Better Code: Concurrency
Sean Parent | Principal Scientist
Better Code

- Regular Type
  - Goal: Implement Complete and Efficient Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Inheritance
- Concurrency
  - Goal: No Raw Synchronization Primitives
- ...
Better Code

- Regular Type
  - Goal: Implement Complete and Efficient Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Inheritance
- Concurrency
  - Goal: No Raw Synchronization Primitives
- ...
Common Themes

- Manage Relationships
- Understand the Fundamentals
- Code Simply
- Local and Equational Reasoning
Concurrency

- Concurrency: when tasks start, run, and complete in overlapping time periods
- Parallelism: when two or more tasks execute simultaneously

Why?
- Enable performance through parallelism
- Improve interactivity by handling user actions concurrent with processing and IO

http://docs.oracle.com/cd/E39445-01/806-5257/6je9h032b/index.html
What are raw synchronization primitives?

- Synchronization primitives are basic constructs such as:
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Condition Variable
Why No Raw Synchronization Primitives?

You Will Likely Get It Wrong
Problems with Locks

template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; 
    };
    object_t* object_m;

public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};
Problems with Locks

template <typename T>
class bad_cow {
  struct object_t {
    explicit object_t(const T& x) : data_m(x) { ++count_m; }
    atomic<int> count_m;
    T data_m; }
  object_t* object_m;
public:
  explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
  ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
  bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
  bad_cow& operator=(const T& x) {
    if (object_m->count_m == 1) object_m->data_m = x;
    else {
      object_t* tmp = new object_t(x);
      --object_m->count_m;
      object_m = tmp;
    }
    return *this;
  }
};
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m;
    };
    object_t* object_m;
public:
    explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }
    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            --object_m->count_m;
            object_m = tmp;
        }
        return *this;
    }
};

• There is a subtle race condition here:
  • if count != 1 then the bad_cow could also be owned by another thread(s)
  • if the other thread(s) releases the bad_cow between these two atomic operations
  • then our count will fall to zero and we will leak the object
Problems with Locks

template<typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) { ++count_m; }
        atomic<int> count_m;
        T data_m; }; 
    object_t* object_m;

    public:
        explicit bad_cow(const T& x) : object_m(new object_t(x)) { }
    ~bad_cow() { if (0 == --object_m->count_m) delete object_m; }
    bad_cow(const bad_cow& x) : object_m(x.object_m) { ++object_m->count_m; }

    bad_cow& operator=(const T& x) {
        if (object_m->count_m == 1) object_m->data_m = x;
        else {
            object_t* tmp = new object_t(x);
            if (0 == --object_m->count_m) delete object_m;
            object_m = tmp;
        }
        return *this;
    }
};
Why do we want concurrency?

Performance through Parallelism
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX

GFlops
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX

Intrinsics
- Auto-vectorization
- OpenCL

![Graph showing Desktop Compute Power with various technologies and their GFlops rating](chart.png)
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX
- Intrinsics
- Auto-vectorization
- OpenCL
- TBB
- GCD
- OpenMP
- C++11

GFlops
0 750 1500 2250 3000

- GPU
- Vectorization
- Multi-thread
- Scalar

© 2014 Adobe Systems Incorporated. All Rights Reserved.
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX
- Intrinsic
- Auto-vectorization
- OpenCL
- TBB
- GCD
- OpenMP
- C++11
- OpenGL
- OpenCL
- CUDA
- Direct Compute
- C++ AMP
- DirectX
- Intrinsic
- Auto-vectorization
- OpenCL
- TBB
- GCD
- OpenMP
- C++11

0 750 1500 2250 3000

- GPU
- Vectorization
- Multi-thread
- Scalar

(GFlops)
Amdahl’s Law

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

http://en.wikipedia.org/wiki/Amdahl%27s_law
Each line represents 10% more synchronization.
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?

