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/E19455-01/806-5237/1qje9532b/index.html
Goal: No Raw Synchronization Primitives
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
template<typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) {}
        atomic<int> count_m{1};
        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) {}
        atomic<int> count_m{1};
        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) {}
    atomic<int> count_m{1};
    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
template <typename T>
class bad_cow {
    struct object_t {
        explicit object_t(const T& x) : data_m(x) {} 
        atomic<int> count_m{1};
        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; 
    } 
};
Problems with Locks

- `bad_cow` is not an atomic type, `bad_cow<int>` is as thread safe as `int`
- `--x` on an atomic is equivalent to `atomic_fetch_sub(x) - 1`
Problems with Locks

- `bad_cow` is not an atomic type, `bad_cow<int>` is as thread safe as `int`
- `--x` on an atomic is equivalent to `atomic_fetch_sub(x) - 1`

- Nobody caught the bug that `count_m` was uninitialized
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)

GPU  Vectorization  Multi-thread  Scalar

0  750  1500  2250  3000 (GFlops)
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

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

Diagram showing GPU, Vectorization, Multi-thread, and Scalar performance in 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 labels for different technologies and their performance metrics in GFlops.](image-url)
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
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
- Straight C++
Amdahl’s Law

\[ S(N) = \frac{1}{1 - P} + \frac{P}{N} \]
Amdahl's Law

Each line represents 10% more synchronization
Why No Raw Synchronization Primitives?

Object

thread

thread

thread
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
Minimize Locks
class registry {
    mutex _mutex;
    unordered_map<string, string> _map;
public:
    void set(string key, string value) {
        unique_lock<mutex> lock(mutex);
        _map.emplace(move(key), move(value));
    }

    auto get(const string& key) -> string {
        unique_lock<mutex> lock(mutex);
        return _map.at(key);
    }
};
“It can be shown that programs that correctly use mutexes and memory_order_seq_cst operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as ‘sequential consistency.’"

– C++11 Standard 1.10.21
Mutexes and Sequential Consistency

\[
\begin{align*}
\text{Op}_4(X) & \rightarrow r_m \\
\vdots & \\
\text{Op}_3(X) & \rightarrow r_2 \\
\text{Op}_1(X) & \rightarrow r_1 \\
\text{Op}_3(X) & \rightarrow r_0
\end{align*}
\]
Mutexes and Sequential Consistency

- A mutex serializes a set of operations, $Op_n$, where the operation is the code executed while the mutex is locked.
- Operations are interleaved and may be executed in any order and may be repeated.
- Each operation takes an argument, $X$, which is the set of all objects mutated under all operations.
  - $X$ may not be safely read or written without holding the lock if it may be modified by a task holding the lock.
- Each operation may yield a result, $r_m$, which can communicate information about the state of $X$ while it's associated operation was executed.

- The same is true of all atomic operations.
class registry {
    serial_queue _q;

    using map_t = unordered_map<string, string>;

    shared_ptr<map_t> _map = make_shared<map_t>();

public:
    void set(string key, string value) {
        _q.async([&_map = _map](string key, string value) {
            _map->emplace(move(key), move(value));
        }, move(key), move(value));
    }

    auto get(string key) -> future<string> {
        return _q.async([&_map = _map](string key) {
            return _map->at(key);
        }, move(key));
    }
};
class registry {
    serial_queue _q;

    using map_t = unordered_map<string, string>;

    shared_ptr<map_t> _map = make_shared<map_t>();

public:
    void set(string key, string value) {
        _q.async([_map = _map](string key, string value) {
            _map->emplace(move(key), move(value));
        }, move(key), move(value));
    }

    auto get(string key) -> future<string> {
        return _q.async([_map = _map](string key) {
            return _map->at(key);
        }, move(key));
    }

    void set(vector<pair<string, string>> sequence) {
        _q.async([_map = _map](vector<pair<string, string>> sequence) {
            _map->insert(make_move_iterator(begin(sequence)), make_move_iterator(end(sequence)));
        }, move(sequence));
    }
};
The transformation mutex to serial queue places an upper-bound
- Synchronization overhead
- Time to issue operation
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

Task

Task...

Task

Thread Thread Thread

Core Core ...

Core
Building a Task System
using lock_t = unique_lock<mutex>;
Building a Task System

```cpp
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();
    }
};
Building a Task System
class task_system {
    const unsigned
    vector<thread>
    notification_queue
    _count{thread::hardware_concurrency()};
    _threads;
    _q;
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));
}
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));
        }
    }
}
Building a Task System
Building a Task System

Task

Task

Task

Thread

Thread

Thread

Core

Core

Core

...
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?
Why No Raw Synchronization Primitives?

Object

STOP

thread

GO

thread

thread

STOP
Building a Task System
Building a Task System
Building a Task System

![Diagram of a task system with Task, Scheduler, Thread, and Core components.]
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

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

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

Task

Scheduler

Task
Task
Task

Task
Task
Task

Thread
Thread
Thread

Core
Core
...Core
Building a Task System
Building a Task System

Diagram showing the flow of tasks through a task system, starting with tasks being scheduled, then distributed among threads, and finally stolen by other threads. The diagram also shows the relationship between tasks and cores.
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};
    }
}
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};
    }
}
```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();
    }
}

class task_system {
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

![Diagram of a task system with tasks, schedulers, and cores](image-url)
Building a Task System

