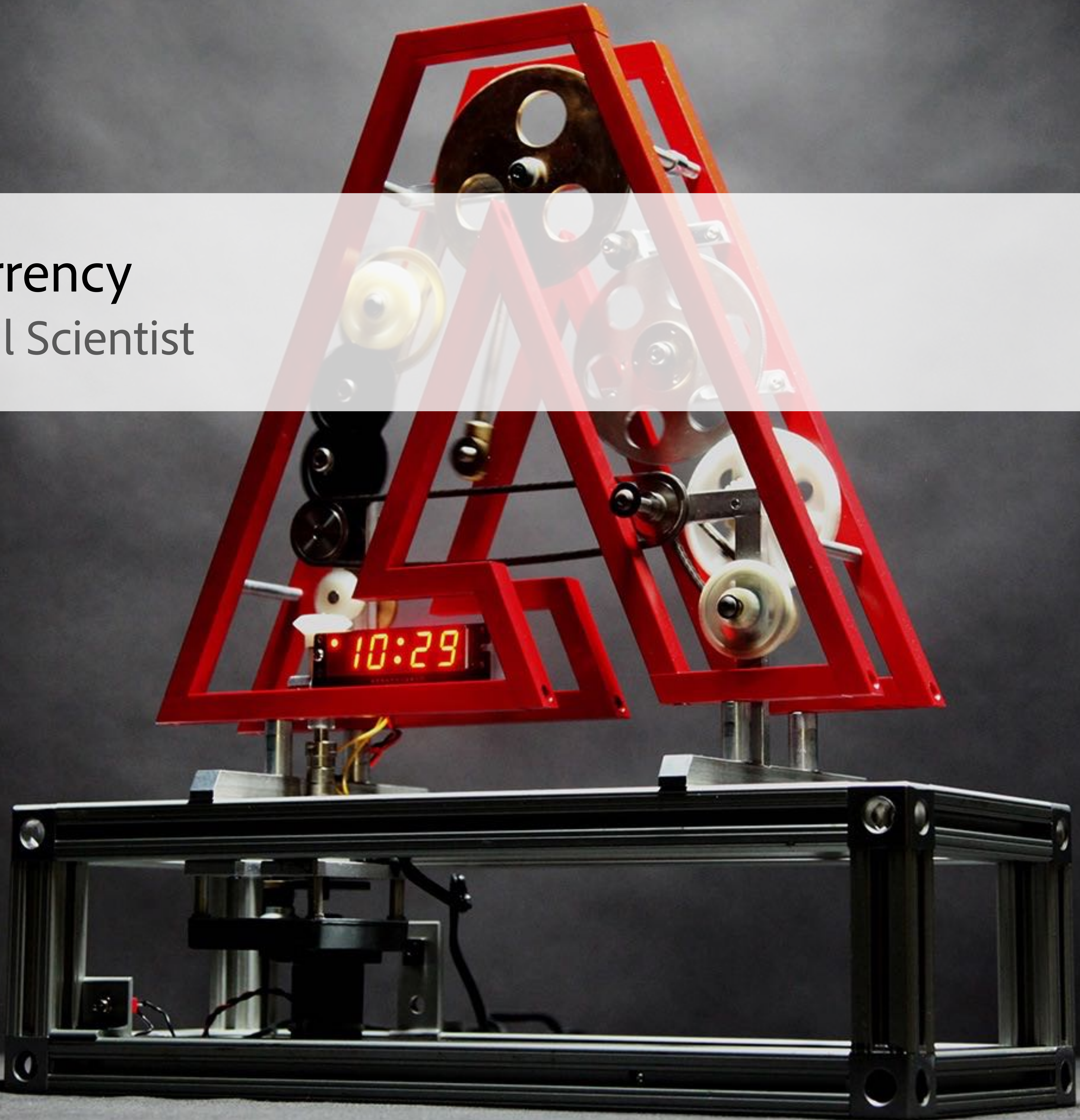
Better Code: Concurrency

Sean Parent | Principal Scientist



Bē
Evolution Bureau

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
- …

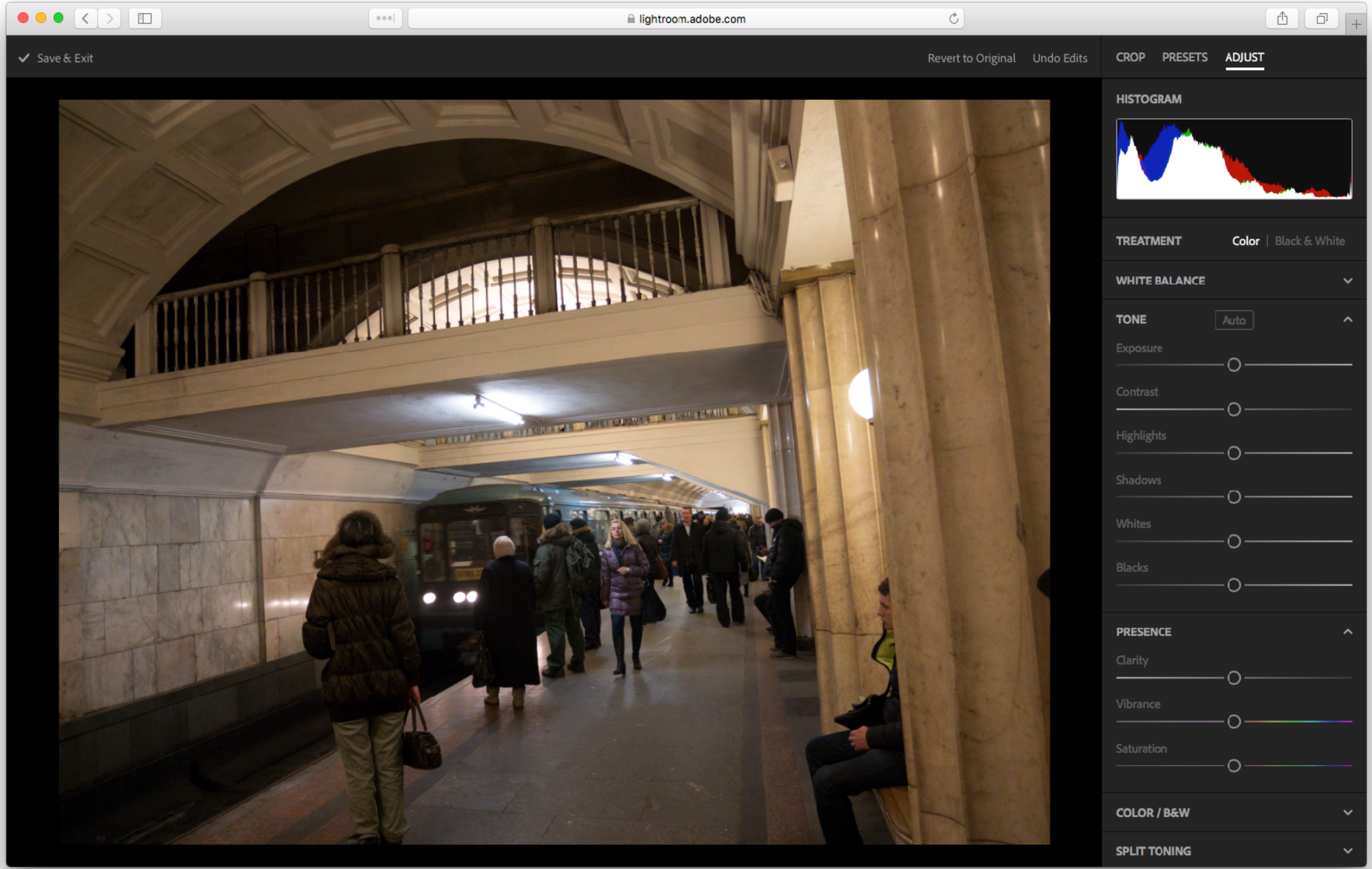
Better Code

- 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


lightroom.adobe.com
Exit
Revert to Original
Undo Edits
CROP
PRESETS
ADJUST
HISTOGRAM
TREATMENT
Color
Black & White
WHITE BALANCE
TONE
Auto
Exposure
Contrast
Highlights
Shadows
Whites
Blacks
PRESENCE
Clarity
COLOR / B&W
SPLIT TONING


lightroom.adobe.com
Save & Exit
Revert to Original
Undo Edits
CROP
PRESETS
ADJUST
HISTOGRAM
TREATMENT
Color
Black & White
WHITE BALANCE
TONE
Auto
Exposure
Contrast
Highlights
Shadows
Whites
Blacks
PRESENCE
Clarity
Vibrance
Saturation
COLOR / B&W
SPLIT TONING

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/E19455-01/806-5257/6je9h032b/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

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) {}
        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;
    }
};
```

Problems with Locks

```
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;
    }
};
```

Problems with Locks

```
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 is 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) {}
        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

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

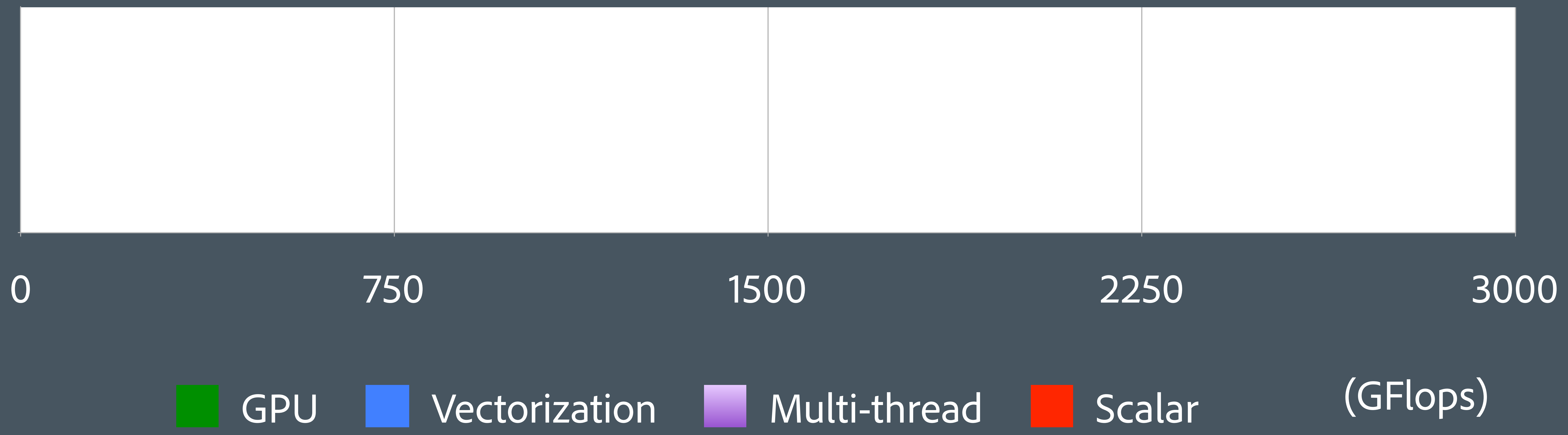
- `bad_cow` is not an atomic type, `bad_cow<int>` is as thread safe as `int`
- `--x` on an atomic is equivalent to `atomic_fetch_sub(x) - 1`