- Thread
- Object
- GO
- STOP
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
STOP
Minimize Locks
Threads and Tasks

- **Thread**: Execution environment consisting of a stack and processor state running in parallel to other threads
- **Task**: A unit of work, often a function, to be executed on a thread
- Tasks are scheduled on a thread pool to optimize machine utilization
C++14 and Tasks

- C++14 does not (really) have a task system
  - Threads
  - Futures

- It is implementation defined if std::async() spins up a thread or executes on a thread pool.
Building a Task System

- Portable Reference Implementation in C++14
- Windows - Window Thread Pool and PPL
- Apple - Grand Central Dispatch (libdispatch)
  - open source, runs on Linux and Android
- Intel TBB - many platforms
  - open source
- HPX - many platforms
  - open source
Building a Task System

Diagram showing the hierarchy of tasks and threads, with arrows indicating the flow from task to thread to core.
using lock_t = unique_lock<mutex>;}
using lock_t = unique_lock<mutex>;

class notification_queue {
    deque<function<void>>() _q;
    mutex _mutex;
    condition_variable _ready;
}
using lock_t = unique_lock<mutex>;

class notification_queue {
    deque<function<void()>> _q;
    mutex _mutex;
    condition_variable _ready;

public:
    void pop(function<void()>& x) {
        lock_t lock(_mutex);
        while (_q.empty()) _ready.wait(lock);
        x = move(_q.front());
        _q.pop_front();
    }
}
using lock_t = unique_lock<mutex>;

class notification_queue {
    deque<function<void()>> _q;
    mutex _mutex;
    condition_variable _ready;

public:
    void pop(function<void()>& x) {
        lock_t lock{_mutex};
        while (_q.empty()) _ready.wait(lock);
        x = move(_q.front());
        _q.pop_front();
    }

    template<typename F>
    void push(F&& f) {
        lock_t lock{_mutex};
        _q.emplace_back(forward<F>(f));
        _ready.notify_one();
    }
};
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;
Building a Task System

class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            _q.pop(f);
            f();
        }
    }
}
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            _q.pop(f);
            f();
        }
    }

public:
    task_system() {
        for (unsigned n = 0; n != _count; ++n) {
            _threads.emplace_back([&, n]{ run(n); });
        }
    }
};
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            _q.pop(f);
            f();
        }
    }

public:
    task_system() {
        for (unsigned n = 0; n != _count; ++n) {
            _threads.emplace_back([&n] { run(n); });
        }
    }

    ~task_system() {
        for (auto & e : _threads) e.join();
    }
}
Building a Task System

class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    notification_queue _q;

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            _q.pop(f);
            f();
        }
    }

    public:
    task_system() {
        for (unsigned n = 0; n != _count; ++n) {
            _threads.emplace_back([&, n]{ run(n); });
        }
    }

    ~task_system() {
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        _q.push(forward<F>(f));
    }
};
Building a Task System

class notification_queue {
    deque<function<void>> _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    void done() {
        unique_lock<mutex> lock{_mutex};
        _done = true;
    }
    _ready.notify_all();
}

bool pop(function<void>&& x) {
    lock_t lock{_mutex};
    while (_q.empty() && !_done) _ready.wait(lock);
    if (_q.empty()) return false;
    x = move(_q.front());
    _q.pop_front();
    return true;
}

template<typename F>
void push(F&& f) {
    lock_t lock{_mutex};
    _q.emplace_back(forward<F>(f));
}
Building a Task System

class notification_queue {
    deque<function<void>>() _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    void done() {
        unique_lock<mutex> lock{_mutex};
        _done = true;
    }

