Designing Multithreaded Programs in C++0x

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

23rd April 2009

Designing Multithreaded Programs in C++0x

- Why multithreading is hard
- Overview of the C++0x tools to help
- Examples
- Testing and designing concurrent code

Multithreading is Hard

- It's not the threads themselves, it's the **communication** that causes problems.
  - Mutable shared state introduces **implicit** communication.
- The number of possible states increases dramatically as the number of threads increases.
- There are several concurrency-specific types of bugs.
- The performance of different approaches can vary considerably, and performance consequences are not obvious.

C++0x Tools for Multithreading

The C++0x toolset is deliberately basic, but there's a couple of real gems. The standard provides:

- Thread Launching
- Mutexes for synchronization
- Condition variables for blocking waits
- Atomic variables for low-level code
- Futures for high level concurrency design
- `std::lock()` for avoiding deadlock

Futures

- A future is a "token" for a value that will be available later
- Focus on **communication** between threads
- Synchronization details left to library

C++0x Support for Futures

- `std::unique_future` and `std::shared_future` — akin to `std::unique_ptr` and `std::shared_ptr`
- `std::packaged_task` where the value is the result of a function call
- `std::promise` where the value is set explicitly
- (Possibly) `std::async()` — library manages thread for the function call

std::unique_future / std::shared_future

- get() blocks until result available and then
  - Returns stored value or
  - Throws stored exception
- Use wait() to wait without retrieving the result
- Use is_ready(), has_value() and has_exception() to query the state.

std::async()

Run a function asynchronously and get a future for the return value:

```
int find_the_answer_to_LtUaE();
std::unique_future<int> the_answer=
    std::async(find_the_answer_to_LtUaE);
```

std::async()

Run a function asynchronously and get a future for the return value:

```
int find_the_answer_to_LtUaE();
std::unique_future<int> the_answer=
    std::async(find_the_answer_to_LtUaE);
```

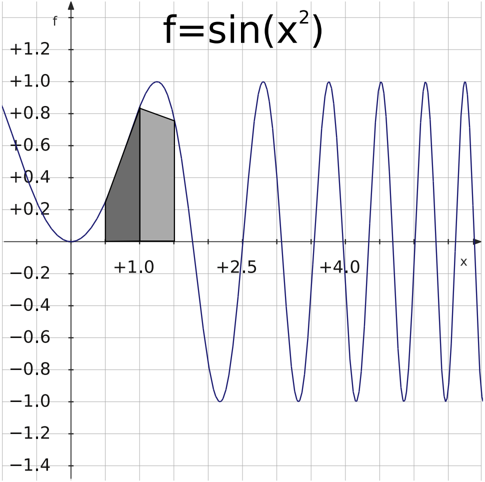
```
std::cout<<the_answer.get()<<std::endl;
```

Approximating std::async()

std::async() is not yet in the working paper, and **may not make it into C++0x**. You can write a version that always starts a new thread quite simply:

```
std::unique_future<typename std::result_of<Func()>::type>
async(Func f)
{
    typedef typename std::result_of<Func()>::type
        result_type;
    std::packaged_task<result_type()> task(f);
    std::unique_future<result_type> uf(task.get_future());
    std::thread t(std::move(task));
    t.detach();
    return uf;
}
```

Numerical Integration



Exception Safety with async()

```
int sum(int* start,int* end)
{
    return std::accumulate(start,end,0);
}

void foo()
{
    int x[]={...};
    std::unique_future<int> res=
        async(std::bind(sum,
                        &x,x+sizeof(x)/sizeof(int)));
    throw some_exception();
} // async call still running?
```

RAII to the rescue (1)

```
template<typename T>
class future_waiter
{
    std::unique_future<T>& future;
public:
    explicit future_waiter(std::unique_future<T>& f):
        future(f)
    {}
    ~future_waiter()
    {
        future.wait();
    }
};
```

RAII to the rescue (2)

Our leaky code now becomes:

```
void foo()
{
    int x[]={...};
    std::unique_future<int> res=
        async(std::bind(sum,
                        &x,x+sizeof(x)/sizeof(int)));
    future_waiter w(res);
    throw some_exception();
}
```

Controlling Threads Manually

Threads are managed manually with `std::thread`.

