The Future of Concurrency in C++

Anthony Williams

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

3rd April 2008

The Future of Concurrency in C++

- Multithreading Support in C++0x
- Existing proposals for TR2
- Beyond TR2

Multithreading Support in C++0x

- The Standard now acknowledges the existence of multi-threaded programs
- New memory model
- Support for thread-local static variables
- Thread Support Library
  - Threads
  - Mutexes
  - Condition Variables
  - One time initialization
  - Asynchronous results — futures

C++0x Thread Library and Boost

- Two-way relationship with Boost
  - Proposals for multithreading heavily influenced by Boost.Thread library
  - Boost 1.35.0 Thread library revised in line with C++0x working draft

Atomics and memory model

- Define the rules for making data visible between threads
- Atomics are generally for experts only
- If you correctly use locks, everything "just works"

Synchronizing Data

There are two critical relationships between operations:

- Synchronizes-with relation
  - Store-release synchronizes-with a load-acquire
- Happens-before relation
  - A sequenced before B in a single thread
  - A synchronizes-with B
  - A happens-before X, X happens-before B

Data races

A data race occurs when:

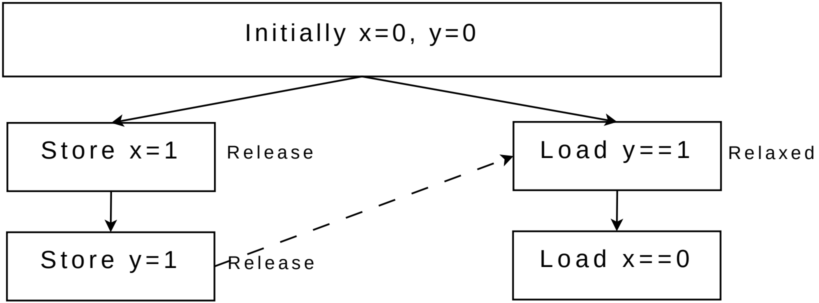
- Two threads access non-atomic data
- At least one access is a write
- There is no *happens-before* relation between the accesses

A lot of multithreaded programming is about avoiding data races

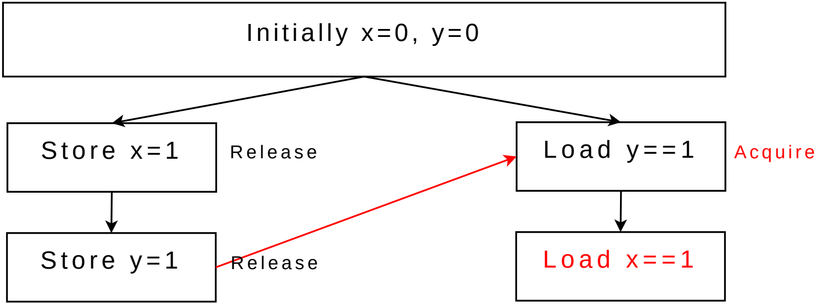
Memory Ordering Constraints

- Sequential Consistency
  - Single total order for all SC ops on all variables
  - default
- Acquire/Release
  - Pairwise ordering rather than total order
  - Independent Reads of Independent Writes don't require synchronization between CPUs
- Relaxed Atomics
  - Read or write data without ordering
  - Still obeys happens-before

