https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html

2009/02 ~ 2010/02

Multithreading in C++0x

Part 1: Starting Threads

Part 2: Starting Threads with Function Objects and Arguments

Part 3: Starting Threads with Member Functions and Reference Arguments

Part 4: Protecting Shared Data

Part 5: Flexible locking with std::unique_lock<>

part 6: Lazy initialization and double-checked locking with atomics

part 7: Locking multiple mutexes without deadlock

part 8: Futures, Promises and Asynchronous Function Calls

Multithreading in C++0x part 1: Starting Threads

Tuesday, 10 February 2009

This is the first of a series of blog posts introducing the new C++0x thread library.

Concurrency and multithreading is all about running multiple pieces of code in parallel. If you have the hardware for it in the form of a nice shiny multi-core CPU or a multi-processor system then this code can run truly in parallel, otherwise it is interleaved by the operating system — a bit of one task, then a bit of another. This is all very well, but somehow you have to specify *what* code to run on all these threads.

High level constructs such as the parallel algorithms in Intel's Threading Building Blocks manage the division of code between threads for you, but we don't have any of these in C++0x. Instead, we have to manage the threads ourselves. The tool for this is std::thread.

Running a simple function on another thread

Let's start by running a simple function on another thread, which we do by constructing a new std::thread object, and passing in the function to the constructor. std::thread lives in the <thread> header, so we'd better include that first.

```
#include <thread>

void my_thread_func()
{}

int main()
{
    std::thread t(my_thread_func);
}
```

If you compile and run this little app, it won't do a lot: though it starts a new thread, the thread function is empty. Let's make it do something, such as print "hello":

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

void my_thread_func()
{
	std::cout<<"hello"<<std::endl;
}

int main()
{
	std::thread t(my_thread_func);
}
```

If you compile and run this little application, what happens? Does it print hello like we wanted? Well, actually there's no telling. It might do or it might not. I ran this simple application several times on my machine, and the output was unreliable: sometimes it output "hello", with a newline; sometimes it output "hello" *without* a newline, and sometimes it **didn't output anything**. What's up with that? Surely a simple app like this ought to behave predictably?

Waiting for threads to finish

Well, actually, no, this app does **not** have predictable behaviour. The problem is we're not waiting for our thread to finish. When the execution reaches the end of main() the program is terminated, whatever the other threads are doing. Since thread scheduling is unpredictable, we cannot know how far the other thread has got. It might have finished, it might have output the "hello", but not processed the std::endl yet, or it might not have even started. In any case it will be abruptly stopped as the application exits.

If we want to *reliably* print our message, we have to ensure that our thread has finished. We do that by *joining* with the thread by calling the join() member function of our thread object:

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

void my_thread_func()
{
	std::cout<<"hello"<<std::endl;
}

int main()
{
	std::thread t(my_thread_func);
	t.join();
}
```

Now, main() will wait for the thread to finish before exiting, and the code will output "hello" followed by a newline every time. This highlights a general point: **if you want a thread to have finished by a certain point in your code you have to wait for it**. As well as ensuring that threads have finished by the time the program exits, this is also important if a thread has access to local variables: we want the thread to have finished before the local variables go out of scope.

Next Time

In this article we've looked at running simple functions on a separate thread, and waiting for the thread to finish. However, when you start a thread you aren't just limited to simple functions with no arguments: in the second part of this series we will look at how to start a thread with function objects, and how to pass arguments to the thread.

Multithreading in C++0x part 2: Starting Threads with Function Objects and Arguments

Tuesday, 17 February 2009

This is the second of a series of blog posts introducing the new C++0x thread library. If you missed the first part, it covered Starting Threads in C++0x with simple functions.

If you read part 1 of this series, then you've seen how easy it is to start a thread in C++0x: just construct an instance of std::thread, passing in the function you wish to run on the new thread. Though this is good, it would be quite limiting if new threads were constrained to run plain functions without any arguments — all the information needed would have to be passed via global variables, which would be incredibly messy. Thankfully, this is not the case. Not only can you run function objects on your new thread, as well as plain functions, but you can pass arguments in too.

Running a function object on another thread

In keeping with the rest of the C++ standard library, you're not limited to plain functions when starting threads — the std::thread constructor can also be called with instances of classes that implement the function-call operator. Let's say "hello" from our new thread using a function object:

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

class SayHello
{
public:
	void operator()() const
	{
		std::cout<<"hello"<<std::endl;
```

```
    }
};

int main()
{
    std::thread t((SayHello()));
    t.join();
}
```

If you're wondering about the extra parentheses around the SayHello constructor call, this is to avoid what's known as *C++'s most vexing parse*: without the parentheses, the declaration is taken to be a declaration of a *function called twhich takes a pointer-to-a-function-with-no-parameters-returning-an-instance-of-SayHello, and which returns a*[*std::thread*] *object*, rather than an object called t of type [std::thread]. There are a few other ways to avoid the problem. Firstly, you could create a named variable of type SayHello and pass that to the [std::thread] constructor:

```
int main()
{
    SayHello hello;
    std::thread t(hello);
    t.join();
}
```

Alternatively, you could use copy initialization:

```
int main()
{
    std::thread t=std::thread(SayHello());
    t.join();
}
```

And finally, if you're using a full C++0x compiler then you can use the new initialization syntax with braces instead of parentheses:

