session 17 - multi-threading

We are going to look at multi-threading in c++. Historically, multi-threading capabilities were provided by os-specific libraries. However, with c++11, we now have the `threading` library. We are going to take a look at how to use this library presently. But first, lets talk about threading in general.

Concurrency is the ability of a process to make progress on more than one task; this could happen in parallel, as is the case when running multiple threads on multiple cores. Or, this could happen by switching repeatedly between tasks, providing forward progress on all of them, regardless of their complexity.

A couple of things to note. In python, we don't typically make use of threading, as Python has something called the GIL, which should make a bit more sense after this session. Instead, Python provides a multi-processing model via pyprocessing.

Threads vs Processes in Concurrent Programming

A process is an execution context with its own environment, memory, processor state, etc. Communication between processes is handled via message passing, as is synchronization. A process is a heavy weight entity which takes a long time to create in relation to a thread.

A thread is a light weight sequence of instructions which can be executed concurrently with other sequences in a multi threading environment, while sharing the same address space. Because threads share the same address space, much more care must be taken when programming them, lest they stomp on each other. However, due to their light weight nature, threads are very cheep to create compared to processes. And because they share a unified address space, one does not have to employ complicated messaging frameworks to communicate between them.

But enough babbling. Lets jump in and take a look at threads in action. Starting with some boilerplate...

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

int main() {
    simple_thread();
    return 0;
}
```

As you can see, we need to include `thread`. This will become important when we implement the `simple_thread`function. Lets do that now.

```
void simple_thread() {
    std::cout << 'I am a thread!" << std::endl;
```

```
}
```

And lets use this function in main.

First we are going to need to create a thread. We do this with the `std::thread` constructor, which can take a function. Once we create a thread, its payload starts running. Next, we need to get the main thread to wait for t1 tofinish. We do this by calling `join` on the thread. You may only call join once on a thread. Otherwise, you willcrash your program. This isn't a problem in our trivial examples, but it can be in longer programs. In order to help you out, threads provide the `joinable` method to test for this.

A thread may also be `detatched` instead of joined, if you have no need to synchronize execution. However, if your main thread finishes before your detatched thread, your program will terminate.

```
int main() {
    // initialize the thread
    std::thread t1(simple_thread); // t1 starts running.

    // wait for it to finish
    if( t1.joinable() )
        t1.join();

}
```

Our example is incredibly simple. The main thread doesn't do anything but wait. Clearly, there is no reason to even use threading with a problem this simple. Let's add some work in the main thread.

```
int main() {
    std::thread t1(function_simple);

    for(int i=0; i<100; i++) {
        std::cout << "from main " << i << std::endl;
    }

    t1.join();
    return 0;
}
```

Believe it or not, even in this trivial program, we already have a potential problem. What happens if an exception gets thrown from main before we call t1.join ? This could spell trouble as we wont be able to clean up. Let's fix this.

We are going to catch any thrown errors in our main thread, and, if we do catch an error, we are going to call join on t1, and then re-throw the caught error.

```
int main() {
```

```
    std::thread t1(function_simple);

    try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();

    return 0;
}
```

Of course, we could avoid this by using RAII to create a class which wraps t1, and which calls join if necessary whenthe wrapper instance is destroyed. However, I am going to leave that for an exercise...

The thread constructor can take any callable as a parameter. That means we can pass it a functor. Lets define a simple functor and play a bit.

```
class Fctor {
public:
    void operator()() {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor " << i << std::endl;
        }
    }
};
```

And lets use it

```
int main() {
    Functor fct;

    std::thread t1(fct);
     try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();
```

```
    return 0;
}
```

When we run this, we get quite a mess. But before we look at cleaning this up, lets take a look at how to pass parameters to the wrapped function.

```
class Fctor {
public:
    void operator()(string msg) {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor says " << msg << std::endl;
        }
    }
};
```

Here is how we pass the string:

```
int main() {
    Functor fct;

    std::string s = "I am a thread!";
    std::thread t1(fct, s);
     try {
        for(int i=0; i<100; i++) {
            std::cout << "from main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();
    return 0;
}
```

What if we want to pass that string by reference in order to be a bit more efficient? Well, threads pass data by value. If we simply do the following, it won't behave as expected:

```
void operator()(std::string& msg){
    ...
}
```

If we really want to do this, we have to use a reference wrapper when passing the string.