Scheduler

Task

Task

Task

Task

Task

Task

Task Stealing

Thread

Thread

Thread

Core

Core

...
• Compared to Apple's Grand Central Dispatch (libdispatch)
• Compared to Apple's Grand Central Dispatch (libdispatch)
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([_f = std::forward<Function>(f)](Args&... args) {
    return _f(std::move(args)...);
}, std::forward<Args>(args)...));

auto result = _p->get_future();

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

return result;
}
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));
  }
};
```
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
No Raw Synchronization Primitives
No Raw Synchronization Primitives

Task

Object
No Raw Synchronization Primitives
No Raw Synchronization Primitives

Task

Task

Object
No Raw Synchronization Primitives
No Raw Synchronization Primitives
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 {
        };
    }
};

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];
}

\[
\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}
\]
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
f(…)

r
• Futures allow minimal code transformations to express dependencies
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
No Raw Synchronization Primitives
No Raw Synchronization Primitives
No Raw Synchronization Primitives

```
Task

future

... future.get()

Task

Args

... Task
```

© 2013 Adobe Systems Incorporated. All Rights Reserved.
No Raw Synchronization Primitives

Task

future

...

future.get()

Result
Futures: What year is this?

- C++14 futures lack:
  - Continuations - \texttt{.then()}
  - Joins - \texttt{when\_all()}
  - Split
  - Cancelation
  - Progress Monitoring (Except Ready)

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

- Blocking on `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 `.then()`
  - Boost futures have them now
Futures: get() deadlock
Futures: get() deadlock

Task

STOP
Futures: get() deadlock

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

```
 Task
   .
   .
   .

 STOP
```

```
 Task
   .
   .
   .
```
Futures: `get()` deadlock

![Diagram showing the deadlocked state of futures with tasks and a stop condition](image)
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
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* conditional blocking (condition variables, wait, …) is problematic with a task system
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* conditional blocking (condition variables, wait, …) is problematic with a task system

Do call `std::future.get()` or `std::future.wait()` when the originating task, or any subordinate task, is on the same queue, even if it is a concurrent queue (i.e. a thread pool).
**Important:** You should never call the `dispatch_sync` or `dispatch_sync_f` function from a task that is executing in the same queue that you are planning to pass to the function. This is particularly important for serial queues, which are guaranteed to deadlock, but should also be avoided for concurrent queues.

Futures: Continuations

```cpp
defibonacci\(\text{cpp\_int}\)\(\text{1'000}\);\)

```

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

```cpp
future\<\text{void}\> y = x.then([](future<\text{cpp\_int}\> x){ cout \(\ll\) x.get() \(\ll\) endl; });
```

// Do something

```cpp
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();
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
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;
```
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
```
Futures: Split
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); });
```
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
• We can write a pseudo-copy, split().
Futures: Continuations
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); });
```
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();
```
```cpp
template<>
future<cpp_int> fibo(cpp_int n)
{
    if (n <= 1)
        return n;
    else
        return fibo(n - 1) + fibo(n - 2);
}

future<cpp_int> x = async([]{
    return fibo<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();
```

```
• Promise is the sending side of a future
• Promises are packaged with a function to formed 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;
```
Promise

```cpp
promise<int> x;
future<int> y = x.get_future();

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

42
Futures: Split

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

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
    });
}
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>
auto split(future<T>& x) {

    auto tmp = std::move(x);

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

    return std::move(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
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
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
stlab future library

- Currently supports
  - Multiple continuations and copy
    - Optimized for rvalues
  - Join (When All, When Any)
  - Cancelation on Destruction (and explicit reset)
    - And detach

- [https://github.com/stlab/libraries/tree/develop](https://github.com/stlab/libraries/tree/develop)
- Thanks to Felix Petriconi
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
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();
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();
struct sum {
    process_state_scheduled _state = await_forever;
    int _sum = 0;

    void await(int n) { _sum += n; }

    int yield() { _state = await_forever; return _sum; }

    void close() { _state = yield_immediate; }

    const auto& state() const { return _state; }
};
Flow Control
Flow Control
Flow Control

Diagram showing sequence of flow control elements.
Flow Control
Flow Control
struct render {
    process_state_scheduled _state = await_forever;
    bool _final = false;
    parameters _params;

    void await(parameters params) {
        _final = false;
        _state = await_immediate;
        _params = params;
    }

    frame yield() {
        auto result = render_frame(_params, _final);
        _final = !_final;
        _state = _final ? await_immediate : await_forever;
        return result;
    }

    void close() { if (_state == await_immediate) _state = yield_immediate; }

    const auto& state() const {
        return _state;
    }
};
Final Thoughts

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

- Slides and code from talk:

- Experimental future and channel library:
  - [https://github.com/stlab/libraries/tree/develop](https://github.com/stlab/libraries/tree/develop)
  - Thanks to Felix Petriconi

- Communicating Sequential Processes (C. A. R. Hoare)