- Nobody caught the bug that `count_m` was uninitialized

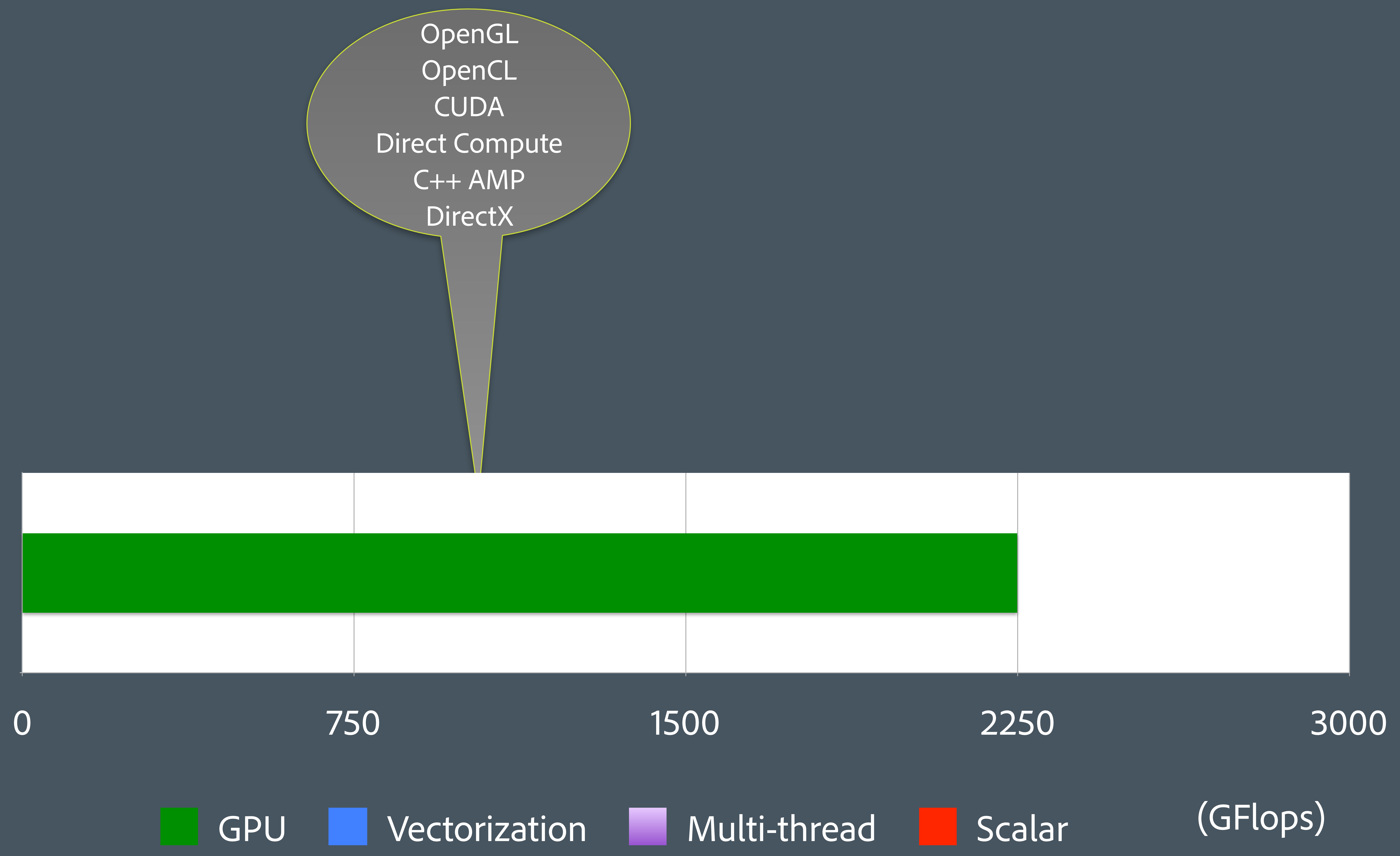
Performance through Parallelism

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

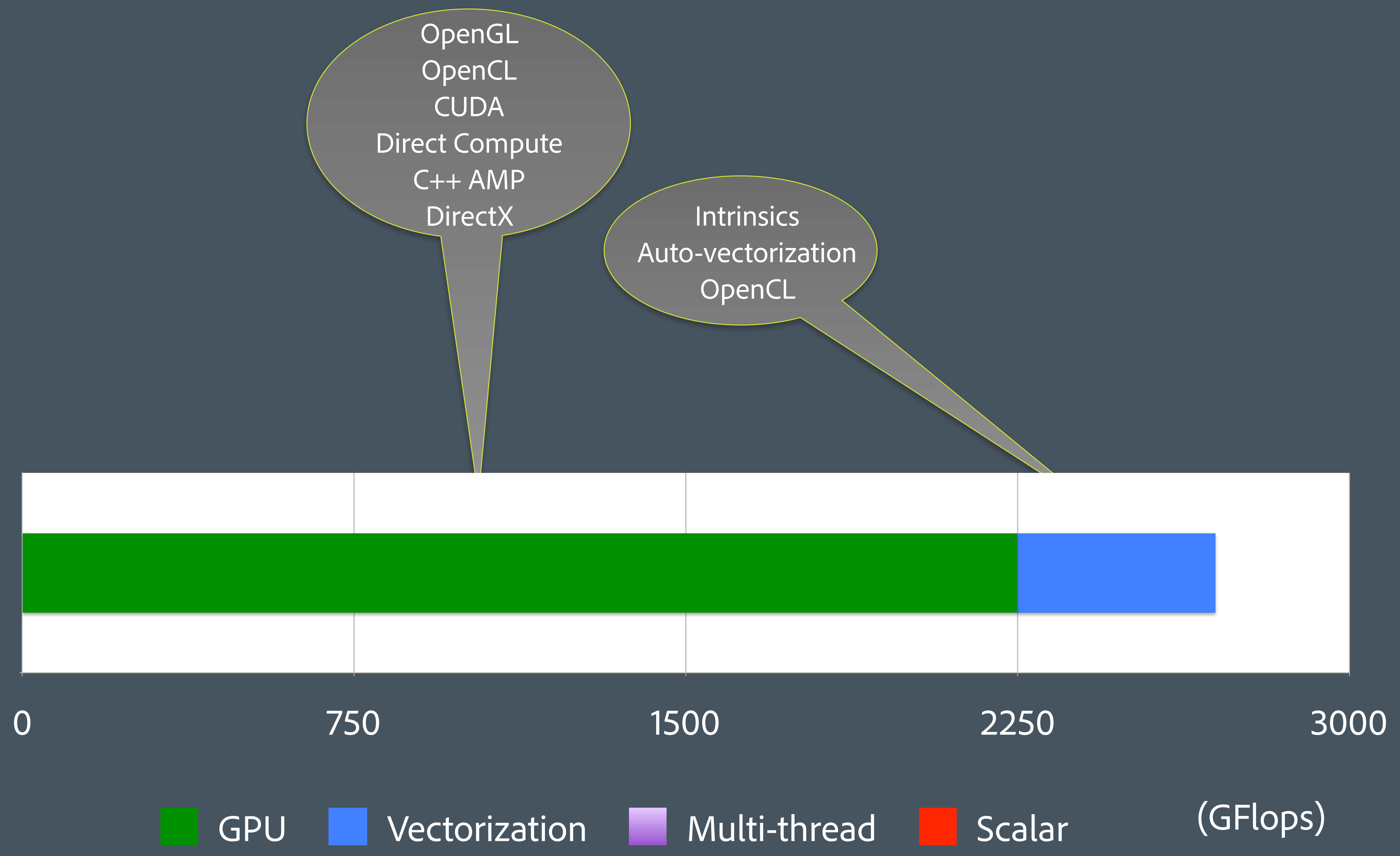
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



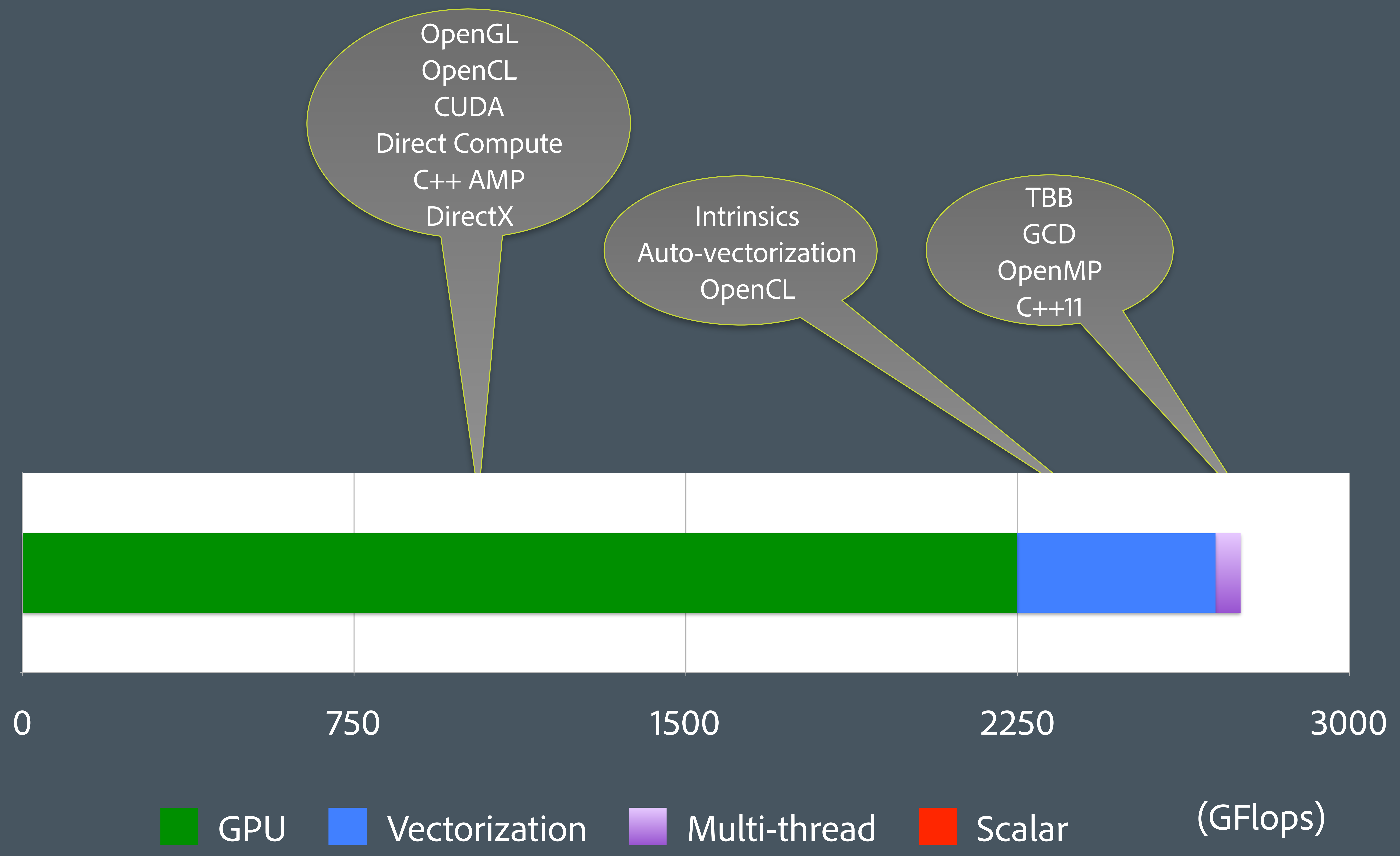
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



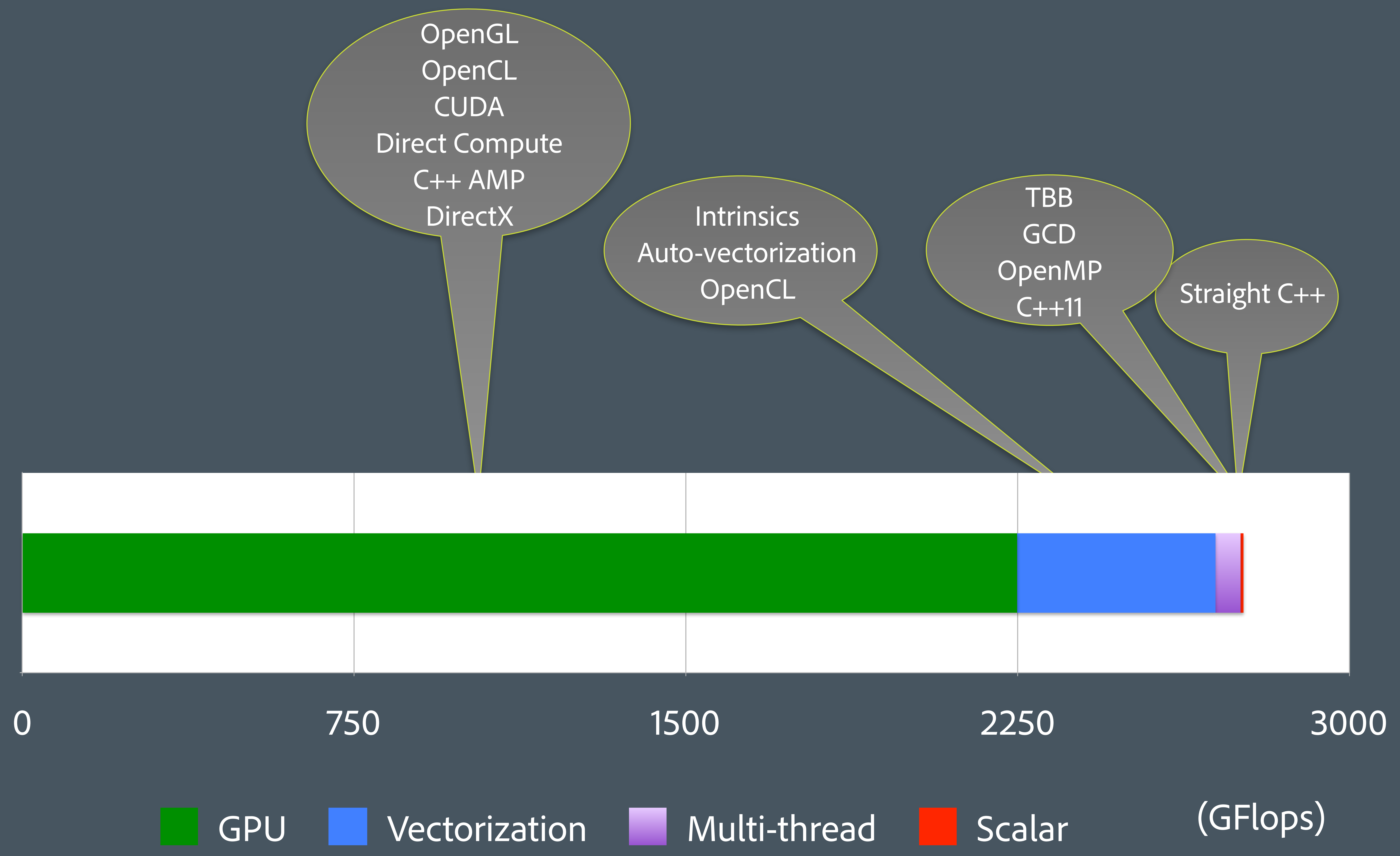
Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