```
std::thread t1(fct, std::ref(s) );
```

Another thing to be aware of. If we try and get a bit more terse and construct the functor in the thread constructor,we will run into "c++'s most vexing parse".

Basically , the following will not do what we want:

```
std::thread t1(Fctor(), s);
```

Instead, we are going to have to wrap the Functor instantiation in a pair of parens:

```
std::thread t1((Fctor()), s);
```

Threads cannot be copied. If you need to transfer ownership of a thread, you must `std::move` it.

thread ids

Threads all have unique ids. You can call `get_id()` on a thread to retrieve it's id. You can retrieve the id of the parent thread using `std::this_thread::get_id()`.

cpu availability

Under normal circumstances, you should avoid creating more threads than you have processors to handle them. In order to determine what that number is programmatically, you can call `std::thread::hardware_concurrency()`.

Example - putting it all together

```
class Fctor {
public:
    void operator()(std::string& ;msg) {
        for(int i =0; i>-100; i-- ) {
            std::cout << "from Fctor " << i << " " << msg << std::endl;
        }
    }
};

int main() {
    std::cout << "Number of available procs " << std::thread::hardware_concurrency() << std
    std::cout << "Main thread's id " << std::this_thread::get_id() << std::endl;

    std::string s = "I am a thread!!!!";
    std::thread t1((Fctor()),std::move(s));

    std::cout << "t1 id " << t1.get_id() << std::endl;

    try {
```

```
        for(int i=0; i<100; i++) {
            std::cout << "From main " << i << std::endl;
        }
    } catch (...) {
        t1.join();
        throw;
    }

    t1.join();

    return 0;
}
```

Data Races and What to Do About Them

In our previous example, you undoubtedly noticed that the output between the t1 thread and the main thread was interleaved. Why did this happen? Both threads were competing for a common resource - cout. We can fix this by using a *mutex*.

Mutex stands for **mu**tual **ex**clusion object. A mutex is an os primitive which is used to protect a resource.

Let's return to our previous example and fix it up a bit.

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

using namespace std;


void shared_print(string msg, int id) {

}

void function_1() {
    for(int i =0; i<100; i++) {
        shared_print(string("from t1:", i);
    }
}

int main() {
    std::thread t1(function_1);

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

```
        shared_print(string("From main: "), i);
    }

    t1.join();

    return 0;
}
```

Ok Let's implement shared_print now. first, I create a global mutex. Then i define the function.

```
mutex mu;

void shared_print(string msg, int id) {
    mu.lock();
    cout << msg << " " << id << endl;
    mu.unlock();
}
```

Now `cout` in shared_print is protected. Lets run this. It is really important to understand what is going on here.

Mutex.... Mutex?

So what is a mutex? Well, essentially, a mutex is simply a special class which acts like an integer in memory, but with a couple of special methods.

mutex.unlock

Unlock is quite simple. It essentially decrements the value of the mutex by one, if it has been locked. If the value of the mutex is zero, it is available for another bit of code to call lock on.

mutex.lock

Locking is a bit trickier. There can only be one lock owner of a mutex at a given time. If another thread wishes to lock the mutex while it is already locked, then the other thread must wait for the owning thread to unlock it. Until the mutex is unlocked, the thread calling lock will be unable to make progress. Attempting to lock an already locked mutex is called *contention*. Locking is a bit magic. In fact, it cannot be implemented without hardware support, because testing and setting the value has to be an atomic operation. Fortunately, all modern hardware provide appropriate implementations.

Problem - exceptions

As before, we have an issue if the code between mu.lock and mu.unlock throws an exception. We could add a try catch block or implement a wrapper via RAII, but the library does this for us already.

