Multi-Threading With C++11 and Boost

C++ Now 2013

Rob Stewart

Overview

- Mutexes
- Lock guards
- Condition variables
- Thread-safe queue
- Barriers
- Periodic callable invocation

Mutexes

Mutexes

- Blocking lock
- Non-blocking lock
- Time limited blocking lock
- Unique or recursive

Using std::mutex

```
#include <cassert>
#include <mutex>

int
main()
{
    std::mutex m;
    m.lock();                 // 1
    assert(!m.try_lock());    // 2
    m.unlock();               // 3
    assert(m.try_lock());     // 4
    m.unlock();               // 5
}
```

std::recursive_mutex

- Just like std::mutex, except…
- Owning thread can lock repeatedly
- Released when unlocked as many times as locked

Using std::timed_mutex

```
#include <cassert>
#include <chrono>
#include <mutex>

int
main()
{
    std::timed_mutex m;
    if (m.try_lock_for(std::chrono::milliseconds(1000)))  // 1
        m.unlock();                                       // 2
    auto time(std::chrono::steady_clock::now()
        + std::chrono::seconds(1));
    if (m.try_lock_until(time))                           // 3
        m.unlock();                                       // 4
}
```

std::recursive_timed_mutex

- Just like std::timed_mutex, except…
- Owning thread can lock repeatedly
- Released when unlocked as many times as locked

Lock Guards

Using std::lock_guard

```
#include <cassert>
#include <mutex>

int
main()
{
    std::mutex m;
    std::lock_guard<std::mutex> _(m);   // 1
    assert(!m.try_lock());              // 2
}
```

Using std::lock_guard (Adopting)

```
#include <cassert>
#include <mutex>

int
main()
{
	std::mutex m;
	m.lock();
	std::lock_guard<std::mutex> _(m, std::adopt_lock);
}
```

Controlling Lock Scope

```
#include <cassert>
#include <mutex>

int main()
{
    std::mutex m;
    {
        std::lock_guard<std::mutex> _(m);   // 1
        // do work with lock                // 2
    }
    // do work without lock                 // 3
}
```

Using std::unique_lock

```
#include <cassert>
#include <mutex>

int main()
{
    std::mutex m;
    {
        std::unique_lock<std::mutex> guard(m);    // 1
        assert(guard.owns_lock());                // 2
        assert(&m == guard.release());            // 3
        assert(!guard.owns_lock());               // 4
    }
    m.unlock();                                   // 5
}
```

Using std::unique_lock (Adopting)

```
#include <cassert>
#include <mutex>

int main()
{
        std::mutex m;
        m.lock();
        std::unique_lock<std::mutex> guard(m, std::adopt_lock);
        assert(guard.owns_lock());
}
```

Using std::unique_lock (Deferred)

```
#include <cassert>
#include <mutex>

int main()
{
    std::mutex m;
    std::unique_lock<std::mutex> guard(m, std::defer_lock);    // 1
    assert(!guard.owns_lock());                                // 2
    guard.lock();                                              // 3
    assert(guard.owns_lock());                                 // 4
}
```

Using std::unique_lock (Try to Lock)

```
#include <cassert>
#include <mutex>

int main()
{
	std::mutex m;
	std::unique_lock<std::mutex> guard(m, std::try_to_lock);   // 1
	assert(guard.owns_lock());                                 // 2
	guard.unlock();                                            // 3
	assert(guard.try_lock());                                  // 4
}
```

Using std::unique_lock (Try to Lock for Duration)

```
#include <chrono>
#include <mutex>

int main()
{
    std::timed_mutex m;
    std::chrono::milliseconds const duration(1000);
    std::unique_lock<std::timed_mutex> guard(m, duration);   // 1
    guard.try_lock_for(duration);                            // 2
}
```