    _ready.notify_all();
}

bool pop(function<void()> & x) {
    lock_t lock{_mutex};
    while (_q.empty() && !_done) _ready.wait(lock);
    if (_q.empty()) return false;
    x = move(_q.front());
    _q.pop_front();
    return true;
}

template<typename F>
void push(F&& f) {
    lock_t lock{_mutex};
    _q.emplace_back(forward<F>(f));
Building a Task System

class notification_queue {
    deque<function<void>>() _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    void done() {
        unique_lock<mutex> lock{_mutex};
        _done = true;
        _ready.notify_all();
    }

    bool pop(function<void>()& x) {
        lock_t lock{_mutex};
        while (_q.empty() && !_done) _ready.wait(lock);
        if (_q.empty()) return false;
        x = move(_q.front());
        _q.pop_front();
        return true;
    }

    template<typename F>
    void push(F&& f) {
        lock_t lock{_mutex};
        _q.emplace_back(forward<F>(f));
    }
}
class notification_queue {
  deque<function<void>>() _q;
  bool _done{false};
  mutex _mutex;
  condition_variable _ready;

public:
  void done() {
    { unique_lock<mutex> lock{_mutex};
      _done = true;
    }
    _ready.notify_all();
  }

  bool pop(function<void>& x) {
    lock_t lock{_mutex};
    while (_q.empty() && !_done) _ready.wait(lock);
    if (_q.empty()) return false;
    x = move(_q.front());
    _q.pop_front();
    return true;
  }

  template<typename F>
  void push(F&& f) {
    lock_t lock{_mutex};
    _q.emplace_back(forward<F>(f));
  }
}
Building a Task System
Building a Task System

Diagram showing the structure of a task system with tasks, threads, and cores.
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
Building a Task System

Task

Task...

Task

Thread

Thread

Thread

Core

Core

Core...Core
Building a Task System

Task

Task

Thread

Thread

Thread

Core

Core

Core
Building a Task System

scheduler

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Thread

Thread

Thread

Thread

Core

Core

Core

Core

...
class task_system {
    const unsigned _count{thread::hardware_concurrency};
    vector<thread> _threads;
    vector<notification_queue> _q{_count};
    atomic<unsigned> _index{0};

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            if (!_q[i].pop(f)) break;
            f();
        }
    }

public:
    task_system() {}

    ~task_system() {
        for (auto& e : _q) e.done();
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
Building a Task System

class task_system {
  const unsigned _count{thread::hardware_concurrency()};
  vector<thread> _threads;
  vector<notification_queue> _q{_count};
  atomic<unsigned> _index{0};

  void run(unsigned i) {
    while (true) {
      function<void()> f;
      if (!_q[i].pop(f)) break;
      f();
    }
  }

public:
  task_system() { }

  ~task_system() {
    for (auto& e : _q) e.done();
    for (auto& e : _threads) e.join();
  }

  template <typename F>
  void async_(F&& f) {
    auto i = _index++;
    _q[i % _count].push(forward<F>(f));
  }
};
Building a Task System

```cpp
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    vector<notification_queue> _q{_count};
    atomic<unsigned> _index{0};

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            if (!_q[i].pop(f)) break;
            f();
        }
    }

    public:
    task_system() {
    }

    ~task_system() {
        for (auto& e : _q) e.done();
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
```
class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    vector<notification_queue> _q{_count};
    atomic<unsigned> _index{0};

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            if (!_q[i].pop(f)) break;
            f();
        }
    }

public:
    task_system() {}

    ~task_system() {
        for (auto& e : _q) e.done();
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
Building a Task System

class task_system {
    const unsigned _count{thread::hardware_concurrency()};
    vector<thread> _threads;
    vector<notification_queue> _q{_count};
    atomic<unsigned> _index{0};

    void run(unsigned i) {
        while (true) {
            function<void()> f;
            if (!_q[i].pop(f)) break;
            f();
        }
    }

public:
    task_system() {}  //

~task_system() {
    for (auto& e : _q) e.done();
    for (auto& e : _threads) e.join();
}

    template<typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
Building a Task System

Scheduler

Task

Task

Task

Task

Task

Task

Task

Task

Thread

Thread

Thread

Core

Core

Core

...
Building a Task System

The diagram shows a hierarchical structure where tasks are scheduled by a scheduler. Each task is linked to a thread and a core. The diagram illustrates how tasks are distributed across multiple cores and threads.
Building a Task System