- Start a thread with the `std::thread` constructor
- Wait for a thread with `t.join()`
- Leave a thread to run in the background with `t.detach()`

Lifetime issues with std::thread (1)

If you don't call join() or detach() on a thread, the destructor calls std::terminate().

```
void do_stuff()
{}

int main()
{
    std::thread t(do_stuff);
} // thread not joined or detached
  // => std::terminate() called.
```

Lifetime issues with std::thread (2)

The call to std::terminate() from the destructor protects against lifetime-related race conditions:

```
void update_value(int* value)
{
    *value=42;
}

int main()
{
    int i;
    std::thread t(update_value,&i);
} // thread may still be running and accessing i
  // => std::terminate() called.
```

Lifetime issues with std::thread (3)

Even if you join at the end of the scope, you've still got the potential for problems:

```
void foo()
{
    int i;
    std::thread t(update_value,&i);
    do_something(); // may throw
    t.join();
} // if exception thrown, join() call skipped
```

Lifetime issues with std::thread (4)

Again, you can handle this with RAII:

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
    ~thread_guard()
    {
        if(t.joinable())
            t.join();
    }
};
```

Lifetime issues with std::thread (5)

Our troublesome code now looks like this:

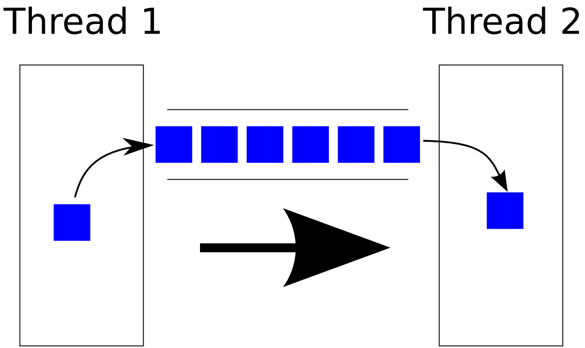
```
void foo()
{
    int i;
    std::thread t(update_value,&i);
    thread_guard guard(t);
    do_something(); // may throw
} // if exception thrown, join() still called
```

Key points

- You **must** explicitly join or detach every thread in **all code paths**.
- You must ensure that a thread or asynchronous task is finished before the data it accesses is destroyed.
- RAII can help with both of these.

Passing a series of data items

Futures are for single data items. What about a series of items?



Passing a series of data items (2)

To pass a series of items in order we need a queue — add items on one end take them off the other.

`std::queue` would do the job, but it's not thread-safe.

The simplest solution is therefore to use a `std::queue` protected by a mutex.

Building a concurrent queue

```
template<typename Data>
class concurrent_queue
{
    std::mutex the_mutex;
    std::queue<Data> the_queue;
public:
    void push(Data const& data)
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    // other member functions
};
```

Racy interfaces

A mutex doesn't save us from bad interface design. `std::queue`'s interface is not designed for concurrency.

Thread A	Thread B
`if(q.empty()) return;`	
	`if(q.empty()) return;`
`Data local=q.front();`	
	`Data local=q.front();`
`q.pop();`	
	`q.pop();`

Encapsulate entire operation under single lock

We need to group the calls to `empty()`, `front()` and `pop()` under the same mutex lock to avoid races:

```
bool concurrent_queue::try_pop(Data& data)
{
    std::lock_guard<std::mutex> lk(the_mutex);
    if(the_queue.empty()) return false;
    data=the_queue.front();
    the_queue.pop();
    return true;
}
```

Waiting for an item

If all we've got is `try_pop()`, the only way to wait is to poll:

```
concurrent_queue<my_class> q;
my_class d;

while(!q.try_pop(d))
    std::this_thread::yield(); // or sleep
do_stuff(d);
```

This is not ideal.

Performing a blocking wait

We want to wait for a particular condition to be true (there is an item in the queue).
This is a job for `std::condition_variable`:

```
void concurrent_queue::wait_and_pop(Data& data)
{
    std::unique_lock<std::mutex> lk(the_mutex);
    the_cv.wait(lk,
                [&the_queue]()
                {return !the_queue.empty();});
    data=the_queue.front();
    the_queue.pop();
}
```

Signalling a waiting thread

To signal a waiting thread, we need to *notify* the condition variable when we push an item on the queue:

```
void concurrent_queue::push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

Contention

- We only have one mutex protecting the data, so only one `push()` or one `pop()` can actually do any work at any one time.
- This can actually have a negative impact on performance when using multiple threads if the contention is too high.
- Can address this with multiple mutexes or a lock-free queue, but the complexity is much higher.
- Lowering contention is usually a better option.