Using std::unique_lock (Try to Lock Until Time)

```
#include <chrono>
#include <mutex>

int main()
{
    std::timed_mutex m;
    auto until(std::chrono::steady_clock::now()
        + std::chrono::seconds(1));
    std::unique_lock<std::timed_mutex> guard(m, until);
    guard.try_lock_until(until);
}
```

Condition Variables

Condition Variables

- Synchronize state changes between threads
- State Changer
  1. Acquires mutex
  2. Changes state
  3. Notifies one or all waiting threads
- State Watcher
  1. Acquires mutex
  2. Waits on condition variable
  3. Examines state
  4. Possibly waits longer (spurious wake-up)

Using std::condition_variable

```
#include <cassert>
#include <condition_variable>
#include <mutex>

typedef std::unique_lock<std::mutex> guard_type;
std::condition_variable cv;
std::mutex lock;
bool state;

void changer()
{
    guard_type _(lock);
    state = true;
    cv.notify_one();
}
• • •
```

Using std::condition_variable (cont.)

```
void watcher()
{
    guard_type guard(lock);
    while (!state)
    {
        cv.wait(guard);
    }
    assert(state);
}
```

Using std::condition_variable (cont.)

```
void changer()
{
    guard_type guard(lock);
    state = true;
    cv.notify_one();
}

void watcher()
{
    guard_type guard(lock);
    cv.wait(guard, [] { return state; });
    assert(state);
}
```

Thread-safe Queue

Thread-safe Queue

- Large design space
  - One or more producers
  - One or more consumers
  - Fixed or dynamic size
  - Storage/container
  - Behavior when full
- No ideal

MPMC Thread-safe Queue

- Multiple producers
- Multiple consumers
- Fixed size boost::circular_buffer for storage
- Producers/consumers may block or not
- Producers signal blocking consumers

MPMC Thread-safe Queue Synopsis

```
#include <condition_variable>
#include <mutex>
#include <boost/circular_buffer.hpp>

template <class T>
class mpmc_queue
{
public:
    mpmc_queue(size_t);

    void pop(T &);
    bool try_pop(T &);

    void push(T const &);
    bool try_push(T const &);
};
```

MPMC Thread-safe Queue v2 Synopsis

```
#include <condition_variable>
#include <mutex>
#include <boost/circular_buffer.hpp>

template <class T>
class mpmc_queue
{
public:
    enum vacancy { had_room, was_full };

    mpmc_queue(size_t);

    void pop(T &);
    bool try_pop(T &);

    vacancy push(T const &);
    vacancy push(T &, T const &);
    bool try_push(T const &);
    . . .
```

MPMC Thread-safe Queue Synopsis

```
    . . .
private:
    typedef std::unique_lock<std::mutex> guard_type;

    std::condition_variable         cv_;
    mutable std::mutex              lock_;
    boost::circular_buffer<T>       queue_;
};
```

mpmc_queue()

```
template <class T>
mpmc_queue<T>::mpmc_queue(size_t const _size)
      : queue_(_size)
{
}
```

pop(T &)

```
template <class T>
void mpmc_queue<T>::pop(T & _data)
{
    guard_type guard(lock_);
    while (queue_.empty())
    {
        cv_.wait(guard);
    }
    _data = queue_.front();
    queue_.pop_front();
}
```

pop(T &) (Predicated Wait)

```
template <class T>
void mpmc_queue<T>::pop(T & _data)
{
    guard_type guard(lock_);
    cv_.wait(guard, [this] { return !queue_.empty(); });
    _data = queue_.front();
    queue_.pop_front();
}
```

try_pop(T &)

```
template <class T>
bool mpmc_queue<T>::try_pop(T & _data)
{
    guard_type _(lock_);
    bool const result(!queue_.empty());
    if (result)
    {
      _data = queue_.front();
        queue_.pop_front();
    }
    return result;
}
```

push(T const &)

```
template <class T>
typename mpmc_queue<T>::vacancy
mpmc_queue<T>::push(T const & _data)
{
    guard_type guard(lock_);
    vacancy const result(queue_.full() ? was_full : had_room);
    queue_.push_back(_data);
    guard.unlock();
    cv_.notify_one();
    return result;
}
```