- Task
  - Scheduler
    - Task
    - Task
    - Task
    - Task
    - Task
    - Task
  - Task Stealing
    - Thread
    - Thread
    - Thread
    - Core
    - Core
    - Core
    - ...
Building a Task System

class notification_queue {
    deque<function<void()>> _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    bool try_pop(function<void()>& x) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock || _q.empty()) return false;
        x = move(_q.front());
        _q.pop_front();
        return true;
    }

    template<typename F>
    bool try_push(F&& f) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock) return false;
        _q.emplace_back(forward<F>(f));
        _ready.notify_one();
        return true;
    }

    void done() {
    }
}
class notification_queue {
    deque<function<void()>> _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

public:
    bool try_pop(function<void()> &x) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock || _q.empty()) return false;
        x = move(_q.front());
        _q.pop_front();
        return true;
    }

    template<typename F>
    bool try_push(F&& f) {
        lock_t lock{_mutex, try_to_lock};
        if (!lock) return false;
        _q.emplace_back(forward<F>(f));
        _ready.notify_one();
        return true;
    }

    void done() {
        { unique_lock<mutex> lock{_mutex};
        }
Building a Task System

```cpp
class notification_queue {
  deque<function<void()>> _q;
  bool _done{false};
  mutex _mutex;
  condition_variable _ready;

public:
  bool try_pop(function<void()>& x) {
    lock_t lock{_mutex, try_to_lock};
    if (!lock || _q.empty()) return false;
    x = move(_q.front());
    _q.pop_front();
    return true;
  }

template<typename F>
  bool try_push(F&& f) {
    lock_t lock{_mutex, try_to_lock};
    if (!lock) return false;
    _q.emplace_back(forward<F>(f));
    _ready.notify_one();
    return true;
  }

  void done() {
    unique_lock<mutex> lock{_mutex};
  }
};
```
void run(unsigned i) {
    while (true) {
        function<void()> f;

        for (unsigned n = 0; n != _count; ++n) {
            if (_q[(i + n) % _count].try_pop(f)) break;
        }
        if (!f && !_q[i].pop(f)) break;

        f();
    }
}

public:
    task_system() { }

    ~task_system() { }

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;

        for (unsigned n = 0; n != _count * K; ++n) {
            if (_q[(i + n) % _count].try_push(forward<F>(f))) return;
        }

        _q[i % _count].push(forward<F>(f));
    }
Building a Task System

```cpp
void run(unsigned i) {
    while (true) {
        function<void()> f;
        for (unsigned n = 0; n != _count; ++n) {
            if (_q[(i + n) % _count].try_pop(f)) break;
        }
        if (!f && !_q[i].pop(f)) break;
        f();
    }
}

public:
    task_system() {}
    ~task_system() {}

    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        for (unsigned n = 0; n != _count * K; ++n) {
            if (_q[(i + n) % _count].try_push(forward<F>(f))) return;
        }
        _q[i % _count].push(forward<F>(f));
    }
```
void run(unsigned i) {
    while (true) {
        function<void()> f;

        for (unsigned n = 0; n != _count; ++n) {
            if (_q[(i + n) % _count].try_pop(f)) break;
        }
        if (!f && !_q[i].pop(f)) break;
        f();
    }
}

public:
    task_system() { }
~task_system() { }

    template<typename F>
    void async_(F&& f) {
        auto i = _index++;

        for (unsigned n = 0; n != _count * K; ++n) {
            if (_q[(i + n) % _count].try_push(forward<F>(f))) return;
        }
        _q[i % _count].push(forward<F>(f));
    }
Building a Task System

Task

Scheduler

Task
Task
Task

Task
Task
Task

Task Stealing

Thread
Thread
Thread

Core
Core
...Core
Building a Task System

![Diagram of a task system with tasks, threads, and cores connected through a scheduler and task stealing mechanism.](image-url)
- Compared to Apple's Grand Central Dispatch (libdispatch)
- Compared to Apple's Grand Central Dispatch (libdispatch)
#include <functional>
#include <future>
#include <type_traits>