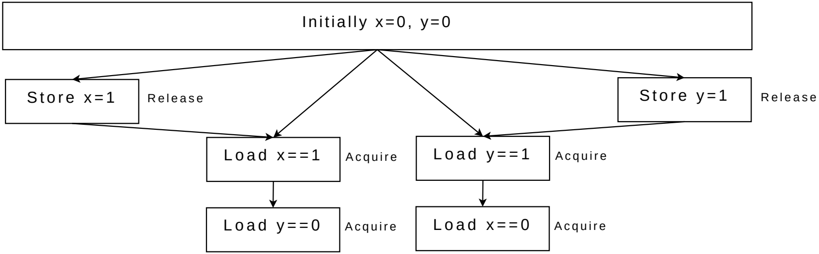
Relaxed Ordering



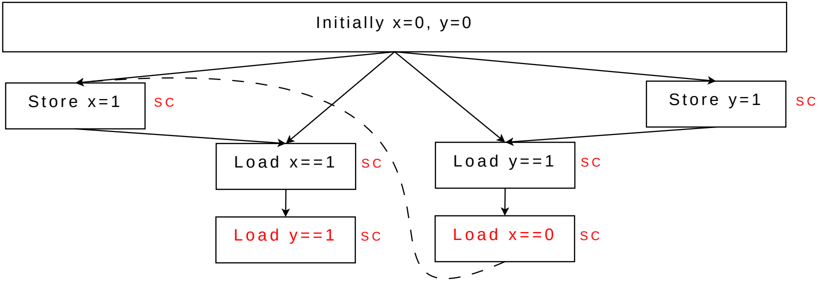
Acquire-Release Ordering



Acquire-Release Ordering



Sequentially Consistent Ordering



Basic interface for atomics

- atomic_flag
  - Boolean flag
  - Must be lock-free
- Atomic integeral types — e.g. atomic_char, atomic_uint, atomic_llong
  - Includes arithmetic operators such as a++, and a|=5
  - Operators return underlying type by value, not reference
  - May not be lock-free — use a.is_lock_free() to check
- atomic_address
  - Represents a void*
  - May not be lock-free — use a.is_lock_free() to check
- Free functions for C compatibility

Generic interface for atomics

- `atomic<T>`
  - derived from `atomic_T` for built-in integral and pointer types
- works with "trivially default constructible and bitwise equality comparable" types
  - Lock-free where possible

Compare and Swap

- Generally put in loop
  - Spurious failure
  - Other thread may change value anyway

```
atomic<int> a;
int desired;
int expected=a;

do
{
    desired=function(expected);
}
while(!a.compare_swap(expected,desired));
```

Fences

- Per-object fences: `a.fence(memory_order)`
  — RMW op which writes same value back.
- Global fences with `atomic_global_fence_compatibility` object (of type `atomic_flag`)

Thread launching

```
std::thread t(func,arg1,arg2);
```

– std::bind semantics

Joining a Thread

```
std::thread t(func);
t.join();
```

A thread can only be joined once.

Detaching a Thread

- Explicitly:

```
    std::thread t1(func);
    t1.detach();
```

- Implicitly:

```
{
    std::thread t2(func);
} // destructor of t2 calls detach()
```

Transferring Ownership

- At most one `std::thread` object per thread.
- Thread objects are movable
  - Can return `std::thread` from functions

    ```
    std::thread start_process(some_args);
    ```

  - Can store `std::thread` objects in standard containers

    ```
    std::vector<std::thread> vec;
    vec.push_back(std::thread(some_func));
    ```

- Can use `t.joinable()` to determine if an object has an associated thread.

Identifying Threads

- Every thread has a unique ID
- Thread IDs represented by instances of `std::thread::id`
  - Value Type: copyable, usable in comparisons
  - Non-equal values form a total order
  - Can be used as keys in associative containers and unordered associative containers
  - Can be written to an output stream
  - Default constructed ID is "Not any Thread".

Obtaining Thread IDs

- `std::this_thread::get_id()`
  returns the ID of the current thread
- `t.get_id()`
  Returns the ID of the thread associated with the `std::thread` instance `t`

Mutexes

There are four mutex types in the current working paper:

- `std::mutex`
- `std::recursive_mutex`
- `std::timed_mutex`
- `std::recursive_timed_mutex`

Locking

- `lock()` and `unlock()` member functions are public
- Scoped locking:
  - `std::lock_guard` template
  - `std::unique_lock` template
    - movable, supports deferred locking, timed locking
    - can itself be used as a "mutex".
- Generic `lock()` function