```
int main()
{
    std::thread t{SayHello()};
    t.join();
}
```

In this case, this is exactly equivalent to our first example with the double parentheses.

Anyway, enough about initialization. Whichever option you use, the idea is the same: your function object is copied into internal storage accessible to the new thread, and the new thread invokes your operator(). Your class can of course have data members and other member functions too, and this is one way of passing data to the thread function: pass it in as a constructor argument and store it as a data member:

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

```
class Greeting
{
	std::string message;
public:
	explicit Greeting(std::string const& message_):
		message(message_)
	{}
	void operator()() const
	{
		std::cout<<message<<std::endl;
	}
};

int main()
{
	std::thread t(Greeting("goodbye"));
	t.join();
}
```

In this example, our message is stored as a data member in the class, so when the Greeting instance is copied into the thread the message is copied too, and this example will print "goodbye" rather than "hello".

This example also demonstrates one way of passing information in to the new thread aside from the function to call — include it as data members of the function object. If this makes sense in terms of the function object then it's ideal, otherwise we need an alternate technique.

Passing Arguments to a Thread Function

As we've just seen, one way to pass arguments in to the thread function is to package them in a class with a function call operator. Well, there's no need to write a special class every time; the standard library provides an easy way to do this in the form of std::bind. The std::bind function template takes a variable number of parameters. The first is always the function or callable object which needs the parameters, and the remainders are the parameters to pass when calling the function. The result is a function object that stores copies of the supplied arguments, with a function call operator that invokes the bound function. We could therefore use this to pass the message to write to our new thread:

```
#include <thread>
#include <iostream>
#include <string>
#include <functional>

void greeting(std::string const& message)
{
	std::cout<<message<<std::endl;
}

int main()
```

```
{
	std::thread t(std::bind(greeting,"hi!"));
	t.join();
}
```

This works well, but we can actually do better than that — we can pass the arguments directly to the std::thread constructor and they will be copied into the internal storage for the new thread and supplied to the thread function. We can thus write the preceding example more simply as:

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

void greeting(std::string const& message)
{
	std::cout<<message<<std::endl;
}

int main()
{
	std::thread t(greeting,"hi!");
	t.join();
}
```

Not only is this code simpler, it's also likely to be more efficient as the supplied arguments can be copied directly into the internal storage for the thread rather than first into the object generated by std::bind, which is then in turn copied into the internal storage for the thread.

Multiple arguments can be supplied just by passing further arguments to the std::thread constructor:

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

void write_sum(int x,int y)
{
	std::cout<<x<<" + "<<y<<" = "<<(x+y)<<std::endl;
}

int main()
{
	std::thread t(write_sum,123,456);
	t.join();
}
```

The std::thread constructor is a variadic template, so it can take any number of arguments up to the compiler's internal limit, but if you need to pass more than a couple of parameters to your thread function then you might like to rethink your design.

Next time

We're not done with starting threads just yet — there are a few more nuances to passing arguments which we haven't covered. In the third part of this series we'll look at passing references, and using class member functions as the thread function.

Multithreading in C++0x part 3: Starting Threads with Member Functions and Reference Arguments

Thursday, 26 February 2009

This is the third of a series of blog posts introducing the new C++0x thread library. The first two parts covered Starting Threads in C++0x with simple functions, and starting threads with function objects and additional arguments.

If you've read the previous parts of the series then you've seen how to start threads with functions and function objects, with and without additional arguments. However, the function objects and arguments are always copied into the thread's internal storage. What if you wish to run a member function other than the function call operator, or pass a reference to an existing object?

The C++0x library can handle both these cases: the use of member functions with std::thread requires an additional argument for the object on which to invoke the member function, and references are handled with std::ref. Let's take a look at some examples.

Invoking a member function on a new thread

Starting a new thread which runs a member function of an existing object: you just pass a pointer to the member function and a value to use as the this pointer for the object in to the std::thread constructor.

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

class SayHello
{
public:
	void greeting() const
	{
		std::cout<<"hello"<<std::endl;
	}
};

int main()
{
	SayHello x;
	std::thread t(&SayHello::greeting,&x);
	t.join();
}
```

You can of course pass additional arguments to the member function too:

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

class SayHello
{
public:
	void greeting(std::string const& message) const
	{
		std::cout<<message<<std::endl;
	}
};

int main()
{
	SayHello x;
	std::thread t(&SayHello::greeting,&x,"goodbye");
	t.join();
}
```

Now, the preceding examples both a plain pointer to a local object for the *this* argument; if you're going to do that, you need to ensure that the object outlives the thread, otherwise there will be trouble. An alternative is to use a heap-allocated object and a reference-counted pointer such as std::shared_ptr<SayHello> to ensure that the object stays around as long as the thread does:

```
#include <>

int main()
{
	std::shared_ptr<SayHello> p(new SayHello);
	std::thread t(&SayHello::greeting,p,"goodbye");
	t.join();
}
```

So far, everything we've looked at has involved copying the arguments and thread functions into the internal storage of a thread even if those arguments are pointers, as in the this pointers for the member functions. What if you want to pass in a *reference* to an existing object, and a pointer just won't do? That is the task of std::ref.

Passing function objects and arguments to a thread by reference

Suppose you have an object that implements the function call operator, and you wish to invoke it on a new thread. The thing is you want to invoke the function call operator on *this particular object* rather than copying it. You could use the member function support to call operator() explicitly, but that seems a bit of a mess given that it *is* callable already. This is the first instance in which std::ref can help — if x is a callable object, then std::ref(x) is too, so we can pass std::ref(x) as our function when we start the thread, as below:

```
#include <thread>
```

```
#include <iostream>
#include <functional> // for std::ref

class PrintThis
{
public:
	void operator()() const
	{
		std::cout<<"this="<<this<<std::endl;
	}
};

int main()
{
	PrintThis x;
	x();
	std::thread t(std::ref(x));
	t.join();
	std::thread t2(x);
	t2.join();
}
```

In this case, the function call operator just prints the address of the object. The exact form and values of the output will vary, but the principle is the same: this little program should output three lines. The first two should be the same, whilst the third is different, as it invokes the function call operator on a *copy* of x. For one run on my system it printed the following:

```
this=0x7fffb08bf7ef
this=0x7fffb08bf7ef
this=0x42674098
```

Of course, std::ref can be used for other arguments too — the following code will print "x=43":

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

void increment(int& i)
{
	++i;
}

int main()
{
	int x=42;
	std::thread t(increment,std::ref(x));
	t.join();
	std::cout<<"x="<<x<<std::endl;
}
```

When passing in references like this (or pointers for that matter), you need to be careful not only that the referenced object outlives the thread, but also that appropriate synchronization is used. In this case

it is fine, because we only access x before we start the thread and after it is done, but concurrent access would need protection with a mutex.

Next time

That wraps up all the variations on starting threads; next time we'll look at using mutexes to protect data from concurrent modification.

Multithreading in C++0x part 4: Protecting Shared Data

Saturday, 04 April 2009

This is the fourth of a series of blog posts introducing the new C++0x thread library. The first three parts covered starting threads in C++0x with simple functions, starting threads with function objects and additional arguments, and starting threads with member functions and reference arguments.

If you've read the previous parts of the series then you should be comfortable with starting threads to perform tasks "in the background", and waiting for them to finish. You can accomplish a lot of useful work like this, passing in the data to be accessed as parameters to the thread function, and then retrieving the result when the thread has completed. However, this won't do if you need to communicate between the threads whilst they are running — accessing shared memory concurrently from multiple threads causes undefined behavior if either thread modifies the data. What you need here is some way of ensuring that the accesses are *mutually exclusive*, so only one thread can access the shared data at a time.

Mutual Exclusion with std::mutex

Mutexes are conceptually simple. A mutex is either "locked" or "unlocked", and threads try and lock the mutex when they wish to access some protected data. If the mutex is already locked then any other threads that try and lock the mutex will have to wait. Once the thread is done with the protected data it unlocks the mutex, and another thread can lock the mutex. If you make sure that threads always lock a particular mutex before accessing a particular piece of shared data then other threads are excluded from accessing the data until as long as another thread has locked the mutex. This prevents concurrent access from multiple threads, and avoids the undefined behavior of data races. The simplest mutex provided by C++0x is std::mutex.

Now, whilst std::mutex has member functions for explicitly locking and unlocking, by far the most common use case in C++ is where the mutex needs to be locked for a specific region of code. This is where the std::lock_guard<> template comes in handy by providing for exactly this scenario. The constructor locks the mutex, and the destructor unlocks the mutex, so to lock a mutex for the duration of a block of code, just construct a std::lock_guard<> object as a local variable at the start of the block.

For example, to protect a shared counter you can use std::lock_guard<> to ensure that the mutex is locked for either an increment or a query operation, as in the following example:

```
std::mutex m;
unsigned counter=0;

unsigned increment()
{
	std::lock_guard<std::mutex> lk(m);
	return ++counter;
}
unsigned query()
{
	std::lock_guard<std::mutex> lk(m);
	return counter;
}
```

This ensures that access to counter is *serialized* — if more than one thread calls query() concurrently then all but one will block until the first has exited the function, and the remaining threads will then have to take turns. Likewise, if more than one thread calls increment() concurrently then all but one will block. Since both functions lock the *same* mutex, if one thread calls query() and another calls increment() at the same time then one or other will have to block. This *mutual exclusion* is the whole point of a mutex.

Exception Safety and Mutexes

Using std::lock_guard<> to lock the mutex has additional benefits over manually locking and unlocking when it comes to exception safety. With manual locking, you have to ensure that the mutex is unlocked correctly on every exit path from the region where you need the mutex locked, *including when the region exits due to an exception*. Suppose for a moment that instead of protecting access to a simple integer counter we were protecting access to a std::string, and appending parts on the end. Appending to a string might have to allocate memory, and thus might throw an exception if the memory cannot be allocated. With std::lock_guard<> this still isn't a problem — if an exception is thrown, the mutex is still unlocked. To get the same behaviour with manual locking we have to use a catch block, as shown below:

```
std::mutex m;
std::string s;

void append_with_lock_guard(std::string const& extra)
{
	std::lock_guard<std::mutex> lk(m);
	s+=extra;
}

void append_with_manual_lock(std::string const& extra)
{
	m.lock();
	try
```

```
    {
        s+=extra;
        m.unlock();
    }
    catch(...)
    {
        m.unlock();
        throw;
    }
}
```

If you had to do this for every function which might throw an exception it would quickly get unwieldy. Of course, you still need to ensure that the code is exception-safe in general — it's no use automatically unlocking the mutex if the protected data is left in a state of disarray.