push(T &, T const &)

```
template <class T>
typename mpmc_queue<T>::vacancy
mpmc_queue<T>::push(T & _oldest, T const & _data)
{
    guard_type guard(lock_);
    vacancy const result(queue_.full() ? was_full : had_room);
    if (was_full == result)
    {
        _oldest = queue_.front();
    }
    queue_.push_back(_data);
    guard.unlock();
    cv_.notify_one();
    return result;
}
```

try_push(T const &)

```
template <class T>
bool mpmc_queue<T>::try_push(T const & _data)
{
    guard_type guard(lock_);
    bool const result(!queue_.full());
    if (result)
    {
        queue_.push_back(_data);
        guard.unlock();
        cv_.notify_one();
    }
    return result;
}
```

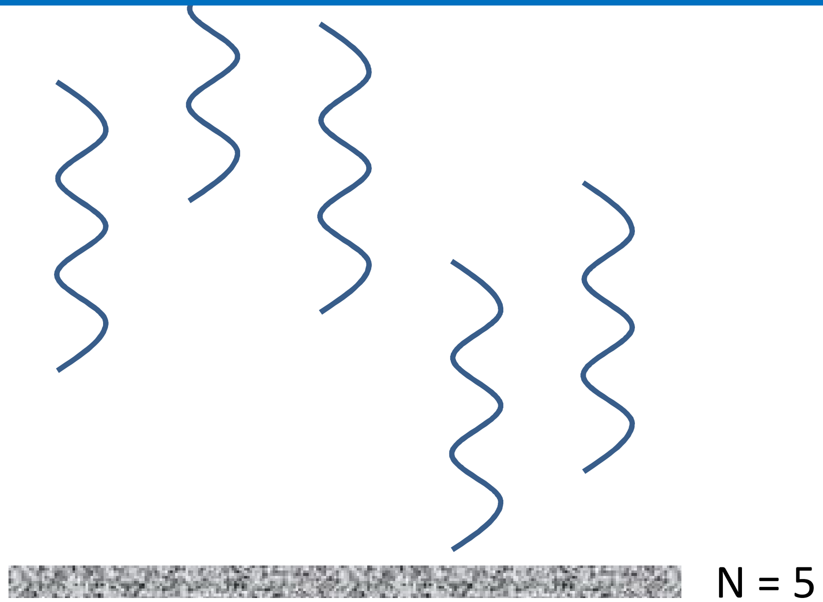
Boost.Threads

vs.