#include <dispatch/dispatch.h>

namespace stlab {

    template <class Function, class ... Args>
    auto async(Function&& f, Args&&... args) {
        using result_type = std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>;
        using packaged_type = std::packaged_task<result_type>();

        auto p = new packaged_type(std::bind(std::forward<Function>(f), std::forward<Args>(args)...));
        auto result = p->get_future();

        dispatch_async_f(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
                         p, []() {[
                            packaged_type* f = static_cast<packaged_type*>(f_);
                            (*f)();
                            delete f;
                          });

        return result;
    }

} // namespace stlab
Written with ASIO (Boost 1.62.0)

```cpp
class task_system {
    io_service _service;
    vector<thread> _threads;
    unique_ptr<io_service::work> _work{make_unique<io_service::work>(_service)};

public:
    task_system() {
        for (unsigned n = 0; n != thread::hardware_concurrency(); ++n) {
            _threads.emplace_back([&] {
                _service.run();
            });
        }
    }

    ~task_system() {
        _work.reset();
        for (auto& e : _threads) e.join();
    }

    template <typename F>
    void async_(F&& f) {
        _service.post(forward<F>(f));
    }
};
```
Task System

- Written with ASIO (Boost 1.62.0)

```cpp
class task_system {
  io_service _service;
  vector<thread> _threads;
  unique_ptr<io_service::work> _work{make_unique<io_service::work>(_service)};

public:
  task_system() {
    for (unsigned n = 0; n != thread::hardware_concurrency(); ++n) {
      _threads.emplace_back([&] {
        _service.run();
      });
    }
  }

  ~task_system() {
    _work.reset();
    for (auto& e : _threads) e.join();
  }

  template<typename F>
  void async_(F&& f) {
    _service.post(forward<F>(f));
  }
};
```
No Raw Synchronization Primitives

Task
No Raw Synchronization Primitives
No Raw Synchronization Primitives

Task

Object
No Raw Synchronization Primitives

Task

Object

Task
No Raw Synchronization Primitives
No Raw Synchronization Primitives
No Raw Synchronization Primitives
```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000'000); });

// Do Something

cout << x.get() << endl;
```
Futures

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000'000); });

// Do Something
cout << x.get() << endl;
```

- Fibonacci is often used as an example for parallel algorithms
- Please stop...
template <typename T, typename N, typename O>
T power(T x, N n, O op)
{
  if (n == 0) return identity_element(op);

  while ((n & 1) == 0) {
    n >>= 1;
    x = op(x, x);
  }

  T result = x;
  n >>= 1;
  while (n != 0) {
    x = op(x, x);
    if ((n & 1) != 0) result = op(result, x);
    n >>= 1;
  }

  return result;
}
template <typename T, typename N, typename O>
T power(T x, N n, O op)
{
    if (n == 0) return identity_element(op);

    while ((n & 1) == 0) {
        n >>= 1;
        x = op(x, x);
    }

    T result = x;
    n >>= 1;
    while (n != 0) {
        x = op(x, x);
        if ((n & 1) != 0) result = op(result, x);
        n >>= 1;
    }
    return result;
}

Egyptian Multiplication (Russian Peasant Algorithm)
See “From Mathematics to Generic Programming” - Alex Stepanov and Dan Rose
template <typename N>
struct multiply_2x2 {
    array<N, 4> operator()(const array<N, 4>& x, const array<N, 4>& y) {
        return { x[0] * y[0] + x[1] * y[2], x[0] * y[1] + x[1] * y[3],
};

template <typename N>
array<N, 4> identity_element(const multiply_2x2<N>&) {
    return { N(1), N(0), N(0), N(1) };}
}
template <typename R, typename N>
R fibonacci(N n) {
    if (n == 0) return R(0);
    return power(array<R, 4>{ 1, 1, 1, 0 }, N(n - 1), multiply_2x2<R>())[0];
}
template <typename N>
struct multiply_2x2 {
    array<N, 4> operator()(const array<N, 4>& x, const array<N, 4>& y) {
        return { x[0] * y[0] + x[1] * y[2], x[0] * y[1] + x[1] * y[3],
    }
};

template <typename N>
array<N, 4> identity_element(const multiply_2x2<N>&) { return { N(1), N(0), N(0), N(1) };  }

template <typename R, typename N>
R fibonacci(N n) {
    if (n == 0) return R(0);
    return power(array<R, 4>{ 1, 1, 1, 0 }, N(n - 1), multiply_2x2<R>())[0];
}

\[
\begin{bmatrix}
1 & 1 \\
1 & 0 \\
\end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\
F_n & F_{n-1} \end{bmatrix}
\]
0.72s to calculate
208,988 digits
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000'000); });

// Do Something

cout << x.get() << endl;
f(…)->r
Futures

- Futures allow minimal code transformations to express dependencies
```cpp
future<cpp_int> x = async([]{
    throw runtime_error("failure");
    return fibonacci<cpp_int>(1'000'000);
});