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

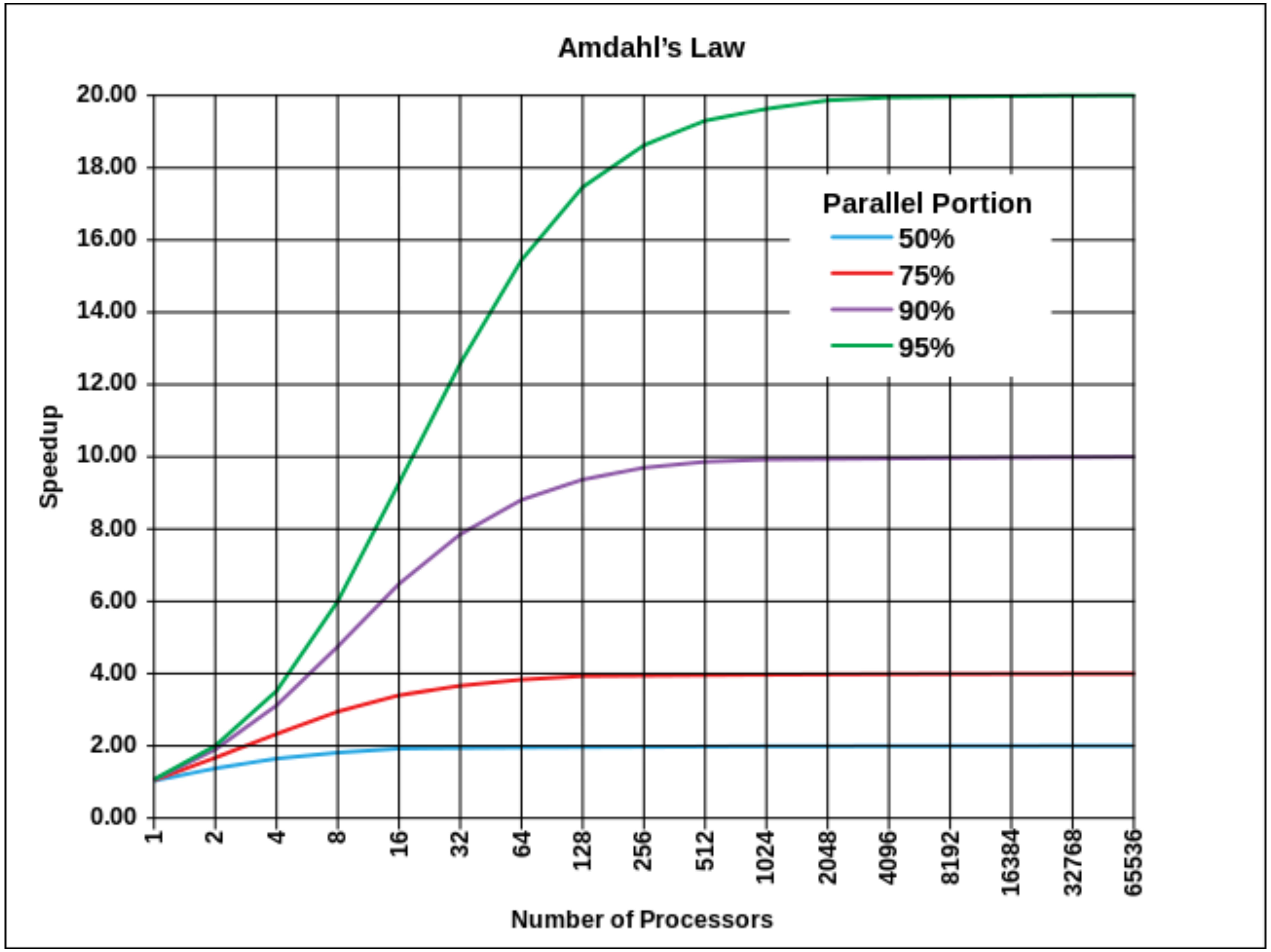


Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



Amdahl's Law

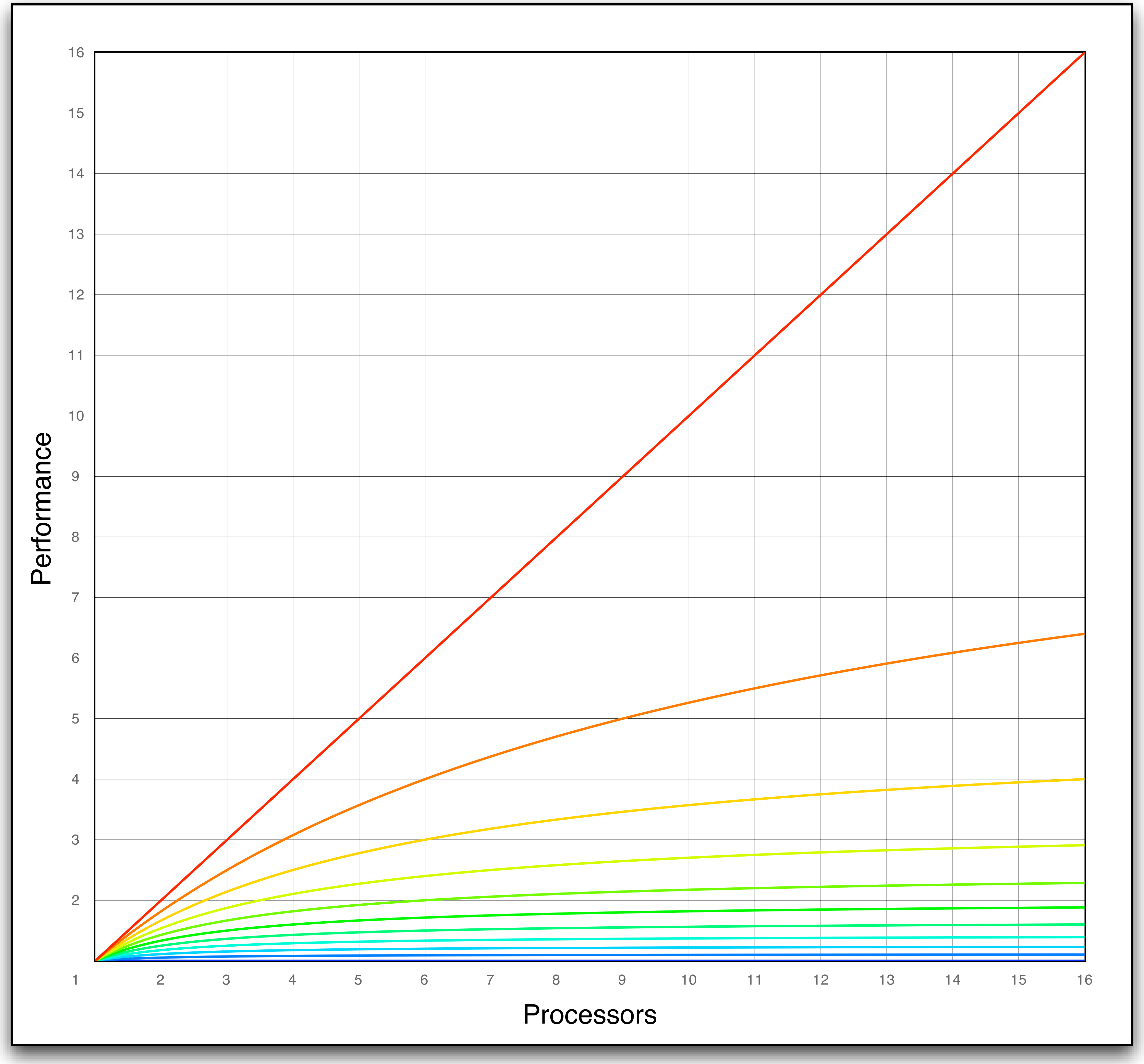
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$



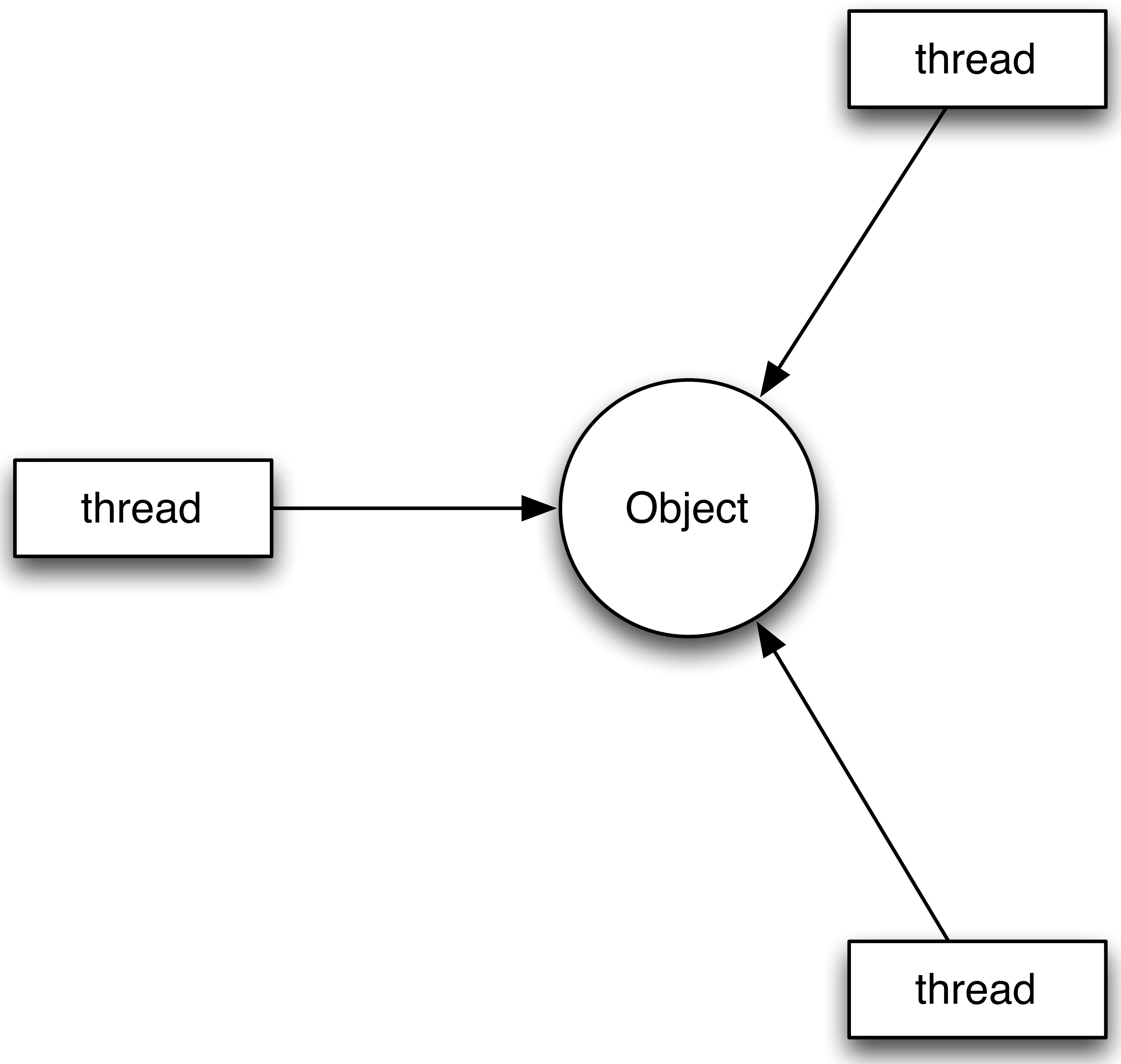
http://en.wikipedia.org/wiki/Amdahl%27s_law