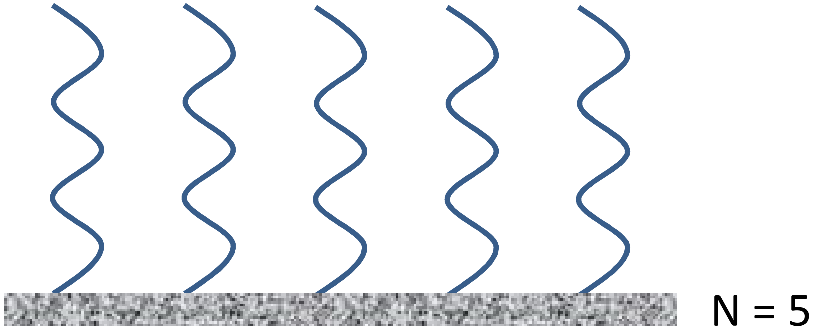
Boost.Threads vs. C++11

Barriers

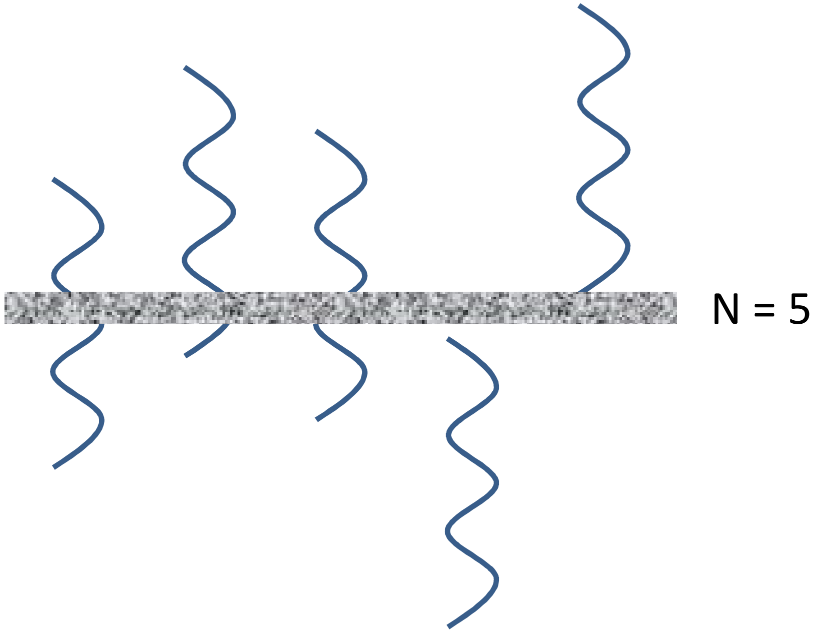
Barriers



Barriers



Barriers



Barrier Uses

- Prevent races between threads with dependencies
  - Wait for parallel algorithm tasks to finish before collecting results
  - Wait for tasks to initialize before starting work
- Force test threads to begin only when all threads exist and are ready

boost::barrier Synopsis

```
// #include <boost/thread/barrier.hpp>

struct boost::barrier
{
    barrier(unsigned);

    barrier(barrier const &) = delete;
    barrier const & operator =(barrier const &) = delete;

    bool wait();
};
```

Using boost::barrier

```
#include <thread>
#include <vector>
#include <boost/thread/barrier.hpp>

unsigned const count(30);                  // 1
boost::barrier barrier(count + 1);         // 2

void work();                               // 3

int main()
{
    for (unsigned i(0); i < count; ++i)    // 4
    {
        boost::thread(work).detach();      // 5
    }
    · · · // next slide
}
```

Using boost::barrier (cont.)

```
void work()
{
    do_initial_work();
    barrier.wait();
    do_remaining_work();
    barrier.wait();
}

int main()
{
    for (unsigned i(0); i < count; ++i) boost::thread(work).detach();
    barrier.wait(); // wait for all threads to do initial work
    barrier.wait(); // wait for all threads to do remaining work
}
```

Using boost::barrier (cont.)

```
void work()
{
    do_initial_work();
    barrier.wait();
    do_remaining_work();
    barrier.wait();
}

int main()
{
    for (unsigned i(0); i < count; ++i) boost::thread(work).detach();
    barrier.wait(); // wait for all threads to do initial work
    barrier.wait(); // wait for all threads to do remaining work
}
```

Using boost::barrier (cont.)

```
void work()
{
      do_initial_work();
      barrier.wait();
      do_remaining_work();
      barrier.wait();
}

int main()
{
      for (unsigned i(0); i < count; ++i) boost::thread(work).detach();
      barrier.wait(); // wait for all threads to do initial work
      barrier.wait(); // wait for all threads to do remaining work
}
```

Barrier Class for C++11

- Barriers are useful
- Barriers aren’t in C++11
- Don’t mix boost::barrier with C++11 threading constructs
  - Duplicate code
  - Extra dependencies
- Need C++11-based barrier

Barrier Class Interface

```
struct barrier
{
    barrier(unsigned);

    barrier(barrier const &) = delete;
    barrier & operator =(barrier const &) = delete;

    void wait();
};
```

Barrier Class Requirements

- Require non-zero thread count
- Waiting threads block until enough waiting
- Release all waiting threads when enough waiting
- Once waiting threads are released, more can wait

