ before (it is newish but prior to C++11). They essentially work the same as Java and C#.

-Empty
Our final method allows us to check if the stack is empty (Listing 2.21).

-Tasks
Our test application is going to have one thread add 1 million values to the stack, and the other extract 1 million values from the stack. The approach is not the most efficient (using exceptions to determine if the stack is empty is not a good idea really), but will provide you with an example of exception handling in C++.
Our first task is called pusher - it's job is to push values onto the stack (Listing 2.22).
Notice the use of yield on line 8. This means that the thread will let another thread in front of it if one is waiting. We are adding this to make the pusher yield to our other task (popper), meaning the stack will appear empty

sometimes. Popper is defined in Listing 2.23.
Popper will try and pop a value from the stack. If it is empty, it will catch an exception and print it. The try-catch construct is similar to Java and C#.

-Main Application

Our main application just needs to create our resources, start the two tasks, and then check if the stack is empty at the end (1 million values pushed minus 1 million values popped). It is shown in Listing 2.24.

Remember that 1 = true, so the stack is empty.