// Do Something

try {
    cout << x.get() << endl;
} catch (const runtime_error& error) {
    cout << error.what() << endl;
}
```
future<cpp_int> x = async([]{
    throw runtime_error("failure");
    return fibonacci<cpp_int>(1'000'000);
});

// Do Something

try {
    cout << x.get() << endl;
} catch (const runtime_error& error) {
    cout << error.what() << endl;
}

failure
No Raw Synchronization Primitives

Task

Args
No Raw Synchronization Primitives
No Raw Synchronization Primitives
No Raw Synchronization Primitives

```
Task

future

...

future.get()

Args

Task

...
```
No Raw Synchronization Primitives
No Raw Synchronization Primitives

```
Task

future

...

future.get()

Result
```
Futures: What year is this?

- C++14 futures lack:
  - Continuations - `.then()`
  - Joins - `when_all()`
  - Split
  - Cancelation
  - Progress Monitoring (Except Ready)

- And C++14 futures don't compose (easily) to add these features
Futures: Continuations

- Blocking on \texttt{std::future.get()} has two problems
  - One thread resource is consumed, increasing contention
  - Possibly causing a deadlock in our tasking system!
  - Any subsequent non-dependent calculations on the task are also blocked

- C++14 doesn’t have continuations
  - GCD has serialized queues and groups
  - PPL has chained tasks
  - TBB has flow graphs
  - TS Concurrency will have \texttt{.then()} 
  - Boost futures have them now
Futures: get() deadlock
Futures: get() deadlock
Futures: get() deadlock

![Diagram of a task and stop state](image-url)
Futures: `get()` deadlock

![Diagram of task deadlock](image-url)
Futures: get() deadlock
Futures: `get()` deadlock

![Diagram showing Futures deadlock](image)
Futures: `get()` deadlock
Futures: get() deadlock
Futures: Continuations

- Blocking on `std::future::get()`
  - Very difficult to use safely with a thread pool
- C++14 allows `std::async()` to use a thread pool

- Not just `get()` - *any* blocking (condition variables, wait, …) is problematic with a task system
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000); });

future<void> y = x.then([](future<cpp_int> x){ cout << x.get() << endl; });

// Do something
y.wait();
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000); });
future<void> y = x.then([](future<cpp_int> x){ cout << x.get() << endl; });

// Do something
y.wait();

43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209
689623239873322471161642996440906533187938298969649928516003704476137795166849228875
Futures vs Completion Handlers

- Completion handlers are callbacks, they must be known prior to the call
  - No need to synchronize between invoking and setting the continuation

- Futures allow setting the continuation after the sending call is in flight
  - Simpler to compose
  - Require synchronization between invoking and setting the continuation
Futures: Joins
Task Systems
auto x = async([]{ return fibonacci<cpp_int>(1'000'000); });
auto y = async([]{ return fibonacci<cpp_int>(2'000'000); });

auto z = when_all(std::move(x), std::move(y)).then([](auto f){
    auto t = f.get();
    return cpp_int(get<0>(t).get() * get<1>(t).get());
});

cout << z.get() << endl;
Futures: Continuations

auto x = async([]{ return fibonacci<cpp_int>(1'000'000); });
auto y = async([]{ return fibonacci<cpp_int>(2'000'000); });

auto z = when_all(std::move(x), std::move(y)).then([](auto f){
    auto t = f.get();
    return cpp_int(get<0>(t).get() * get<1>(t).get());
});

cout << z.get() << endl;

f is a future tuple of futures
Futures: Continuations