  — Allows locking of more than one mutex without deadlock

Condition Variables

- Two types of condition variables:
  - `std::condition_variable`
  - `std::condition_variably_any`
- The difference is the lock parameter to the wait functions:
  - `void condition_variable::wait( unique_lock<std::mutex>& lock);`
  - `template<typename lock_type> void condition_variable_any::wait( lock_type& lock);`

Condition Variables and Predicates

- Condition variables are subject to spurious wakes
- Correct usage requires a loop:

```
std::unique_lock<std::mutex> lk(some_mutex);
while(!can_continue())
{
    some_cv.wait(lk);
}
```

- Predicate version makes things simpler:

```
std::unique_lock<std::mutex> lk(some_mutex);
some_cv.wait(lk,&can_continue);
```

Timed waits with condition variables

- The overload of `condition_variable::timed_wait()` that takes a duration is particularly error-prone:

```
while(!can_continue())
{
    some_cv.timed_wait(lk,std::milliseconds(3));
}
```

  This may actually be equivalent to just using `wait()`, in the event of spurious wake-ups

- The predicate overload avoids this problem:

```
some_cv.timed_wait(lk,std::milliseconds(3),
                   &can_continue);
```

One-time Initialization

- Provided by `std::call_once`

General Usage of call_once

```
std::once_flag flag;

std::call_once(flag,some_function);
// calls some_function()

std::call_once(flag,some_other_function,arg1,arg2);
// calls some_other_function(arg1,arg2)
```

– std::bind semantics again

Lazy initialization of class members

```
class X
{
    some_resource_handle h;
    std::once_flag flag;
    void init_resource();
public:
    X():h(no_resource){}
    void do_something()
    {
        std::call_once(flag,&X::init_resource,this);
        really_do_something(h);
    }
};
```

Thread-local static variables

- Not yet in WP: N2545 by Lawrence Crowl
- Each thread has its own instance of the variable
- Use the `thread_local` keyword:
  `static thread_local int x;`
- Any variable with static storage duration can be declared `thread_local`:
  - Namespace-scope variables
  - `static` data members of classes
  - `static` variables declared at block scope
- `thread_local` variables can have constructors and destructors.

Asynchronous Value Computation

- Not yet in WP: N2561
  — Deltef Vollman, Howard Hinnant and myself
- Value is result of a task running on another thread.
- No control over how or when value is computed by recipient.
- Answer to how to return a value from a thread.

Futures

Two templates for futures:

- `std::unique_future<T>` — like `std::unique_ptr<T>`
  - sole owner
  - read once (move)
- `std::shared_future<T>` — like `std::shared_ptr<T>`
  - multiple owners
  - can be read multiple times (copy)
- Can move a `std::unique_future<T>` into a `std::shared_future<T>`

Getting the values: std::unique_future<T>

- `R move()`
  - blocks until ready
  - throws if already moved
  - throws if future has a stored exception
- `bool try_move(R&)`
  - returns false if not ready() or already moved.
- State query functions:
  `is_ready()`, `has_value()`, `has_exception()`, `was_moved()`
- Wait for ready:
  `wait()`, `timed_wait()`

Getting the value: `std::shared_future<T>`

- `R const& get()`
  `operator R const&()`
  - Blocks until ready
  - Returns reference to stored value
  - Throws if future has a stored exception
- `bool try_get(R&)`
- State Query functions:
  `is_ready()`, `has_value()`, `has_exception()`
  No `was_moved()` has the result can't be moved
- Wait for ready:
  `wait()`, `timed_wait()`

Generating Asynchronous values

Two ways of generating asynchronous values:

- `std::packaged_task<T>`
  — value is the result of a function call
- `std::promise<T>`
  — explicit functions for populating the value

Packaged Tasks

- A `std::packaged_task<T>` is like `std::function<T()>` — it wraps any function or callable object, and invokes it when `std::packaged_task<T>::operator()` is invoked.
- Return value populates a `std::unique_future<T>` rather than being returned to caller
- Simplest way to get the return value from a thread