Next time

Next time we'll take a look at the std::unique_lock<> template, which provides more options thanstd::lock_guard<>.

Multithreading in C++0x part 5: Flexible locking with std::unique_lock<>

Wednesday, 15 July 2009

This is the fifth in a series of blog posts introducing the new C++0x thread library. So far we've looked at the various ways of starting threads in C++0x and protecting shared data with mutexes. See the end of this article for a full set of links to the rest of the series.

In the previous installment we looked at the use of std::lock_guard<> to simplify the locking and unlocking of a mutex and provide exception safety. This time we're going to look at the std::lock_guard<>'s companion class template std::unique_lock<>. At the most basic level you use it like std::lock_guard<> — pass a mutex to the constructor to acquire a lock, and the mutex is unlocked in the destructor — but if that's all you're doing then you really ought to use std::lock_guard<> instead. The benefit to using std::unique_lock<> comes from two things:

1. you can transfer ownership of the lock between instances, and
2. the std::unique_lock<> object does not have to own the lock on the mutex it is associated with.

Let's take a look at each of these in turn, starting with transferring ownership.

Transferring ownership of a mutex lock

between std::unique_lock<> instances

There are several consequences to being able to transfer ownership of a mutex lock between std::unique_lock<> instances: you can return a lock from a function, you can store locks in standard containers, and so forth.

For example, you can write a simple function that acquires a lock on an internal mutex:

```
std::unique_lock<std::mutex> acquire_lock()
{
	static std::mutex m;
	return std::unique_lock<std::mutex>(m);
}
```

The ability to transfer lock ownership between instances also provides an easy way to write classes that are themselves movable, but hold a lock internally, such as the following:

```
class data_to_protect
{
public:
	void some_operation();
	void other_operation();
};

class data_handle
{
private:
	data_to_protect* ptr;
	std::unique_lock<std::mutex> lk;

	friend data_handle lock_data();

	data_handle(data_to_protect* ptr_,std::unique_lock<std::mutex> lk_):
		ptr(ptr_),lk(lk_)
	{}
public:
	data_handle(data_handle && other):
		ptr(other.ptr),lk(std::move(other.lk))
	{}
	data_handle& operator=(data_handle && other)
	{
		if(&other != this)
		{
			ptr=other.ptr;
			lk=std::move(other.lk);
			other.ptr=0;
		}
		return *this;
	}
```

```
        void do_op()
        {
                ptr->some_operation();
        }
        void do_other_op()
        {
                ptr->other_operation();
        }
};

data_handle lock_data()
{
        static std::mutex m;
        static data_to_protect the_data;
        std::unique_lock<std::mutex> lk(m);
        return data_handle(&the_data,std::move(lk));
}

int main()
{
        data_handle dh=lock_data(); // lock acquired
        dh.do_op();                 // lock still held
        dh.do_other_op();           // lock still held
        data_handle dh2;
        dh2=std::move(dh);          // transfer lock to other handle
        dh2.do_op();                // lock still held
}                                   // lock released
```

In this case, the function lock_data() acquires a lock on the mutex used to protect the data, and then transfers that along with a pointer to the data into the data_handle. This lock is then held by the data_handle until the handle is destroyed, allowing multiple operations to be done on the data without the lock being released. Because the std::unique_lock<> is movable, it is easy to make data_handle movable too, which is necessary to return it from lock_data.

Though the ability to transfer ownership between instances is useful, it is by no means as useful as the simple ability to be able to manage the ownership of the lock separately from the lifetime of the std::unique_lock<> instance.

Explicit locking and unlocking a mutex with a std::unique_lock<>

As we saw in part 4 of this series, std::lock_guard<> is very strict on lock ownership — it owns the lock from construction to destruction, with no room for manoeuvre. std::unique_lock<> is rather lax in comparison. As well as acquiring a lock in the constructor as for std::lock_guard<>, you can:

- construct an instance without an associated mutex at all (with the default constructor);
- construct an instance with an associated mutex, but leave the mutex unlocked (with the deferred-locking constructor);

- construct an instance that *tries* to lock a mutex, but leaves it unlocked if the lock failed (with the try-lock constructor);
- if you have a mutex that supports locking with a timeout (such as std::timed_mutex) then you can construct an instance that tries to acquire a lock for either a specified time period or until a specified point in time, and leaves the mutex unlocked if the timeout is reached;
- lock the associated mutex if the std::unique_lock<> instance doesn't currently own the lock (with the lock() member function);
- try and acquire lock the associated mutex if the std::unique_lock<> instance doesn't currently own the lock (possibly with a timeout, if the mutex supports it) (with the try_lock(), try_lock_for() and try_lock_until() member functions);
- unlock the associated mutex if the std::unique_lock<> *does* currently own the lock (with the unlock() member function);
- check whether the instance owns the lock (by calling the owns_lock() member function;
- release the association of the instance with the mutex, leaving the mutex in whatever state it is currently (locked or unlocked) (with the release() member function); and
- transfer ownership between instances, as described above.