Key points

- Mutexes don't protect you if your interface is racy.
- Put entire operation inside one lock to avoid races.
- Condition variables allow blocking waits.
- `std::lock_guard` and `std::unique_lock` provide RAII locking.
- Notify with mutex unlocked for maximum performance.
- Contention is still a performance killer.

Deadlock example

Suppose you have a class with some internal state, which you've protected with a mutex in order to make it thread-safe. Suppose also you want to write a comparison operator:

```
class X {
    mutable std::mutex the_mutex;
    int some_data;
public:
    bool operator<(X const& other) {
        std::lock_guard<std::mutex> lk(the_mutex);
        std::lock_guard<std::mutex> lk(other.the_mutex);
        return some_data < other.some_data;
    }
};
```

This **seems** perfectly safe at first glance...

Deadlock (2)

... but it isn't! If you've got two objects `x1` and `x2`, and two threads are trying to compare them, but different ways round:

Thread A	Thread B
`if(x1 < x2) ...`	`if(x2 < x1) ...`

The two threads will acquire the mutexes in opposite orders, which provides the possibility of deadlock.

Use `std::lock` to avoid deadlocks

If you **do** need to acquire two (or more) locks in order to perform an operation, `std::lock` is your friend. It guarantees to lock all the supplied mutexes without deadlock, whatever order they are given in. Our code then becomes:

```
bool X::operator<(X const& other)
{
    std::unique_lock<std::mutex> l1(the_mutex,
                                    std::defer_lock);
    std::unique_lock<std::mutex> l2(other.the_mutex,
                                    std::defer_lock);
    std::lock(l1,l2);
    return some_data < other.some_data;
}
```

Key points

- You can construct a `std::unique_lock` without locking using the `std::defer_lock` parameter.
- `std::lock` avoids deadlock for locks acquired together.
  - It works on any `Lockable` object.
- You can still get deadlock if locks acquired separately.

Concurrency-related Bugs

There are essentially two types of concurrency-related bug:

- Race Conditions: Data Races, broken invariants, lifetime issues
- Unwanted blocking: Deadlock, livelock

Locating concurrency-related bugs

- Write simple testable code
- Limit communication between threads to self-contained sections
- Code reviews
- More code reviews
- Brute force testing
- Combination simulation testing
- Testing with a debug library

Code Reviews

Here's a few things to think about when reviewing multithreaded code:

- Where are the communication paths?
- Which data is shared?
- How is the shared data protected?
- Where could other threads be when this thread is here?
- Which mutexes does this thread hold?
- Which mutexes can other threads hold?
- Is the data still valid?
- If the data could be changed, how can we avoid this?

Considerations for designing concurrent code

- How to divide work between threads
  - Before processing begins (e.g. static problems, problem size fixed at runtime)
  - Dynamically during processing (e.g. recursive problems)
  - Divide by task type (e.g. pipeline architecture)
- Performance
  - Cost of launching a thread and thread communication
  - Data Proximity
  - Contention
    - False sharing
  - Oversubscription
- Exception Safety

References and Further Reading

The current C++0x committee draft: N2857
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2857.pdf`

My blog: `http://www.justsoftwaresolutions.co.uk/blog/`

The documentation for my `just::thread` library is available online at `http://www.stdthread.co.uk/doc/`

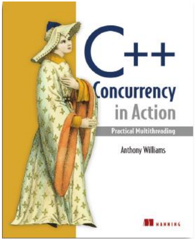
just::thread

just::thread provides a complete implementation of the C++0x thread library for MSVC 2008. gcc/linux support is currently in alpha testing.

For a 25% discount go to:

http://www.stdthread.co.uk/accu2009

My book



C++ Concurrency in Action: Practical Multithreading with the new C++ Standard, currently available under the Manning Early Access Program at

http://www.manning.com/williams/

Enter discount code **aupromo40** for a 40% discount.