Amdahl's Law

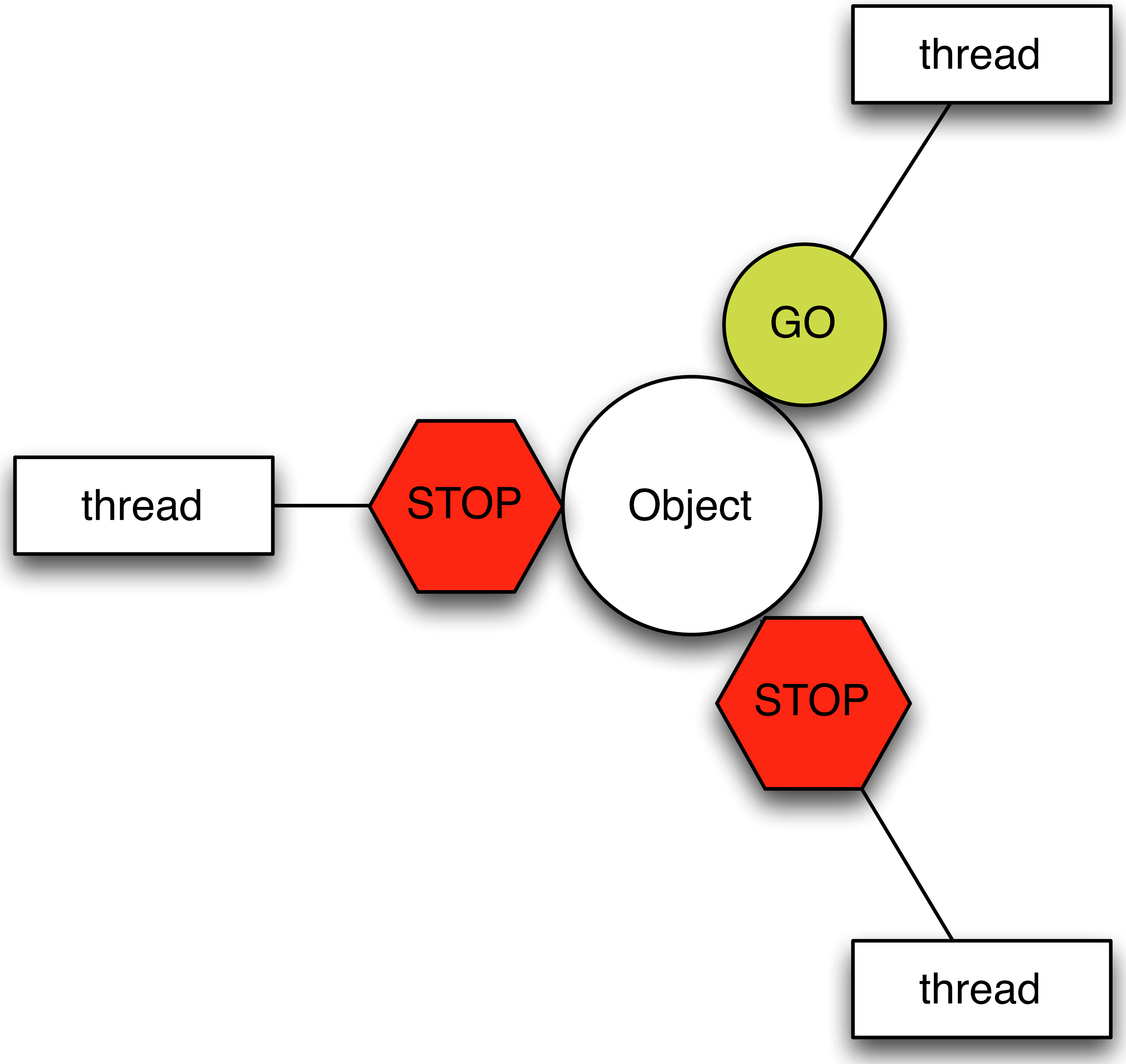
Each line represents 10% more synchronization



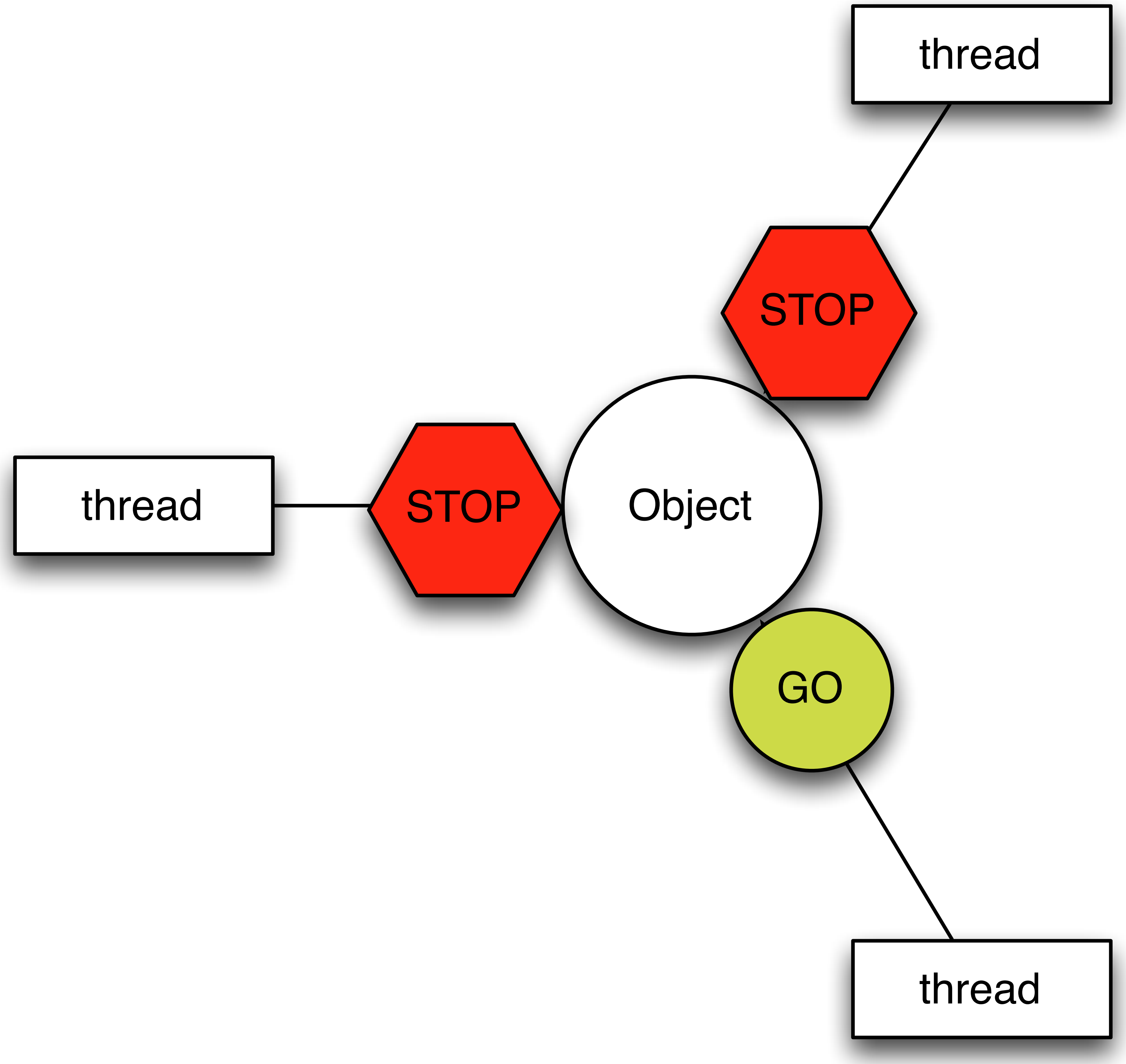
Why No Raw Synchronization Primitives?



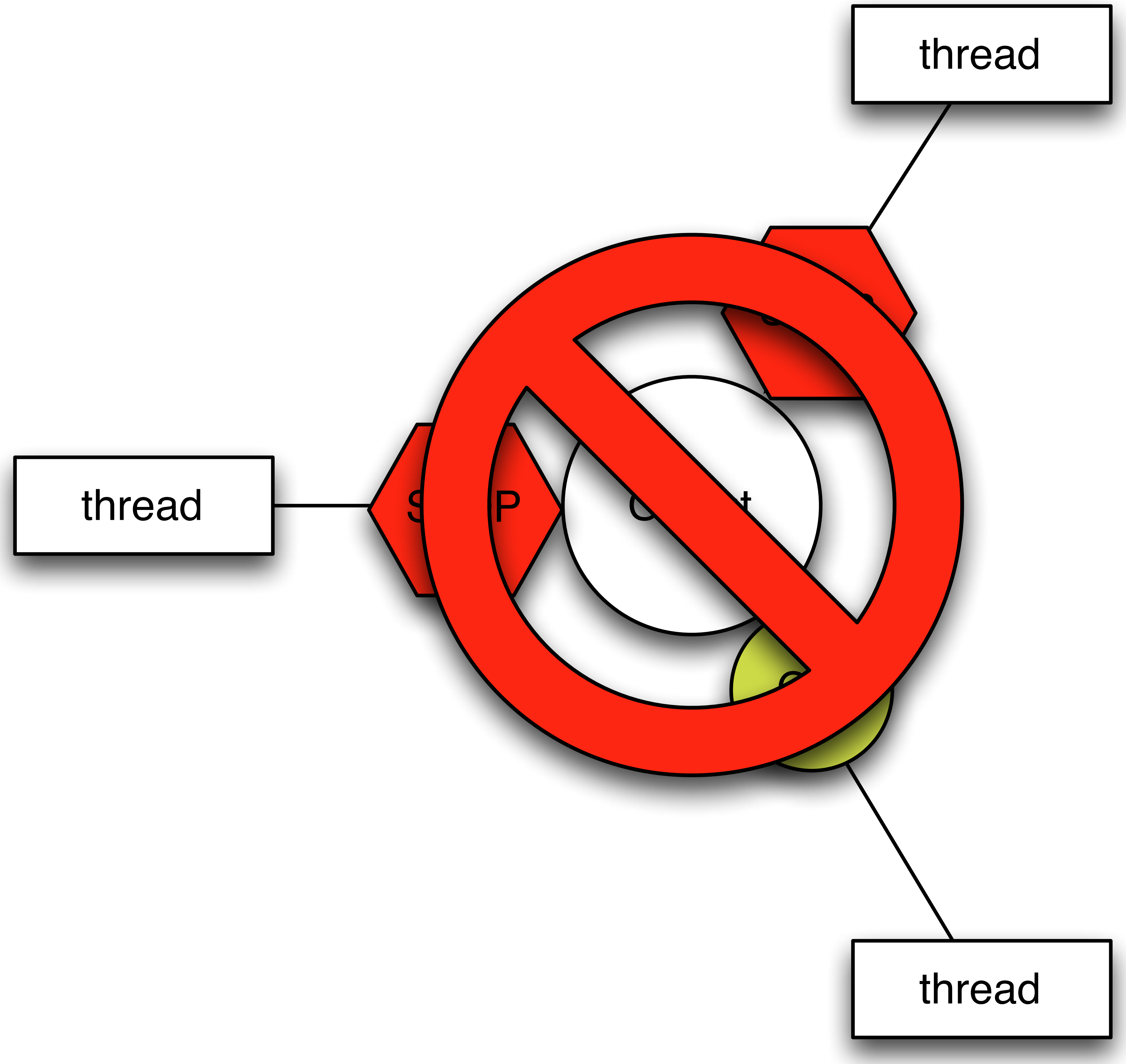
Why No Raw Synchronization Primitives?



Why No Raw Synchronization Primitives?



Why No Raw Synchronization Primitives?