As you can see, std::unique_lock<> is quite flexible: it gives you complete control over the underlying mutex, and actually meets all the requirements for a *Lockable* object itself. You can thus have astd::unique_lock<std::unique_lock<std::mutex>> if you really want to! However, even with all this flexibility it still gives you exception safety: if the lock is held when the object is destroyed, it is released in the destructor.

std::unique_lock<> and condition variables

One place where the flexibility of std::unique_lock<> is used is with std::condition_variable.std::condition_variable provides an implementation of a *condition variable*, which allows a thread to wait until it has been notified that a certain condition is true. When waiting you must pass in a std::unique_lock<> instance that owns a lock on the mutex protecting the data related to the condition. The condition variable uses the flexibility of std::unique_lock<> to unlock the mutex whilst it is waiting, and then lock it again before returning to the caller. This enables other threads to access the protected data whilst the thread is blocked. I will expand upon this in a later part of the series.

Other uses for flexible locking

The key benefit of the flexible locking is that the lifetime of the lock object is independent from the time over which the lock is held. This means that you can unlock the mutex before the end of a function is reached if certain conditions are met, or unlock it whilst a time-consuming operation is performed (such as waiting on a condition variable as described above) and then lock the mutex again once the time-consuming operation is complete. Both these choices are embodiments of the common advice to hold a lock for the minimum length of time possible without sacrificing exception safety when the

lock *is* held, and without having to write convoluted code to get the lifetime of the lock object to match the time for which the lock is required.

For example, in the following code snippet the mutex is unlocked across the time-consuming load_strings() operation, even though it must be held either side to access the strings_to_process variable:

```
std::mutex m;
std::vector<std::string> strings_to_process;

void update_strings()
{
	std::unique_lock<std::mutex> lk(m);
	if(strings_to_process.empty())
	{
		lk.unlock();
		std::vector<std::string> local_strings=load_strings();
		lk.lock();
		strings_to_process.insert(strings_to_process.end(),
                                          local_strings.begin(),local_strings.end());
	}
}
```

Next time

Next time we'll look at the use of the new std::lock() and std::try_lock()function templates to avoid deadlock when acquiring locks on multiple mutexes.

Multithreading in C++0x part 6: Lazy initialization and double-checked locking with atomics

Thursday, 13 August 2009

This is the sixth in a series of blog posts introducing the new C++0x thread library. So far we've looked at the various ways of starting threads in C++0x and protecting shared data with mutexes. See the end of this article for a full set of links to the rest of the series.

I had intended to write about the use of the new std::lock() for avoiding deadlock in this article. However, there was a post on comp.std.c++ this morning about lazy initialization, and I thought I'd write about that instead.std::lock() can wait until next time.

Lazy Initialization

The classic lazy-initialization case is where you have an object that is expensive to construct but isn't always needed. In this case you can choose to only initialize it on first use:

```
class lazy_init
{
	mutable std::unique_ptr<expensive_data> data;
public:
	expensive_data const& get_data() const
	{
		if(!data)
		{
			data.reset(new expensive_data);
		}
		return *data;
	}
};
```

However, we can't use this idiom in multi-threaded code, since there would be a data race on the accesses to data. Enter std::call_once() — by using an instance of std::once_flag to protect the initialization we can make the data race go away:

```
class lazy_init
{
	mutable std::once_flag flag;
	mutable std::unique_ptr<expensive_data> data;

	void do_init() const
	{
		data.reset(new expensive_data);
	}
public:
	expensive_data const& get_data() const
	{
		std::call_once(flag,&lazy_init::do_init,this);
		return *data;
	}
};
```

Concurrent calls to get_data() are now safe: if the data has already been initialized they can just proceed concurrently. If not, then all threads calling concurrently except one will wait until the remaining thread has completed the initialization.

Reinitialization

This is all very well if you only want to initialize the data *once*. However, what if you need to update the data — perhaps it's a *cache* of some rarely-changing data that's expensive to come by. std::call_once() doesn't support multiple calls (hence the name). You could of course protect the data with a mutex, as shown below:

```
class lazy_init_with_cache
{
	mutable std::mutex m;
	mutable std::shared_ptr<const expensive_data> data;
```

```
public:
    std::shared_ptr<const expensive_data> get_data() const
    {
        std::lock_guard<std::mutex> lk(m);
        if(!data)
        {
            data.reset(new expensive_data);
        }
        return data;
    }
    void invalidate_cache()
    {
        std::lock_guard<std::mutex> lk(m);
        data.reset();
    }
};
```

Note that in this case we return a std::shared_ptr<const expensive_data> rather than a reference to avoid a race condition on the data itself — this ensures that the copy held by the calling code will be valid (if out of date) even if another thread calls invalidate_cache() before the data can be used.

This "works" in the sense that it avoids data races, but if the updates are rare and the reads are frequent then this may cause unnecessary serialization when multiple threads call get_data() concurrently. What other options do we have?