```cpp
auto x = async([]{ return fibonacci<cpp_int>(1'000'000); });
auto y = async([]{ return fibonacci<cpp_int>(2'000'000); });

auto z = when_all(std::move(x), std::move(y)).then([](auto f){
    auto t = f.get();
    return cpp_int(get<0>(t).get() * get<1>(t).get());
});

cout << z.get() << endl;
```

*f is a future tuple of futures*

*result is 626,964 digits*
```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = x.then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
```
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });
future<cpp_int> y = x.then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
```

Assertion failed: (px != 0), function operator->, file shared_ptr.hpp, line 648.
Continuations

- Desired behavior
  - A future should behave as a *regular* type - a token for the actual value
    - shared_futures let me “copy” them around and do multiple get() operations
  - But not multiple continuations
Continuations

- We can write a pseudo-copy, split().
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = split(x).then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
```
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = split(x).then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });

future<void> done = when_all(std::move(y), std::move(z)).then([](auto f){
    auto t = f.get();
    cout << get<0>(t).get() << endl;
    cout << get<1>(t).get() << endl;
});

done.wait();
Futures: Continuations

```cpp
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(100); });

future<cpp_int> y = split(x).then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });

future<void> done = when_all(std::move(y), std::move(z)).then([](auto f){
    auto t = f.get();
    cout << get<0>(t).get() << endl;
    cout << get<1>(t).get() << endl;
});

done.wait();
```

708449696358523830150
23614989878617461005
Building Blocks

- Promise is the sending side of a future
- Promises are packaged with a function to form a packaged task
  - Packaged tasks handle the exception marshalling through a promise
```cpp
promise<int> x;
future<int> y = x.get_future();

x.set_value(42);
cout << y.get() << endl;
```
```cpp
promise<int> x;
future<int> y = x.get_future();

x.set_value(42);
cout << y.get() << endl;
```

42
template <typename T> 
auto split(future<T>& x) {

    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
Futures: Split

```cpp
template <typename T>
auto split(future<T>& x) {
    auto tmp = std::move(x);
    promise<T> p;
    x = p.get_future(); // replace x with new future
    return tmp.then([&p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
```
template<typename T>
auto split(future<T>& x) {
    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
template <typename T>
auto split(future<T>& x) {
    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
template <typename T>
auto split(future<T>& x) {
    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
Template <typename T>

```cpp
auto split(future<T>& x) {
    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
```
template<typename T>
auto split(future<T>& x) {

    auto tmp = std::move(x);

    promise<T> p;
    x = p.get_future(); // replace x with new future

    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
Futures: Split

```cpp
template <typename T>
auto split(future<T>& x) {

    auto tmp = std::move(x);

promise<T> p;
    x = p.get_future(); // replace x with new future

return tmp.then([&p = std::move(p)](auto _tmp) mutable {
    if (_tmp.has_exception()) {
        auto error = _tmp.get_exception_ptr();
        _p.set_exception(error);
        rethrow_exception(error);
    }