Minimize Locks

Minimize Locks

Mutexes and Sequential Consistency

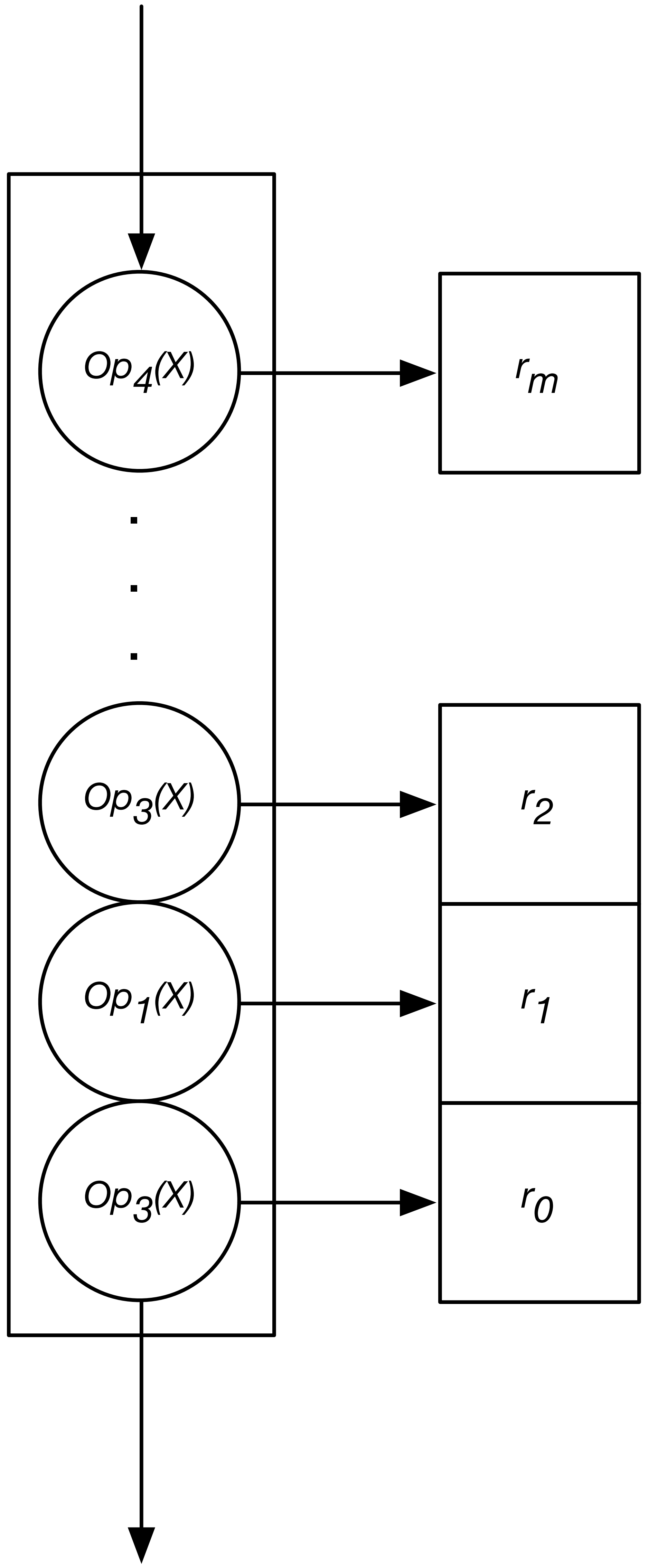
```
class registry {
    mutex _mutex;
    unordered_map<string, string> _map;
  public:
    void set(string key, string value) {
        unique_lock<mutex> lock(mutex);
        _map.emplace(move(key), move(value));
    }

    auto get(const string& key) -> string {
        unique_lock<mutex> lock(mutex);
        return _map.at(key);
    }
};
```

Mutexes and Sequential Consistency

"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



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

Mutexes and Sequential Consistency

```
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));
    }
};
```

Mutexes and Sequential Consistency

```
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));
    }
};
```

Mutexes and Sequential Consistency

- 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

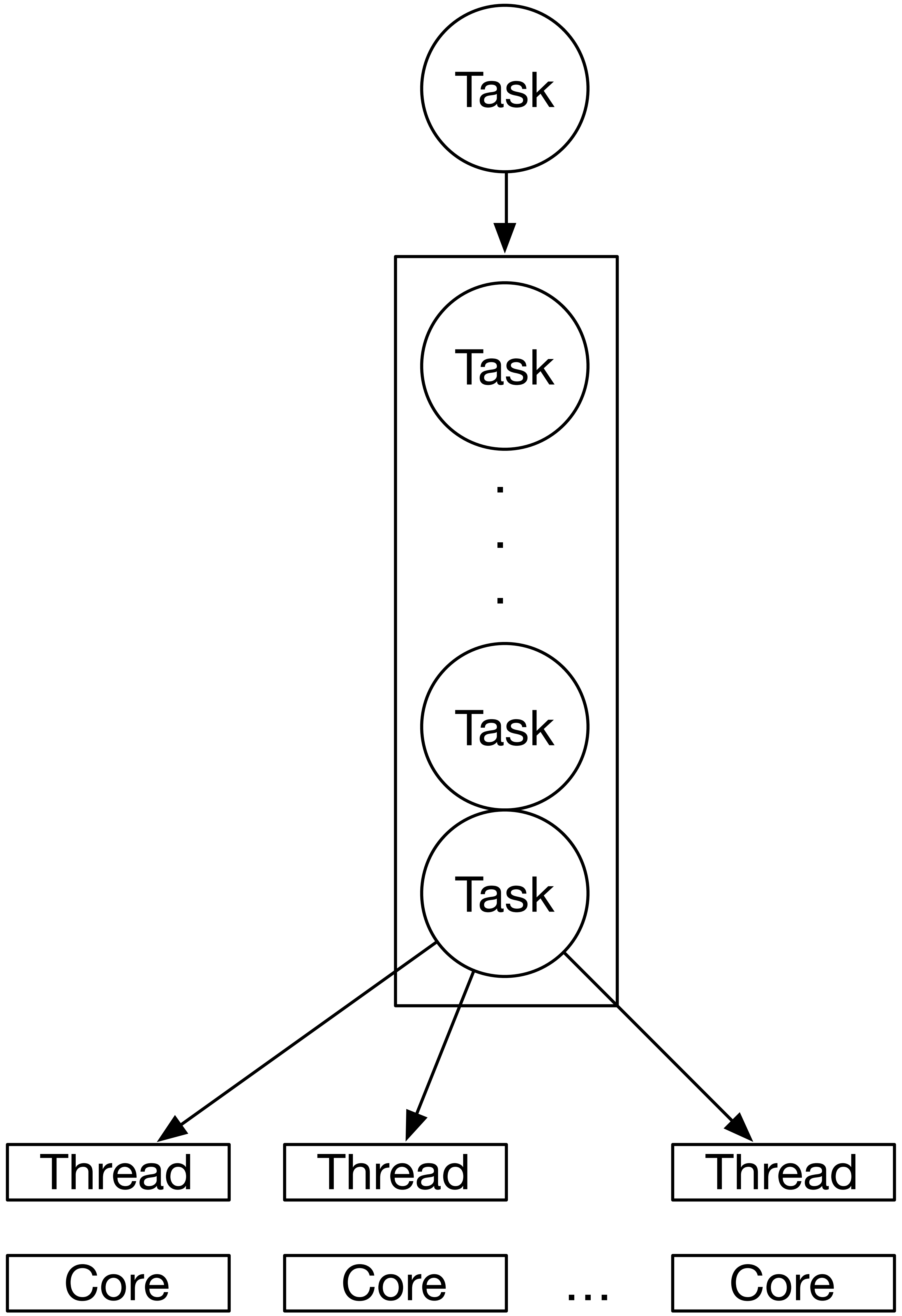
- C++14 does not (really) have a task system
  - Threads
  - Futures

- 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



http://docs.oracle.com/cd/E19253-01/816-5137/ggedn/index.html

Building a Task System

Building a Task System

```
using lock_t = unique_lock<mutex>;
```

Building a Task System

```
using lock_t = unique_lock<mutex>;

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

Building a Task System

```
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();
    }
```

Building a Task System

```
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

Building a Task System

```
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();
        }
    }
```

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); });
        }
    }
```

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();
    }
```

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));
```

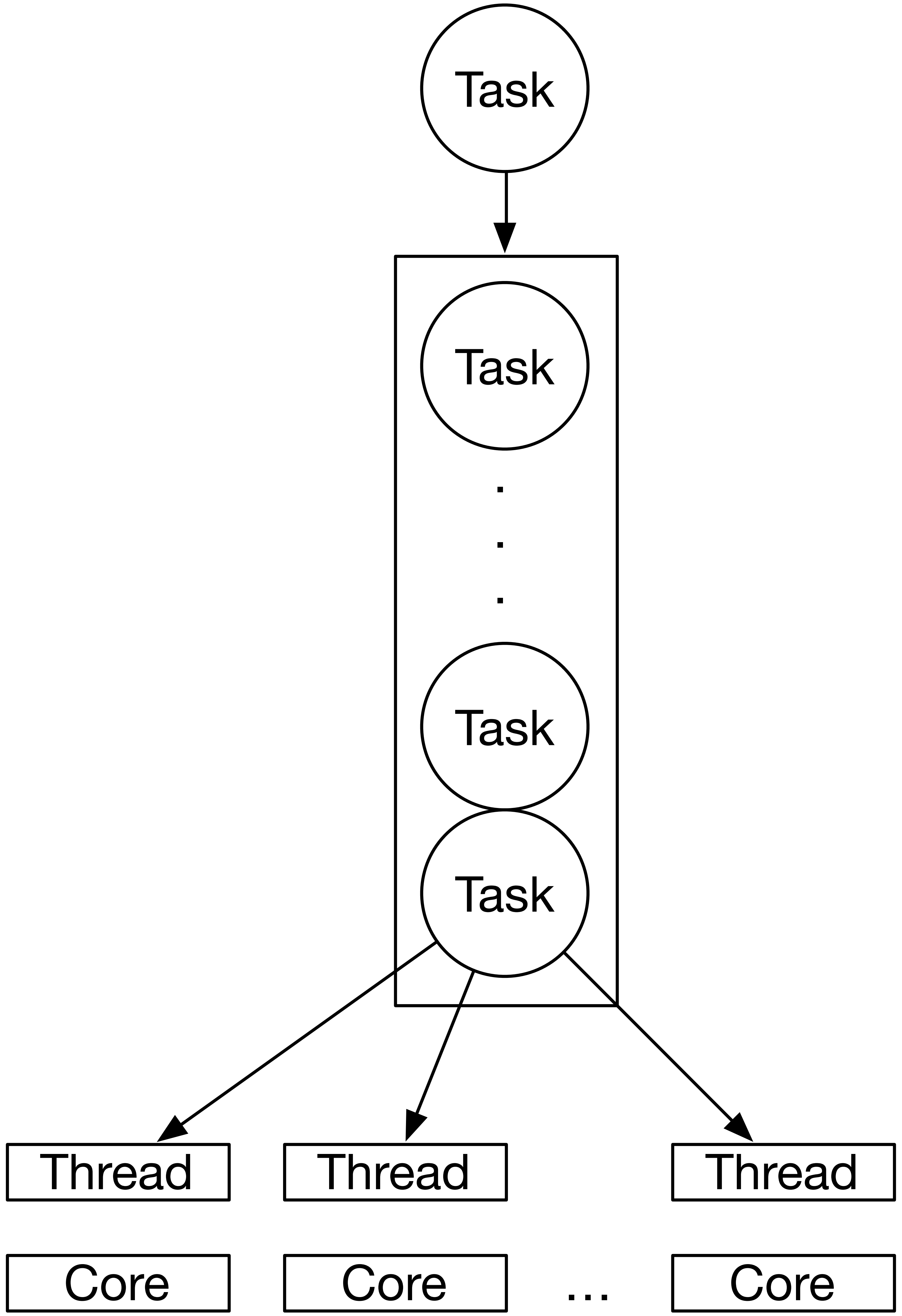
Building a Task System