Double-checked locking returns

Much has been written about how double-checked locking is broken when using multiple threads. However, the chief cause of the problem is that the sample code uses plain non-atomic operations to check the flag outside the mutex, so is subject to a data race. You can overcome this by careful use of the C++0x atomics, as shown in the example below:

```
class lazy_init_with_cache
{
    mutable std::mutex m;
    mutable std::shared_ptr<const expensive_data> data;

public:
    std::shared_ptr<const expensive_data> get_data() const
    {
        std::shared_ptr<const expensive_data> result=
            std::atomic_load_explicit(&data,std::memory_order_acquire);
        if(!result)
        {
            std::lock_guard<std::mutex> lk(m);
            result=data;
            if(!result)
            {
                result.reset(new expensive_data);
                std::atomic_store_explicit(&data,result,std::memory_order_release);
```

```
                }
            }
            return result;
        }
        void invalidate_cache()
        {
            std::lock_guard<std::mutex> lk(m);
            std::shared_ptr<const expensive_data> dummy;
            std::atomic_store_explicit(&data,dummy,std::memory_order_relaxed);
        }
};
```

Note that in this case, *all writes to data use atomic operations, even those within the mutex lock*. This is necessary in order to ensure that the atomic load operation at the start of get_data() actually has a coherent value to read — there's no point doing an atomic load if the stores are not atomic, otherwise you might atomically load some half-written data. Also, the atomic load and store operations ensure that the reference count on the std::shared_ptr object is correctly updated, so that the expensive_data object is correctly destroyed when the last std::shared_ptr object referencing it is destroyed.

If our atomic load actually returned a non-NULL value then we can use that, just as we did before. However, if it returned NULL then we need to lock the mutex and try again. This time we can use a plain read of data, since the mutex is locked. If we still get NULL then we need to do the initialization. However, we can't just call data.reset() like before, since that is not atomic. Instead we must create a local std::shared_ptr instance with the value we want, and store the value with an atomic store operation. We can use result for the local value, since we want the value in that variable anyway.

In invalidate_cache() we must also store the value using std::atomic_store_explicit(), in order to ensure that the NULL value is correctly read in get_data(). Note also that we must also lock the mutex here, in order to avoid a data race with the initialization code inside the mutex lock in get_data().

Memory ordering

By using std::atomic_load_explicit() and std::atomic_store_explicit() we can specify the memory ordering requirements of the operations. We could have just used std::atomic_load() and std::atomic_store(), but those would have implied std::memory_order_seq_cst, which is overkill in this scenario. What we need is to ensure that if a non-NULL value is read in get_data() then the actual creation of the associated object *happens-before* the read. The store in get_data() must therefore use std::memory_order_release, whilst the load usesstd::memory_order_acquire.

On the other hand, the store in invalidate_cache() can merrily use std::memory_order_relaxed, since there is no data associated with the store: if the load in get_data() reads NULL then the mutex will be locked, which will handle any necessary synchronization.

Whenever you use atomic operations, you have to make sure that the memory ordering is correct, and that there are no races. Even in such a simple case such as this it is not trivial, and I would not recommend it unless profiling has shown that this is really a problem.

Update: As if to highlight my previous point about the trickiness of atomic operations, Dmitriy correctly points out in the comments that the use of std::shared_ptr to access the expensive_data implies a reference count, which is a real performance suck in multithreaded code. Whatever the memory ordering constraints we put on it, every thread doing the reading has to update the reference count. This is thus a source of contention, and can seriously limit scalability even if it doesn't force full serialization. The same issues apply to multiple-reader single-writer mutexes — Joe Duffy has written about them over on his blog. Time it on your platform with just a mutex (i.e. no double-checked locking), and with the atomic operations, and use whichever is faster. Alternatively, use a memory reclamation scheme specially tailored for your usage.

Next time

In the next part of this series I'll cover the use of std::lock() that I was intending to cover in this installment.

Multithreading in C++0x part 7: Locking multiple mutexes without deadlock

Friday, 21 August 2009

This is the seventh in a series of blog posts introducing the new C++0x thread library. So far we've looked at the various ways of starting threads in C++0x and protecting shared data with mutexes. See the end of this article for a full set of links to the rest of the series.

After last time's detour into lazy initialization, our regular programming resumes with the promised article on std::lock().

Acquiring locks on multiple mutexes

In most cases, your code should only ever hold a lock on one mutex at a time. Occasionally, you might nest your locks, e.g. by calling into a subsystem that protects its internal data with a mutex whilst holding a lock on another mutex, but it's generally better to avoid holding locks on multiple mutexes at once if at all possible.

However, sometimes it is *necessary* to hold a lock on more than one mutex, because you need to perform an operation on two distinct items of data, each of which is protected by its own mutex. In order to perform the operation correctly, you need to hold locks on both the mutexes, otherwise another thread could change one of the values before your operation was complete. For example, consider a bank transfer between two accounts: we want the transfer to be a single action to the outside world, so we need to acquire the lock on the mutex protecting each account. You could naively implement this like so:

```
class account
{
    std::mutex m;
    currency_value balance;
public:

    friend void transfer(account& from,account& to,
                         currency_value amount)
    {
        std::lock_guard<std::mutex> lock_from(from.m);
        std::lock_guard<std::mutex> lock_to(to.m);
        from.balance -= amount;
        to.balance += amount;
    }
};
```

Though it looks right at first glance (both locks are acquired before data is accessed), there is a potential for deadlock in this code. Consider two threads calling transfer() at the same time on the same accounts, but one thread is transferring money from account A to account B, whereas the other is transferring

money from account B to account A. If the calls to transfer() are running concurrently, then both threads will try and lock the mutex on their from account. Assuming there's nothing else happening in the system, this will succeed, as for thread 1 the from account is account A, whereas for thread 2 the from account is account B. Now each thread tries to acquire the lock on the mutex for the corresponding to. Thread 1 will try and lock the mutex for account B (thread 1 is transferring from A to B). Unfortunately, it will block because thread 2 holds that lock. Meanwhile thread B will try and lock the mutex for account A (thread 2 is transferring from B to A). Thread 1 holds this mutex, so thread 2 must also block — **deadlock**.