Returning a value from a thread with std::packaged_task<T>

```
template<typename Callable>
std::unique_future<std::result_of<Callable()>::type>
run_in_thread(Callable func)
{
    typedef std::result_of<Callable()>::type rtype;
    std::packaged_task<rtype> task(std::move(func));
    std::unique_future<rtype> res(task.get_future());
    std::thread(std::move(task)).detach();
    return std::move(res);
}
```

Promises

- Value can come from any number of possible sources — e.g. first worker in pool to calculate result
- More explicit interface:
  - `p.set_value(some_value)`
  - `p.set_exception(some_exception)`

TR2

- Already some proposals for C++0x which have been retargeted to TR2
- `shared_mutex`, `upgrade_mutex` (from N2094)
- thread pools (from N2094, N2185, N2276)

shared_mutex

- Provides a multiple-reader/single-writer mutex
- single writer:
  - `m.lock()/m.unlock()`
  - `std::unique_lock<shared_mutex>`
- multiple readers:
  - `m.lock_shared()/m.unlock_shared()`
  - `shared_lock<shared_mutex>`

upgrade_mutex

- multiple readers + single upgrader / single writer
- The one and only upgrader can upgrade to a writer
  - Blocks until all readers have finished
  - Prevents other writers acquiring lock
  - Allows thread to rely on data read prior to upgrade
- Lock/unlock upgrader with:
  - `m.lock_upgrade()`/`m.unlock_upgrade()`
  - `upgrade_lock<upgrade_mutex>`
- Upgrade with:
  - `m.unlock_upgrade_and_lock()`
  - Move-construction of an `upgrade_lock<upgrade_mutex>` to `unique_lock<upgrade_mutex>`
- Locks can be downgraded

boost::shared_mutex

In boost 1.35.0, boost::shared_mutex provides all this functionality.

Thread Pools

- Universal agreement that we need to provide some kind of thread pool support.
- Exact API is not yet clear.
- N2094, N2185, N2276 provide distinct but similar APIs.
- Philipp Henkel has written a thread pool library that works with boost
  — `http://threadpool.sourceforge.net`.
- Yigong Liu's Join library provides an alternative approach
  — `http://channel.sourceforge.net`

TR2 and beyond I

- Thread Interruption
  - Present in Boost 1.35.0
  - Interrupt a thread by calling `t.interrupt()` on a thread object `t`.
  - Thread throws `thread_interrupted` exception at next *interruption point*
  - *Interruption points* include `condition_variable::wait()`, `this_thread::sleep()` and `interruption_point()`
  - Interruption can be disabled with instances of `disable_interruption`
  - The `thread_interrupted` exception can be caught: the thread can then be interrupted again

TR2 and beyond II

- Thread-safe containers:
  - `concurrent_queue`
  - `concurrent_stack`
  - `concurrent_list`
  - `concurrent_unordered_map`
- Parallel algorithms
  - `parallel_find`
  - `parallel_sort`
  - `parallel_accumulate`
  - `parallel_for`
- Intel TBB provides some of these
  — `http://threadingbuildingblocks.org/`

TR2 and beyond III

- Software Transactional Memory (STM)
  Allows for ACID guarantees in concurrent code, just like in databases
- OpenMP (`http://www.openmp.org`)
  A set of compiler directives to highlight code that should be run in parallel
- Auto-parallelisation in compilers
  A step beyond OpenMP — compilers identify parallelizable regions automatically.
  The current Intel compiler has basic support for this, with the `-parallel` command-line option.

References and Further Reading

- The current C++0x working paper: N2588
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf
- The Boost 1.35.0 thread docs
  http://www.boost.org/doc/libs/1_35_0/doc/html/thread.html
- The futures proposal: N2561
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2561.html
- My book: *C++ Concurrency in Action: Practical Multithreading*, due to be published by Manning end of 2008/early 2009.