```
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));
```

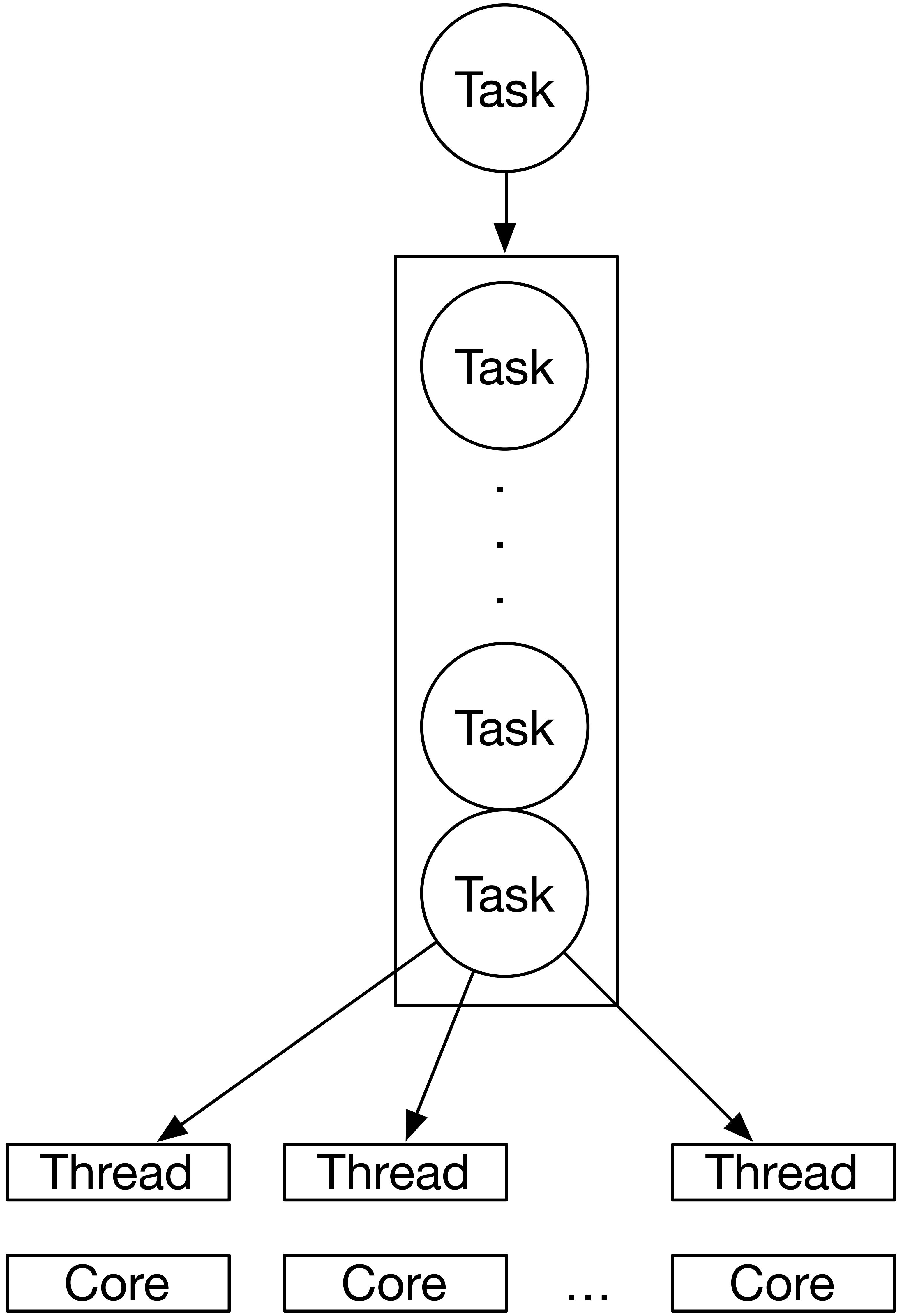
Building a Task System