```
std::mutex mu;

void shared_print(string msg, int i) {
    lock_guard<mutex> guard(mu); // RAII
    cout << msg << " " << id << endl;
}
```

Problem - cout still callable

We generally want to package the mutex with the resource we are trying to protect in order to make it impossible to access the resource directly. Here is an example class which handles this:

```
class LogFile {
    std::mutex m_mutex;
    ofstream f;
public:
    LogFile() {
        f.open("/tmp/log.txt");
    }

    void shared_print(string id, string msg) {
        std::lock_guard<std::mutex> locker(m_mutex);
        f << "From " << id << ": " << value << endl;
    }
};

void function_1(LogFile& log) {
    for(int i=0; i>-100; i--) {
        log.shared_print(i, string("From t1"));
    }
}

int main() {
    LogFile log;
    std::thread t1(function_1, std::ref(log));

    for(int i=0; i<100; i++) {
       log.shared_print(i, string("From Main"));
    }
```

```
    t1.join();
    return 0;
}
```

Note that you need to make certain that your class truly protects the resource under guard. For instance, if you implement a method which returns a reference to f (the ofstream). And never allow a user defined function to operateon f.

Problem - Deadlock

Deadlock happens when no threads can make progress. You can get yourself into trouble if you are managing multiple mutexes and you lock them by hand. If, for instance, you lock them in different orders in different threads, you can end up with deadlock. In order to help manage locking in appropriate order, C++ has the std::lock() call.

Lets look at an example. Lets say we have a class representing a shipping container which contains things, and which provides a mutex.

```
struct ShippingContainer {
    explicit ShippingContainer(int num) : things{num} {};

    int things;
    std::mutex m_mu;
};
```

Furthermore, lets say we have a transfer function which transfers things form one container to the other. We certainly don't want our things to go missing, so we need to handle this appropriately. Here is our function:

```
void transfer(ShippingContainer &from, ShippingContainer &to, int num) {
    //
    std::unique_lock<std::mutex> lock_from(from.m_mu, std::defer_lock);
    std::unique_lock<std::mutex> lock_to(to.m_mu, std::defer_lock);

    // lock both not worying about order
    std::lock(lock_from, lock_to);

    if (num > from.things)
        num = from.things;

    from.things -= num;
    to.things   += num;
    // unlocking happens in destructors.
}
```

Now lets use this function to transfer stuff in threads.

```
int main(){
    ShippingContainer st_a(156);
    ShippingContainer st_b(45);

    std::thread t1(transfer, std::ref(st_a),std::ref(st_b), 20);
    std::thread t2(transfer, std::ref(st_a), std::ref(st_b), 11);

    t1.join();
    t2.join();

    std::cout << "shipping container a has " << st_a.things << " things " << std::endl;
    std::cout << "shipping container b has " << st_b.things << " things " << std::endl;

    return 0;
}
```

1. Prefer locking single mutex at a time.
2. Avoid locking a mutex and then calling a user function.
3. Use std::lock() to lock more than one mutex.
4. Lock the mutex in same order for all threads.

Unique_lock and Lazy Initialization

The unique_lock class behaves like our lock_guard, but it adds some additional capabilities. For one, it provides lazy initialization. You can instantiate it but lock it at a later date. You can also lock and unlock it, which you cannot do with the lock_guard, which locks when constructed, and unlocks when destroyed.

```
class LogFile {
    std::mutex m_mu;
    ofstream m_f;
public:
    LogFile() {
        m_f.open("/tmp/log.txt");
    }

    void shared_print(int id, std::string& msg) {
        std::unique_lock<std::mutex> locker(m_mu, std::defer_lock);
        // do something else

        // we can lock at a later date
        locker.lock();
        m_f << msg << " " << id << std::endl;
        locker.unlock();
```

```
    }
};
```

So why use a lock_guard at all? A unique_lock is a bit more expensive. So, if you don't need to use its capabilities, it is better to stick with the lock_guard.

More work

Looking back at our LogFile class, what if we decide that we want to move the file open call into the shared_print method? We will certainly need a lock around opening the file. Otherwise, multiple callers will be able to lock the file. Lets try and make some modifications to support this.