Non-zero Thread Count

```
#include <stdexcept>

barrier::barrier(unsigned const _count)
{
    if (0 == _count)
    {
        throw std::invalid_argument(
            “barrier thread count cannot be zero");
    }
}
```

Block Threads Until Enough Waiting

- Track number of waiting threads
- When too few waiting, wait for more
- When enough waiting, release waiters

When Too Few, Wait for More

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (++waiters_ != expected_)
    {
        cv_.wait(guard);
    }
}
```

Barrier Class Interface (updated)

```
class barrier
{
public:
    barrier(unsigned);

    void wait();

private:
    std::condition_variable    cv_;
    unsigned                   expected_;
    std::mutex                 lock_;
    unsigned                   waiters_;
};
```

When Enough, Release Waiters

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (++waiters_ == expected_)
    {
        cv_.notify_all();
    }
    else
    {
        cv_.wait(guard);
    }
}
```

Once Released, More Can Wait

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (++waiters_ == expected_)
    {
        waiters_ = 0;
        cv_.notify_all();
    }
    else
    {
        cv_.wait(guard);
    }
}
```

Spurious Wakeups

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (++waiters_ == expected_)
    {
        waiters_ = 0;
        cv_.notify_all();
    }
    else
    {
        cv_.wait(guard); // need a condition
    }
}
```

Spurious Wakeups

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (1 == ++waiters_)
    {
        proceed_ = false;
    }
    if (waiters_ == expected_)
    {
        waiters_ = 0;
        proceed_ = true;
        cv_.notify_all();
    }
    else
    {
        cv_.wait(guard, [this] { return proceed_; });
    }
}
```

Spurious Wakeups (cont.)

What happens when some threads wait a second time when some waiting the first time have not awakened?

- First thread waiting second time resets proceed_
- Lagging, but notified, threads blocked

Spurious Wakeups

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    if (1 == ++waiters_)
    {
        proceed_ = false;
    }
    if (waiters_ == expected_)
    {
        waiters_ = 0;
        proceed_ = true;
        cv_.notify_all();
    }
    else
    {
        cv_.wait(guard, [this] { return proceed_; });
    }
}
```

Tracking Generations of Waiters

```
void barrier::wait()
{
    std::unique_lock<std::mutex> guard(lock_);
    unsigned const generation(generation_);
    if (++waiters_ == expected_)
    {
        waiters_ = 0;
        ++generation_;
        cv_.notify_all();
    }
    else
    {
        while (generation == generation_)
        {
            cv_.wait(guard);
        }
    }
}
```

Barrier Class Definition (updated)

```
class barrier
{
public:
    barrier(unsigned);

    void wait();

private:
    std::condition_variable    cv_;
    unsigned                   expected_;
    unsigned                   generation_;
    std::mutex                 lock_;
    unsigned                   waiters_;
};
```

Update Constructor

```
#include <stdexcept>

barrier::barrier(unsigned const _count)
    : expected_(_count)
    , generation_(0)
    , waiters_(0)
{
    if (0 == _count)
    {
        throw std::invalid_argument(
            “barrier thread count cannot be zero");
    }
}
```

Periodic Invocation

Periodic Invocation

- Invoke callable
  - With a fixed interval between
  - At a regular rate
- Thread count
  - One per callable
  - Pooled

Fixed Interval

```
void invoke(std::function<void()> const & _task,
    std::chrono::milliseconds const _interval)
{
    for (;;)
    {
        _task();
        std::this_thread::sleep_for(_interval):
    }
}
```

Invoking a Task in a Thread

```
void invoke(std::function<void()> const &, std::chrono::milliseconds);

void task() { /* some work */ }

int main()
{
    std::thread thread(invoke, task, std::chrono::milliseconds(500));
    thread.join();
}
```

Exceptions from Tasks

- What if the task emits an exception?
- std::thread calls std::terminate()
- Must prevent exceptions from propagating