Avoiding deadlock with std::lock()

In order to avoid this problem, you need to somehow tell the system to lock the two mutexes together, so that one of the threads acquires both locks and deadlock is avoided. This is what the std::lock() function is for — you supply a number of mutexes (it's a variadic template) and the thread library ensures that when the function returns they are all locked. Thus we can avoid the race condition in transfer() by writing it as follows:

```
void transfer(account& from,account& to,
                    currency_value amount)
{
    std::lock(from.m,to.m);
    std::lock_guard<std::mutex> lock_from(from.m,std::adopt_lock);
    std::lock_guard<std::mutex> lock_to(to.m,std::adopt_lock);
    from.balance -= amount;
    to.balance += amount;
}
```

Here we use std::lock() to lock both mutexes safely, then adopt the ownership into the std::lock_guard instances to ensure the locks are released safely at the end of the function.

Other mutex types

As mentioned already, std::lock() is a function template rather than a plain function. Not only does this mean it can merrily accept any number of mutex arguments, but it also means that it can accept any *type* of mutex arguments. The arguments don't even all have to be the same type. You can pass anything which implements lock(), try_lock()and unlock() member functions with appropriate semantics. As you may remember from part 5 of this series, std::unique_lock<> provides these member functions, so you can pass an instance of std::unique_lock<> to std::lock(). Indeed, you could also write transfer() using std::unique_lock<> like this:

```
void transfer(account& from,account& to,
                    currency_value amount)
{
    std::unique_lock<std::mutex> lock_from(from.m,std::defer_lock);
    std::unique_lock<std::mutex> lock_to(to.m,std::defer_lock);
    std::lock(lock_from,lock_to);
```

```
    from.balance -= amount;
    to.balance += amount;
}
```

In this case, we construct the std::unique_lock<> instances without actually locking the mutexes, and then lock them both together with std::lock() afterwards. You still get all the benefits of the deadlock avoidance, and the same level of exception safety — which approach to use is up to you, and depends on what else is happening in the code.

Exception safety

Since std::lock() has to work with any mutex type you might throw at it, including user-defined ones, it has to be able to cope with exceptions. In particular, it has to provide sensible behaviour if a call to lock() or try_lock() on one of the supplied mutexes throws an exception. The way it does this is quite simple: if a call to lock() ortry_lock() on one of the supplied mutexes throws an exception then unlock() is called on each of the mutexes for which this call to std::lock() currently owns a lock. So, if you are locking 4 mutexes and the call has successfully acquired 2 of them, but a call to try_lock() on the third throws an exception then the locks on the first two will be released by calls to unlock().

The upshot of this is that if std::lock() completes successfully then the calling thread owns the lock on all the supplied mutexes, but if the call exits with an exception then from the point of view of lock ownership it will be as-if the call was never made — any additional locks acquired have been released again.

No silver bullet

There are many ways to write code that deadlocks: std::lock() only addresses the particular case of acquiring multiple mutexes together. However, if you need to do this then std::lock() ensures that you don't need to worry about lock ordering issues.

If your code acquires locks on multiple mutexes, but std::lock() isn't applicable for your case, then you need to take care of lock ordering issues another way. One possibility is to enforce an ordering by using a hierarchical mutex. If you've got a deadlock in your code, then things like the deadlock detection mode of my just::thread library can help you pinpoint the cause.

Next time

In the next installment we'll take a look at the "futures" mechanism from C++0x. Futures are a high level mechanism for passing a value between threads, and allow a thread to wait for a result to be available without having to manage the locks directly.

Multithreading in C++0x part 8: Futures, Promises and Asynchronous Function Calls

Thursday, 11 February 2010

This is the eighth in a series of blog posts introducing the new C++0x thread library. See the end of this article for a full set of links to the rest of the series.

In this installment we'll take a look at the "futures" mechanism from C++0x. Futures are a high level mechanism for passing a value between threads, and allow a thread to wait for a result to be available without having to manage the locks directly.

Futures and asynchronous function calls

The most basic use of a future is to hold the result of a call to the new std::async function for running some code asynchronously:

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
	std::future<int> the_answer=std::async(calculate_the_answer_to_LtUaE);
	do_stuff();
	std::cout<<"The answer to life, the universe and everything is "
		<<the_answer.get()<<std::endl;
}
```

The call to std::async takes care of creating a thread, and invoking calculate_the_answer_to_LtUaE on that thread. The main thread can then get on with calling do_stuff() whilst the immensely time consuming process of calculating the ultimate answer is done in the background. Finally, the call to the get() member function of the std::future<int> object then waits for the function to complete and ensures that the necessary synchronization is applied to transfer the value over so the main thread can print "42".

Sometimes asynchronous functions aren't really asynchronous

Though I said that std::async takes care of creating a thread, that's not necessarily true. As well as the function being called, std::async takes a *launch policy* which specifies whether to start a new thread or create a "deferred function" which is only run when you wait for it. The default launch policy for std::async is std::launch::any, which means that the implementation gets to choose for you. If you

really want to ensure that your function is run on its own thread then you need to specify the std::launch::async policy:

```
std::future<int> the_answer=std::async(std::launch::async,calculate_the_answer_to_LtUaE);
```

Likewise, if you really want the function to be executed in the get() call then you can specify the std::launch::sync policy:

```
std::future<int> the_answer=std::async(std::launch::sync,calculate_the_answer_to_LtUaE);
```

In most cases it makes sense to let the library choose. That way you'll avoid creating too many threads and overloading the machine, whilst taking advantage of the available hardware threads. If you need fine control, you're probably better off managing your own threads.

Divide and Conquer

std::async can be used to easily parallelize simple algorithms. For example, you can write a parallel version offor_each as follows:

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    ptrdiff_t const range_length=last-first;
    if(!range_length)
        return;
    if(range_length==1)
    {
        f(*first);
        return;
    }

    Iterator const mid=first+(range_length/2);

    std::future<void> bgtask=std::async(&parallel_for_each<Iterator,Func>,
                                        first,mid,f);
    try
    {
        parallel_for_each(mid,last,f);
    }
    catch(...)
    {
        bgtask.wait();
        throw;
    }
    bgtask.get();
}
```

This simple bit of code recursively divides up the range into smaller and smaller pieces. Obviously an empty range doesn't require anything to happen, and a single-point range just requires calling f on the one and only value. For bigger ranges then an asynchronous task is spawned to handle the first half, and then the second half is handled by a recursive call.

The try - catch block just ensures that the asynchronous task is finished before we leave the function even if an exception in order to avoid the background tasks potentially accessing the range after it has been destroyed. Finally, the get() call waits for the background task, *and* propagates any exception thrown from the background task. That way if an exception is thrown during any of the processing then the calling code will see an exception. Of course if more than one exception is thrown then some will get swallowed, but C++ can only handle one exception at a time, so that's the best that can be done without using a custom composite_exception class to collect them all.

Many algorithms can be readily parallelized this way, though you may want to have more than one element as the minimum range in order to avoid the overhead of spawning the asynchronous tasks.

Promises

An alternative to using std::async to spawn the task and return the future is to manage the threads yourself and use the std::promise class template to provide the future. Promises provide a basic mechanism for transferring values between threads: each std::promise object is associated with a single std::future object. A thread with access to the std::future object can use wait for the result to be set, whilst another thread that has access to the corresponding std::promise object can call set_value() to store the value and make the future *ready*. This works well if the thread has more than one task to do, as information can be made ready to other threads as it becomes available rather than all of them having to wait until the thread doing the work has completed. It also allows for situations where multiple threads could produce the answer: from the point of view of the waiting thread it doesn't matter where the answer came from, just that it is there so it makes sense to have a single future to represent that availability.

For example, asynchronous I/O could be modeled on a promise/future basis: when you submit an I/O request then the async I/O handler creates a promise/future pair. The future is returned to the caller, which can then wait on the future when it needs the data, and the promise is stored alongside the details of the request. When the request has been fulfilled then the I/O thread can set the value on the promise to pass the value back to the waiting thread before moving on to process additional requests. The following code shows a sample implementation of this pattern.

```
class aio
{
    class io_request
    {
        std::streambuf* is;
        unsigned read_count;
        std::promise<std::vector<char> > p;
    public:
        explicit io_request(std::streambuf& is_,unsigned count_):
            is(&is_),read_count(count_)
        {}

        io_request(io_request&& other):
            is(other.is),
```

```
            read_count(other.read_count),
            p(std::move(other.p))
        {}

        io_request():
            is(0),read_count(0)
        {}

        std::future<std::vector<char> > get_future()
        {
            return p.get_future();
        }

        void process()
        {
            try
            {
                std::vector<char> buffer(read_count);

                unsigned amount_read=0;
                while((amount_read != read_count) &&
                        (is->sgetc()!=std::char_traits<char>::eof()))
                {
                    amount_read+=is->sgetn(&buffer[amount_read],read_count-amount_read);
                }

                buffer.resize(amount_read);

                p.set_value(std::move(buffer));
            }
            catch(...)
            {
                p.set_exception(std::current_exception());
            }
        }
    };

    thread_safe_queue<io_request> request_queue;
    std::atomic_bool done;

    void io_thread()
    {
        while(!done)
        {
            io_request req=request_queue.pop();
            req.process();
        }
    }

    std::thread iot;

public:
```

```
    aio():
        done(false),
        iot(&aio::io_thread,this)
    {}

    std::future<std::vector<char> > queue_read(std::streambuf& is,unsigned count)
    {
        io_request req(is,count);
        std::future<std::vector<char> > f(req.get_future());
        request_queue.push(std::move(req));
        return f;
    }

    ~aio()
    {
        done=true;
        request_queue.push(io_request());
        iot.join();
    }
};

void do_stuff()
{}

void process_data(std::vector<char> v)
{
    for(unsigned i=0;i<v.size();++i)
    {
        std::cout<<v[i];
    }
    std::cout<<std::endl;
}

int main()
{
    aio async_io;

    std::filebuf f;
    f.open("my_file.dat",std::ios::in | std::ios::binary);

    std::future<std::vector<char> > fv=async_io.queue_read(f,1048576);

    do_stuff();
    process_data(fv.get());

    return 0;
}
```

Next Time

The sample code above also demonstrates passing exceptions between threads using the set_exception() member function of std::promise. I'll go into more detail about exceptions in multithreaded next time.