```
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));
```

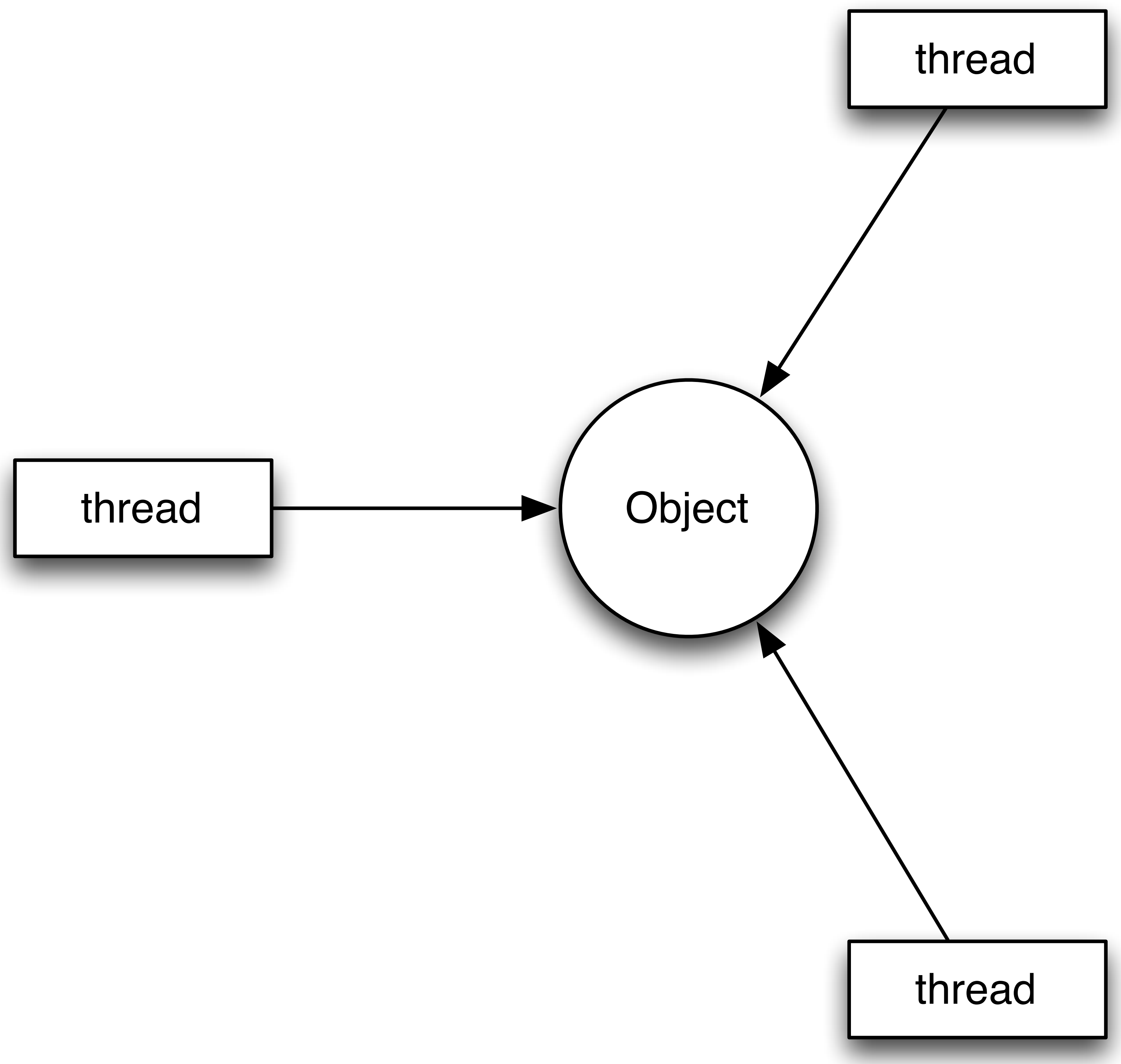
Building a Task System



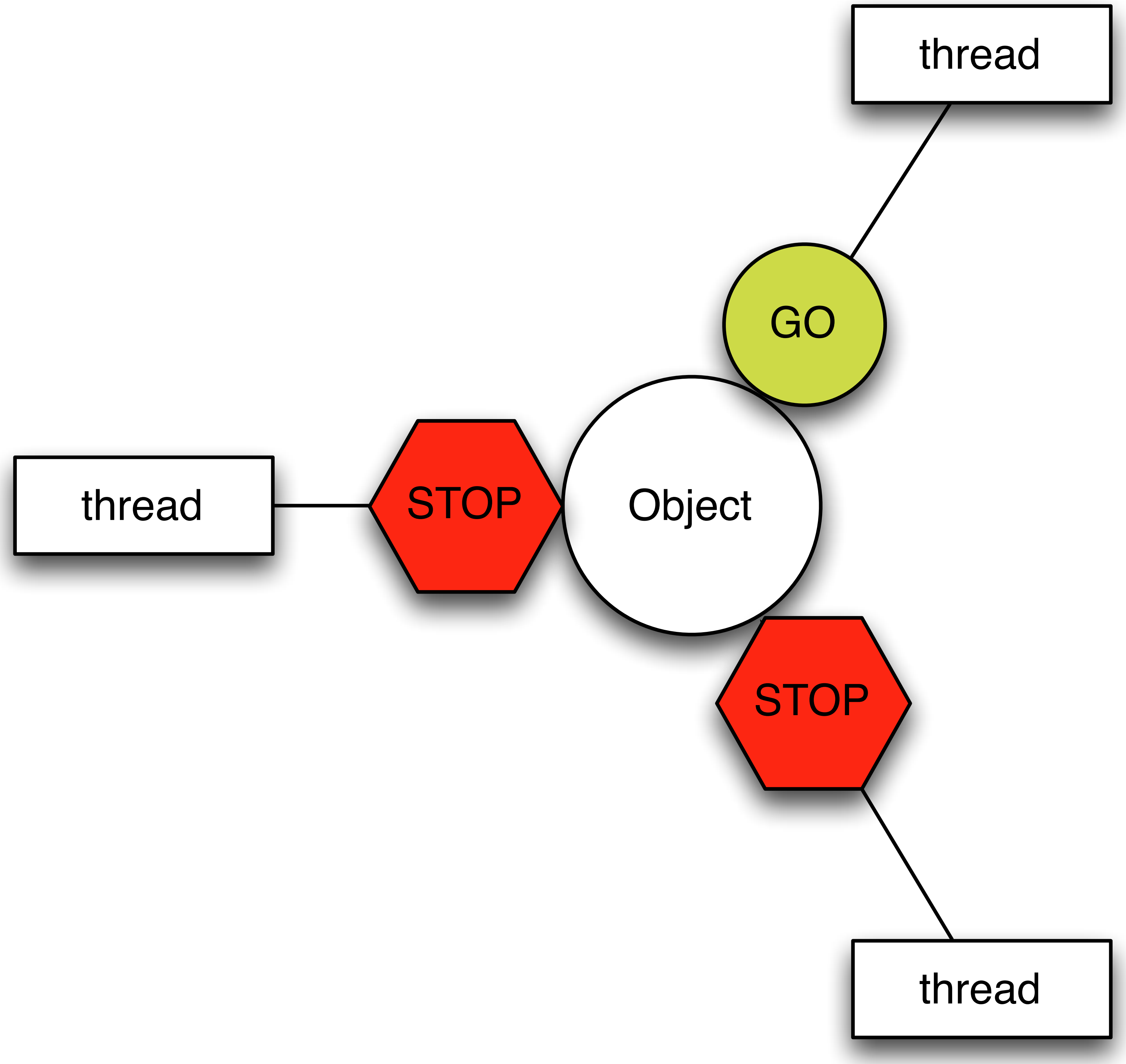
Building a Task System



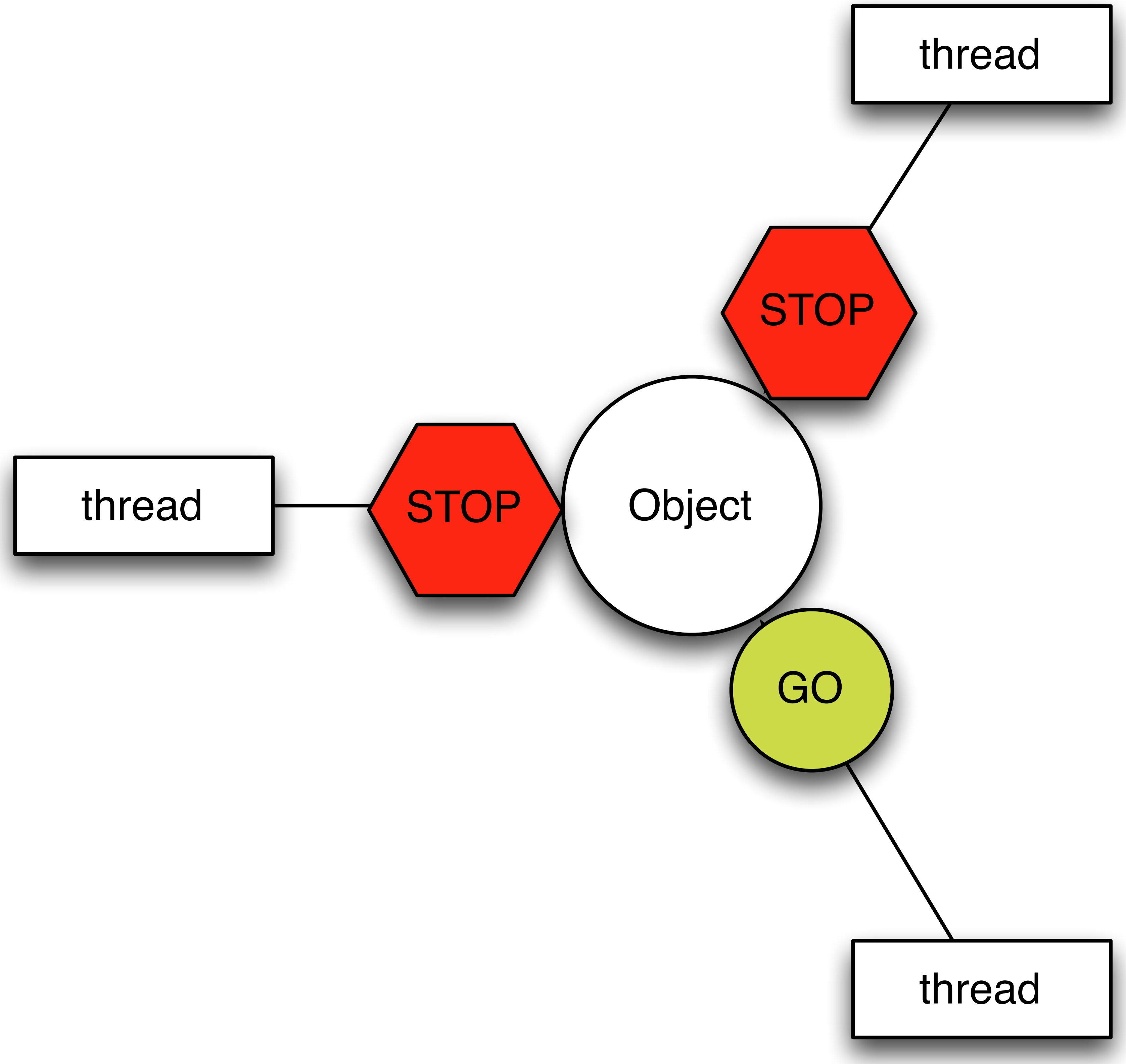
Why No Raw Synchronization Primitives?



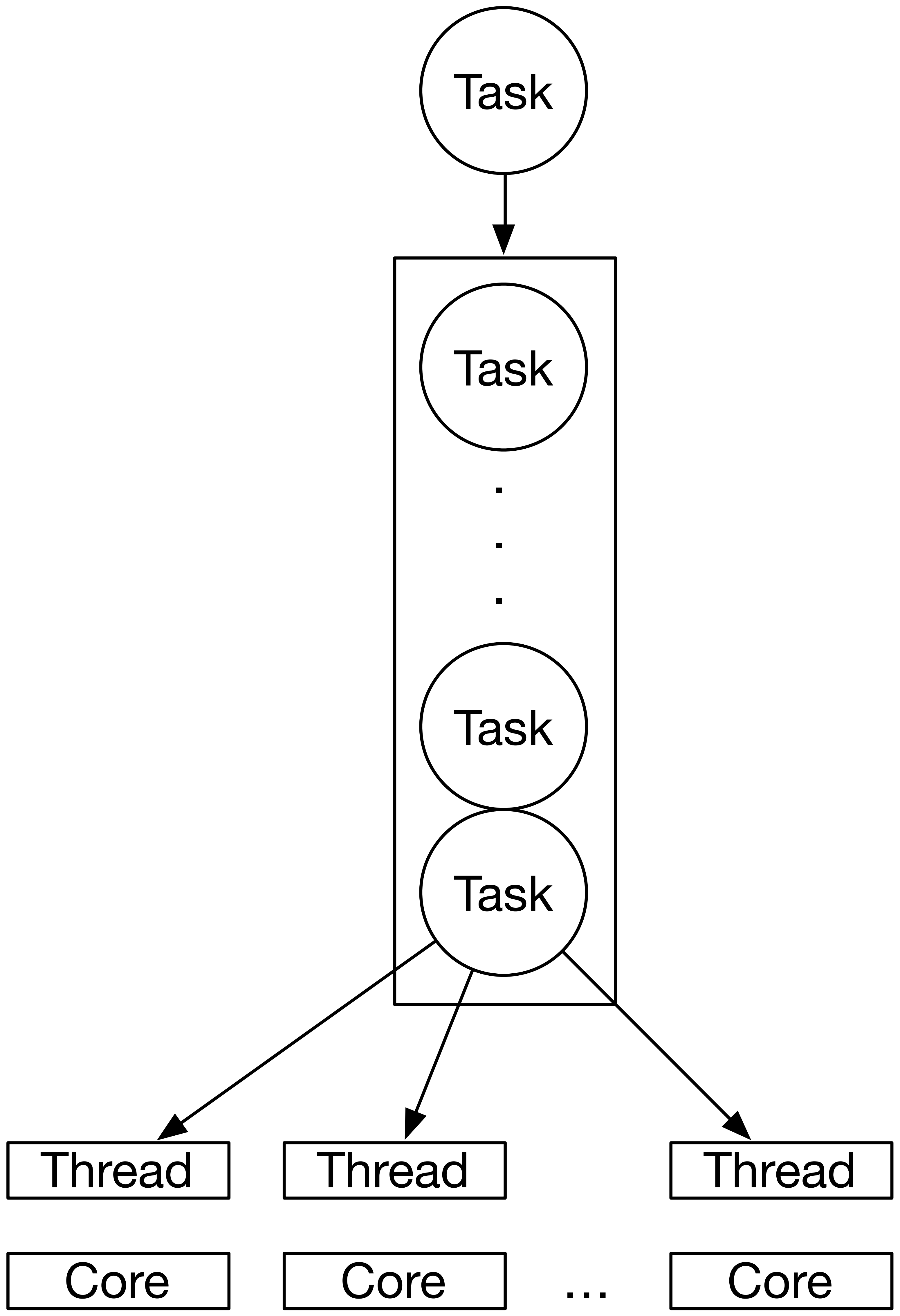
Why No Raw Synchronization Primitives?



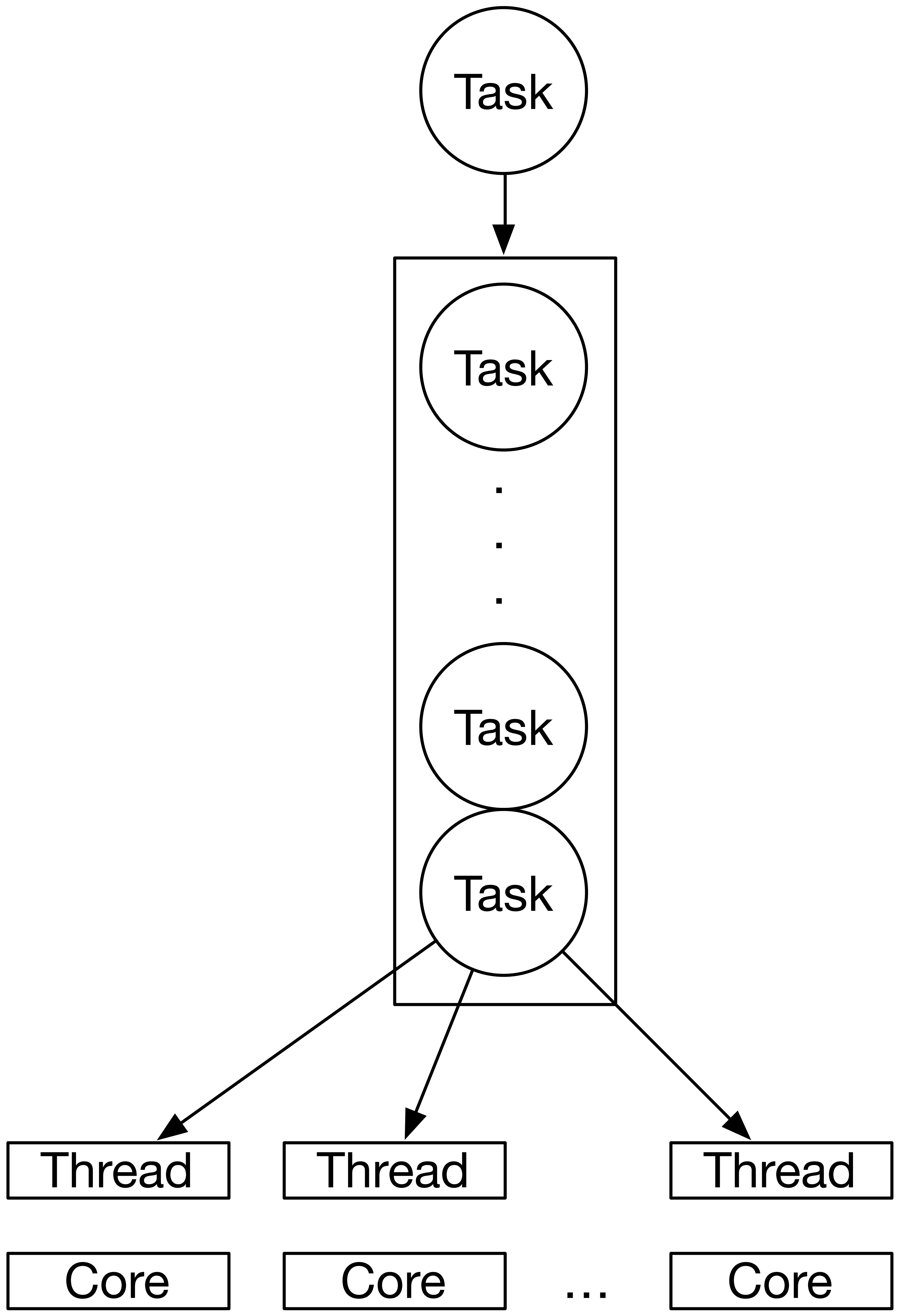
Why No Raw Synchronization Primitives?



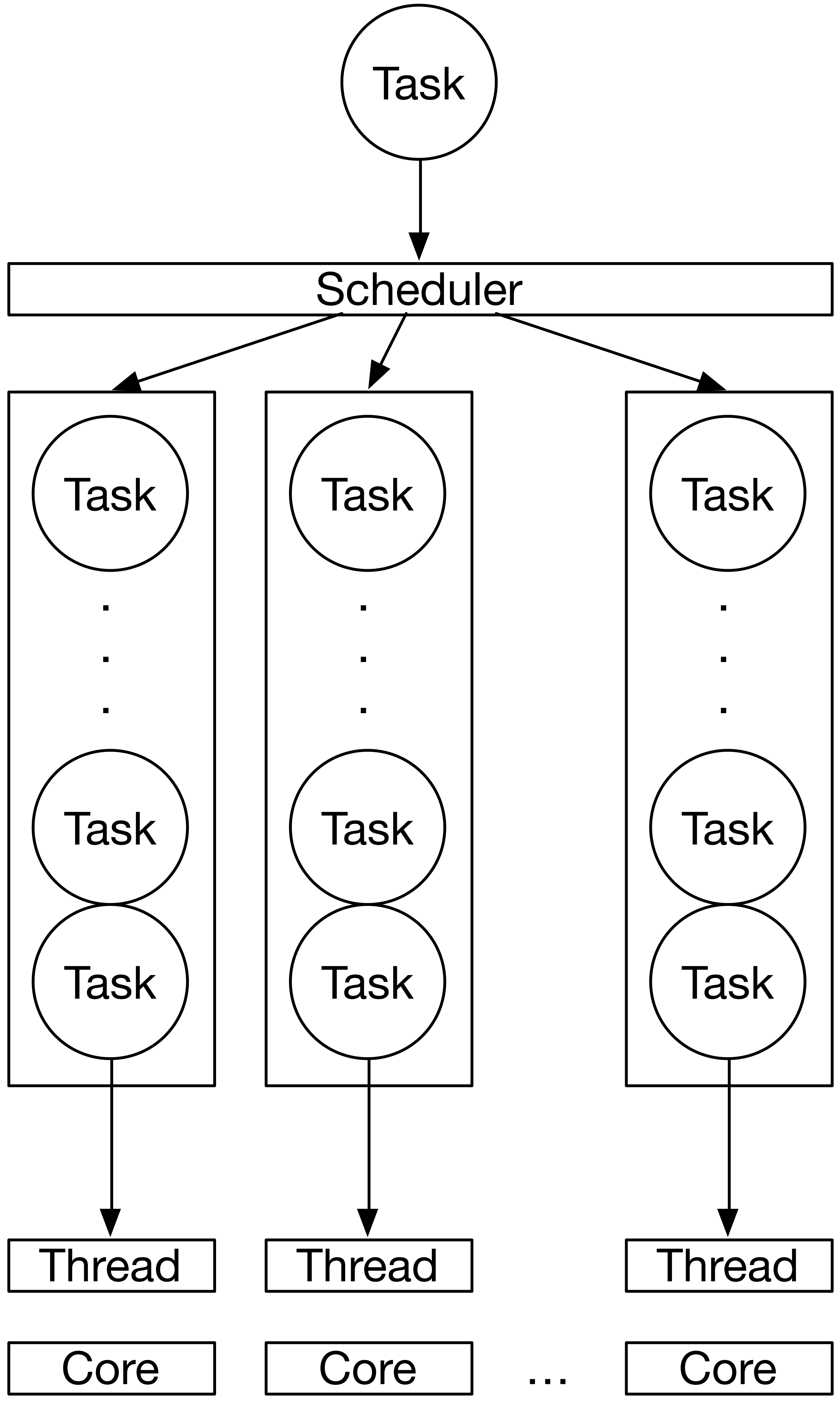
Building a Task System



Building a Task System



Building a Task System



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));
    }
};
```