```
class LogFile {
    std::mutex m_mu;
    std::mutex m_mu_open;
    ofstream m_f;
public:
    LogFile() {

    }

    void shared_print(int id, std::string& msg) {
        if(!m_f.is_open()) {
            std::unique_lock<std::mutex> locker_fo(m_mu_open)
            m_f.open("/tmp/log.txt");
        }

        std::unique_lock<std::mutex> locker(m_mu, std::defer_lock);


        // we can lock at a later date
        locker.lock();
        m_f << msg << " " << id << std::endl;
        locker.unlock();

    }
};
```

Great. But... Is this code threadsafe?

No, not really. Since locker_fo is instantiated in the `if block`, it also goes out of scope at the end of the if block. It is possible for two threads to open the same file. What can be done. Well, we can move locker_fo out into the surrounding scope.

```
    void shared_print(int id, std::string& msg)  {
        {
            std::unique_lock<std::mutex> locker_fo(m_mu_open);
            if(!m_f.is_open()) {
                m_f.open("/tmp/log.txt");
            }
        }
        ...
    }
```

Great. Now this is threadsafe. But is it efficient? We are locking around the file open test each time we call shared_print. We are hindering the program from running concurrently. Fortunately, the standard library provides a primitive for just such an occasion... The std::once_flag.

```
class LogFile{
    std::mutex m_mutex;
    std::once_flag m_flag;
    std::ofstream m_f;

public:
    LogFile {}
    void shared_print(int id, std::string& msg) {
        std::call_once(m_flag, [&](){ m_f.open("/tmp/log.txt); });

        std::unique_lock<std::mutex> locker(m_mutex, std::defer_lock);
        locker.lock();
        m_f << msg << " " << id << std::endl;
        std::cout << msg << " " << id << std::endl;
        locker.unlock();
    }
    // clean up
    ~LogFile() {
        if(m_f)
            m_f.close();
    }
};
```

Condition Variables

Up to this point, we have been using the mutex to synchronize access to objects. But we are going to go beyond mutexes today. Lets say we have the following functions:

```
#include <mutex>
#include <iostream>
```

```
#include <string>
#include <deque>
#include <chrono>

std::deque<int> q;
std::mutex pvc_mu;


void pvc_func_1() {
    int count = 10;
    while(count > 0){
        std::unique_lock<std::mutex> locker(pvc_mu);
        q.push_front(count);
        locker.unlock();
        std::this_tread::sleep_for(std::chrono::seconds(1));
        count--;
    }
}

void pvc_func_2() {
    int data = 0;
    while (data != 1) {
        std::unique_lock<std::mutex> locker(pvc_mu);
        if(!q.empty()) {
            data = q.back();
            q.pop_back();
            locker.unlock();
            std::cout << "t goet vlaue from t1: " << data << std::endl;
        } else {
            locker.unlock();
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
}
```

Now lets go ahead and use them.

```
int main() {
    std::thread t1(pvc_func_1);
    std::thread t2(pvc_func_2);

    t1.join();
    t2.join();

    return 0;

};
```

Ok. Well this works. But... It isn't very efficient. The second thread spends a lot of time in the busy waiting state. That is inefficient. It is taking up cycles sleeping, and tuning this can be a pain. Rather than pursue this strategy, C++ provides the `condition variable`. Let's change it to work with condition variables.

```
std::mutex vc_mu;

std::condition_variable cond;

void cv_func_1() {
    int count = 10;
    while(count < 0) {
        std::unique_lock<mutex> locker(cv_mu);
        q.push_front(count);
        locker.unlock();
        cond.notify_one(); // notify one waiting thread if there is one
        std::this_thread::sleep_for(std::chrono::seconds(1) );
        count--;
    }
 }


 void cv_func_2() {
    int data = 0;
    while(data != 1) {
        std::unique_lock<std::mutex> locker(cv_mu);
        cond.wait(locker, [](){ return !q.empty();} );
        data = q.back();
        q.pop_back();
        locker.unlock();

        std::cout << "T2 got a value from t1: " << data << endl;
    }
 }
```

With condition variables, we can make sure that threads are running in a specific order. Note that the wait method takes a predicate in addition to the lock. The predicate guards against "spurious wakes", which may happen, at least in via pthreads (the underlying library). You would think that "spurious wakeups" would be a design flaw, but according to the author (I googled it), they are part of the design. Sounds strange to me, but in any case, enforce your loop invariants via the predicate.

Promises, Futures, and Async

C++11 has introduced a couple of new primitives for concurrency.