    auto value = _tmp.get();
    _p.set_value(value); // assign to new "x" future
    return value; // return value through future result
});
}
```
future<
cpp_int>
x = async([]{
  return fibonacci<
cpp_int>(100);
});

future<
cpp_int>
y = split(x).then([](future<
cpp_int>
x){
  return cpp_int(x.get() * 2);
});
future<
cpp_int>
z = x.then([](future<
cpp_int>
x){
  return cpp_int(x.get() / 15);
});

future<void>
done = when_all(std::move(y), std::move(z)).then([](auto f){
  auto t = f.get();
  cout << get<0>(t).get() << endl;
  cout << get<1>(t).get() << endl;
});
done.wait();

708449696358523830150
23614989878617461005
Cancelation

- When the (last) future destructs
  - The associated task that has not started, should not execute (NOP)
  - The resource held by that task should be released
- Since that task may hold futures for other tasks, the system unravels
Cancellation

- When the (last) future destructs
  - The associated task that has not started, should not execute (NOP)
  - The resource held by that task should be released
    - Since that task may hold futures for other tasks, the system unravels

- I do not know of a good way to compose such cancelation with current futures
  - Except to create something more complex than re-implementing futures
Cancelation
Channels
What if we persist the graph?
What if we persist the graph?

- Allow multiple invocations of the tasks by setting the source values
- Each change triggers a notification to the sink values
  - This is a reactive programming model and futures are known as **behaviors** or **channels**
Accumulators and Generator

- Each operation does not have to be a 1:1 mapping of input to output
- Coroutines are one way to write n:m functions
Futures: Continuations

```cpp
channel<int> send;

auto hold = send
    | [](const receiver<int>& r) {
        int sum = 0;
        while(auto v = co_await r) {
            sum += v.get();
        }
        return sum;
    }
    | [](int x){ cout << x << '\n'; };

send(1);
send(2);
send(3);
send.close();
```
Futures: Continuations

```cpp
channel<int> send;

auto hold = send
    | [](const receiver<int>& r) {
        int sum = 0;
        while(auto v = co_await r) {
            sum += v.get();
        }
        return sum;
    }
    | [](int x) { cout << x << 'n'; };

send(1);
send(2);
send(3);
send.close();
```
Flow Control
Flow Control
Flow Control
Flow Control
Flow Control
Property Models
How do the graphs change during execution?
Property Model

\[ R\{a, b, c\} \]

 Flowchart of the property model with nodes labeled as follows:
- Node 'c'
- Node 'a'
- Node 'b'

Connections between the nodes:
- 'c' is connected to 'R\{a, b, c\}'
- 'a' is connected to 'R\{a, b, c\}'
- 'R\{a, b, c\}' is connected to 'b'
- 'b' is connected to another node (not labeled in the image)
A function is a directed relationship

- We can remove the arrows by providing a package of functions to represent the relationship
  - \[ a = b \times c \]
  - \[ b = a / c \]
  - \[ c = a / b \]
- This forms a type of constraint system called a *property model*
- Flow is determined by value, or *cell*, priority
- Relationships can be conditional, so long as predicate can be determined regardless of flow
- Cells can only have one in-edge for a given flow or the system is over constrained
Reflowing a property model doesn’t require all relationships to be resolved
- The task representing them can still be executing concurrently
- This creates a single dependency graph that is appended to for each new flow and is pruned and ***unravels*** as tasks are complete
- Value set in source A then B
- Operation connected to B is high latency
- Value is set in source C causing reflow
  - Sink X is no longer needed, pending operation is canceled
- Source A is discarded
- Intermediate value I is shared between flows, once determined
- Final values determined by source B (via I) and source C
Property Model

- Source C
- Sink
- Sink
- Source B
Property Models

- Very useful for UI behavior

- Significant information is in the graph
  - Source / Derived values form a partition set
    - Easily model checked
  - Equal result regardless of source order
    - Form an operational transform, useful for collaborative editing
  - A value is *implied* by the current state it only has in in-edge
  - Source values determine *intent*
  - Values disconnected from result (sink) are *don't care*
Final Thoughts

- Perhaps representing such systems as if it were imperative code is not the correct approach
- Instead the a graph description can be compiled and statically validated

- Slides and code from talk:
  

- Experimental future library:
  
  - [https://github.com/stlab/libraries/tree/develop](https://github.com/stlab/libraries/tree/develop)
No raw synchronization primitives
No raw synchronization primitives

Communicating Sequential Tasks
No raw synchronization primitives

Communicating Sequential Tasks

Better Code