Control Exceptions in Threads

```
void invoke(std::function<void()> const & _task,
    std::chrono::milliseconds const _interval)
{
    try
    {
        for (;;)
        {
            _task();
            std::this_thread::sleep_for(_interval):
        }
    }
    catch (...)
    {
        // handle somehow
    }
}
```

Not Interruptible

```
void invoke(std::function<void()> const & _task,
    std::chrono::milliseconds const _interval)
{
    for (;;)
    {
        _task();
        std::this_thread::sleep_for(_interval):
    }
}
```

Interruption in Boost.Threads

- boost::threads are interruptible
  - boost::thread::interrupt()
  - Triggers boost::thread_interrupted exception
- Interruptible at *interruption points*
  - boost::thread::interruption_point()
  - boost::this_thread::sleep(), sleep_for(), sleep_until()
  - boost::condition_variable::wait(), wait_for(), wait_until()
  - Others

Managing boost::thread_interrupted

- Handle the exception
- Allow it to propagate
  - Not an error
  - Thread exits quietly

No Interruption in C++11

- std::thread is not interruptible
- Monitor synchronized flag
  - Guard bool with mutex
  - Use std::atomic_bool
- Can create interrupter class to manage flag

thread_interrupter Synopsis

```
class thread_interrupter
{
public:
     thread_interrupter();

     void interrupt();
     bool interrupted() const;
     void check_for_interruption() const;

private:
     std::atomic_bool   interrupted_;
};
```

thread_interrupter Implementation

```
thread_interrupter::thread_interrupter()
	: interrupted_(false)
{
}

void thread_interrupter::interrupt()
{
	interrupted_ = true;
}

bool thread_interrupter::interrupted() const
{
	return interrupted_;
}
```

thread_interrupter Implementation (cont.)

```
struct thread_interrupted
{
};

void thread_interrupter::check_for_interruption() const
{
    if (interrupted_)
    {
        throw thread_interrupted();
    }
}
```

Interruptible

```
void invoke(std::function<void()> const & _task,
    std::chrono::milliseconds const _interval,
    thread_interrupter const & _interrupter)
{
    try
    {
        for (;;)
        {
            _task();
            _interrupter.check_for_interruption();
            std::this_thread::sleep_for(_interval):
            _interrupter.check_for_interruption();
        }
    }
    catch (thread_interrupted)
    {
    }
}
```

Invoking a Task in a Thread

```
void invoke(std::function<void()> const &, std::chrono::milliseconds,
    thread_interrupter const &);

void task() { /* some work */ }

int main()
{
    thread_interrupter interrupter;
    std::thread thread(invoke, task, std::chrono::milliseconds(500),
        std::cref(interrupter));
    std::this_thread::sleep_for(std::chrono::seconds(5));
    interrupter.interrupt();
    thread.join();
}
```

Regular Interval

```
void invoke(std::function<void()> const & _task,
    std::chrono::milliseconds const _interval)
{
    for (;;)
    {
        auto const start(std::chrono::system_clock::now());
        _task();
        auto const stop(std::chrono::system_clock::now());
        auto const elapsed(stop – start);
        if (elapsed < _interval)
        {
            std::this_thread::sleep_for(_interval – elapsed):
        }
    }
}
```

Oversubscription

- One thread per scheduled task
- Each thread requires resources
  - Stack space
  - Kernel structures
  - Scheduling overhead
- Given N cores
  - Cannot execute more than N tasks simultaneously
  - A task can be quiescent (sleep_for(), blocked on I/O)

Oversubscribed?

- t_n: Time to run task n
- i_n: Interval for task n
- I_n: Idle time for task n

$$I_n = i_n - t_n$$

$$\sum_{n=0}^{N} I_n < 0$$

Addressing Oversubscription

- Allocate or allow a maximum number of threads
- Put timed tasks in chronological queue
- Thread behavior
  - Dequeue a task
  - Invoke function
  - Enqueue again when next due
- *Thread pooling*

Task Queue

```
typedef std::chrono::system_clock::time_point time_type;
typedef std::chrono::milliseconds interval_type;

struct scheduled_task
{
    std::function<void()> function;
    interval_type         interval;
    time_type             time;
};

chronological_queue<scheduled_task> queue;
```

Thread Function

```
try
{
    scheduled_task task;
    for (;;)
    {
        queue.pop(task);
        task.function();
        interrupter.check_for_interruption();
        task.time += task.interval;
        queue.push(task);
    }
}
catch (thread_interrupted)
{
}
```

Thread Function

```
try
{
    scheduled_task task;
    for (;;)
    {
        queue.pop(task);
        task.function();
        interrupter.check_for_interruption();
        task.time += task.interval;
        queue.push(task);
    }
}
catch (thread_interrupted)
{
}
```

Must block until next task's time

Thread Function Interruption

```
try
{
    scheduled_task task;
    for (;;)
    {
        queue.pop(task);
        task.function();
        interrupter.check_for_interruption();
        task.time += task.interval;
        queue.push(task);
    }
}
catch (thread_interrupted)
{
}
```

Blocking

Non-blocking Thread Function

```
. . .
while (!queue.try_pop(task))
{
    std::this_thread::yield();
    interrupter.check_for_interruption();
}
task.function();
interrupter.check_for_interruption();
task.time += _interval;
while (!queue.try_push(task))
{
    std::this_thread::yield();
    interrupter.check_for_interruption();
}
. . .
```

Non-blocking Thread Function

```
. . .
queue.pop(task, interrupter);
task.function();
interrupter.check_for_interruption();
task.time += _interval;
queue.push(task, interrupter);
. . .
```

Summary

- Mutexes
- Lock guards
- Condition variables
- Thread-safe queue
- Barriers
- Periodic callable invocation

Questions?

Resources

- http://www.boost.org/libs/thread/index.html
- http://www.stdthread.co.uk/doc/
- http://en.cppreference.com/w/cpp/thread
- *C++ Concurrency in Action: Practical Multithreading* (Williams)

Dealing With std::thread's Destructor Semantics

Thread Destructor

- std::thread’s destructor terminates app if thread joinable
- Must call detach() or join() even when exceptions occur
- Detach only when certain thread is independent
- Join using RAII

Two Options

- Hold reference to std::thread and join, only if joinable, in destructor
- Move std::thread into object
  - Require joinable in constructor
    - Precondition
    - Exception
  - Join in destructor

Option One: thread_guard

```
class thread_guard
{
public:
    explicit thread_guard(std::thread & _thread);

    thread_guard(thread_guard const &) = delete;
    void operator =(thread_guard const &) = delete;

    ~thread_guard();

private:
    std::thread & thread_;
};
```

thread_guard Implementation

```
thread_guard::thread_guard(std::thread & _thread)
    : thread_(_thread)
{ }

thread_guard::~thread_guard()
{
    if (thread_.joinable())
    {
        thread_.join();
    }
}
```

Option Two: scoped_thread

```
class scoped_thread
{
public:
    explicit scoped_thread(std::thread _thread);

    scoped_thread(scoped_thread const &) = delete;
    void operator =(scoped_thread const &) = delete;

    ~scoped_thread();

    std::thread::id get_id() const noexcept;
    std::thread::native_handle_type native_handle();

private:
    std::thread thread_;
};
```

scoped_thread Implementation

```
scoped_thread::scoped_thread(std::thread _thread)
    : thread_(std::move(_thread))
{
    if (!thread_.joinable())
    {
        throw std::logic_error("Thread not joinable");
    }
}

scoped_thread::~scoped_thread()
{
    thread_.join();
}
```

scoped_thread Implementation

```
std::thread::id scoped_thread::get_id() const noexcept
{
    return thread_.get_id();
}

std::thread::native_handle_type scoped_thread::native_handle()
{
    return thread_.native_handle();
}
```