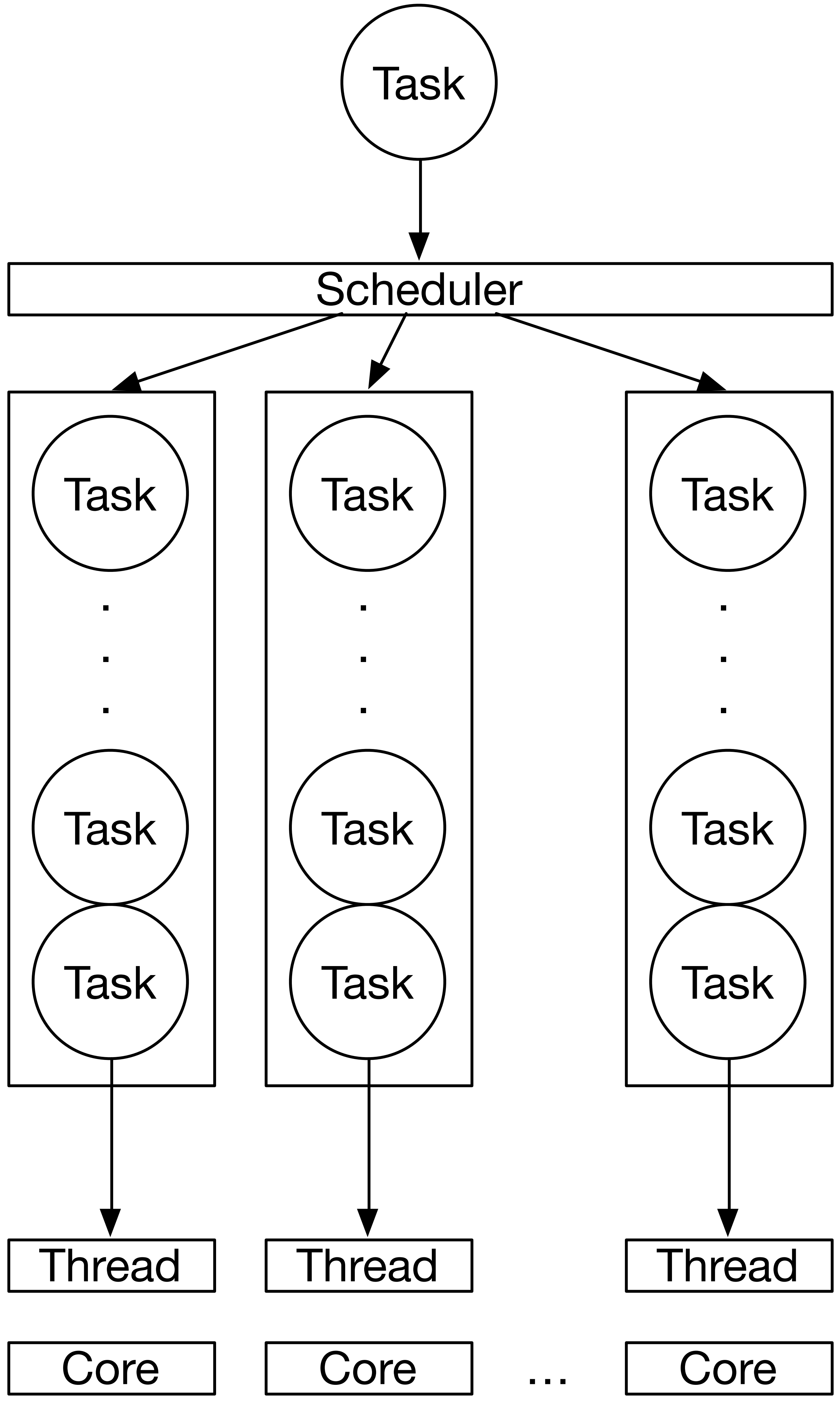
Building a Task System

```
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));
    }
};
```

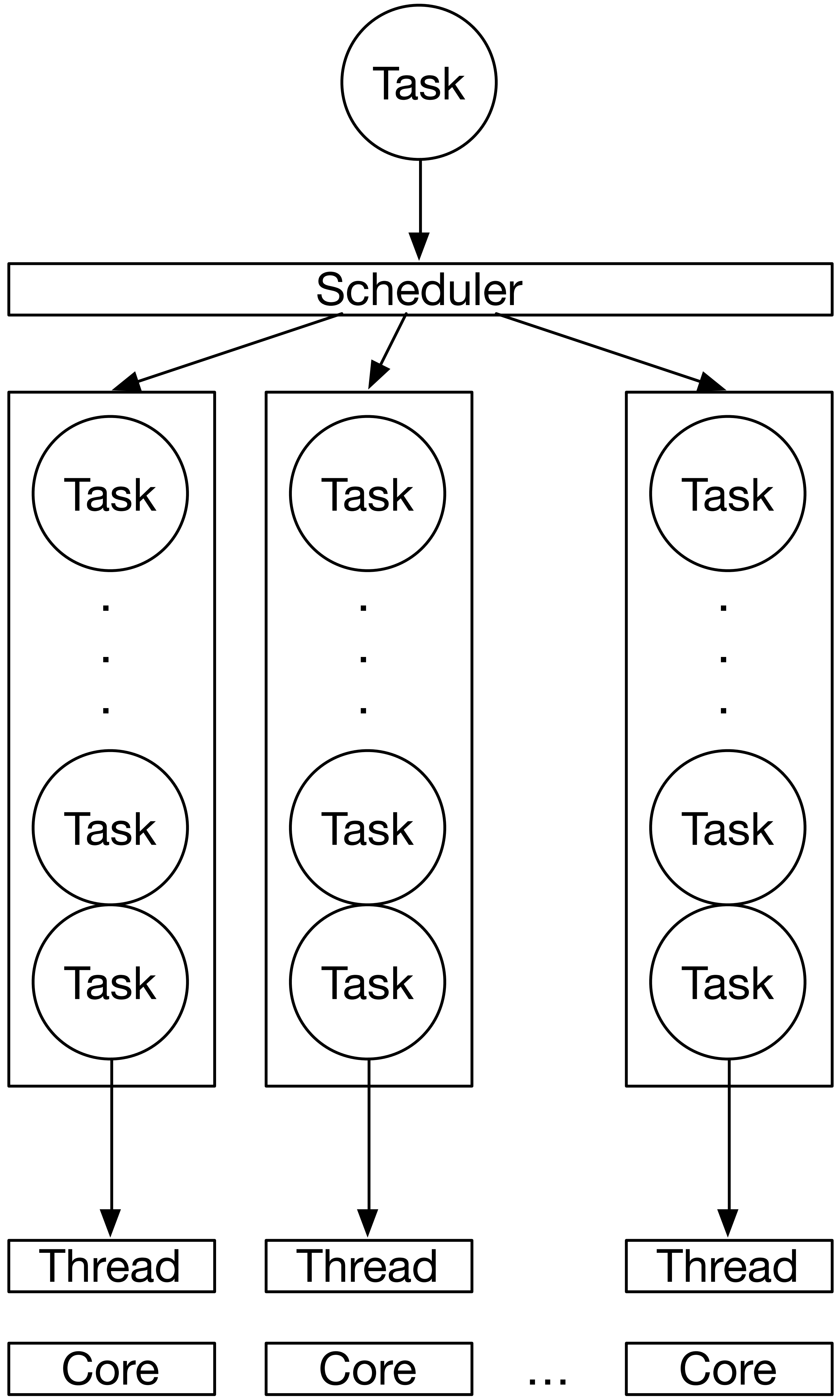
Building a Task System

```
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));
    }
};
```

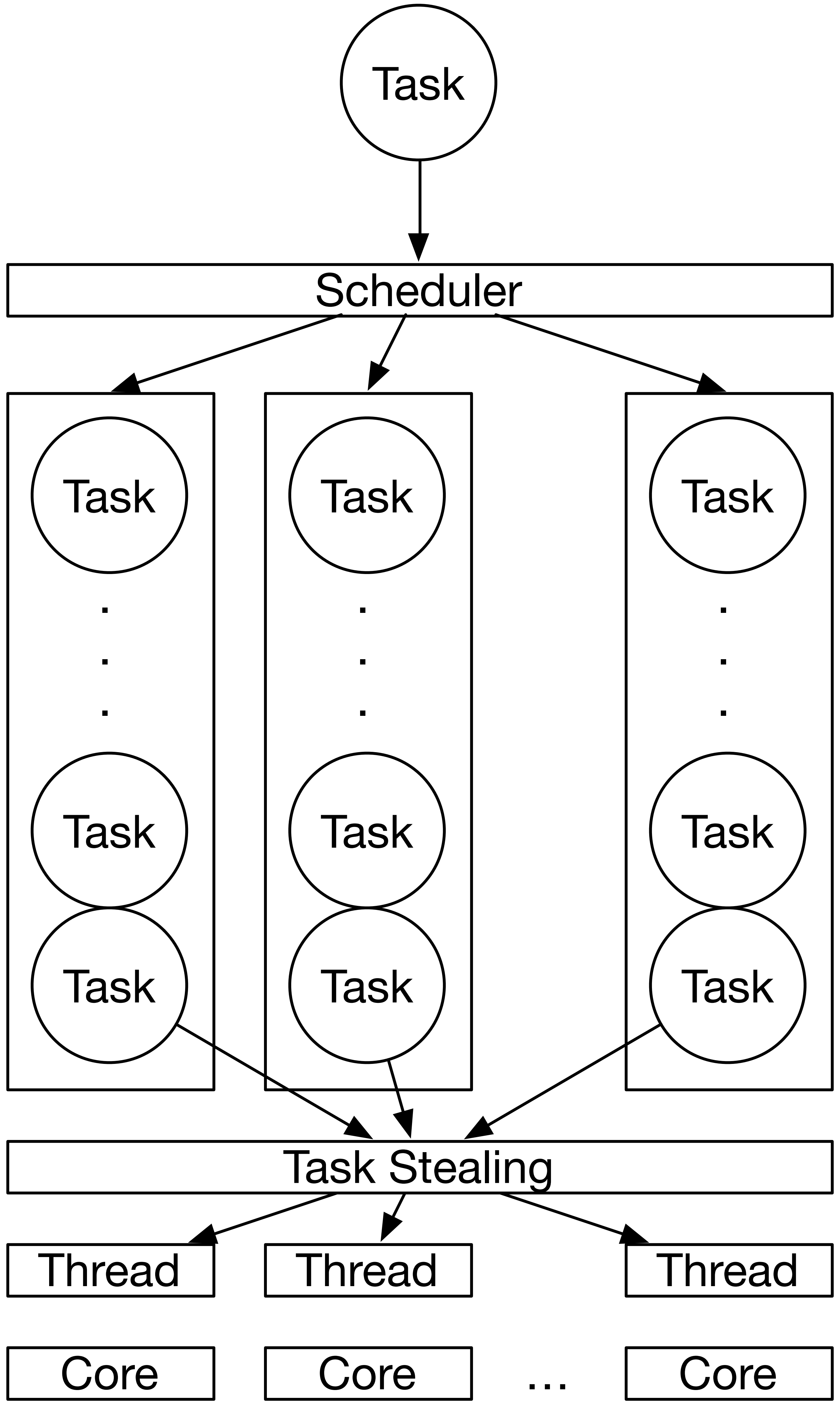
Building a Task System



Building a Task System



Building a Task System



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() {
        {
            unique_lock<mutex> lock{_mutex};
```

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() {
        {
            unique_lock<mutex> lock{_mutex};
```

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() {
        {
            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));
    }
};
```

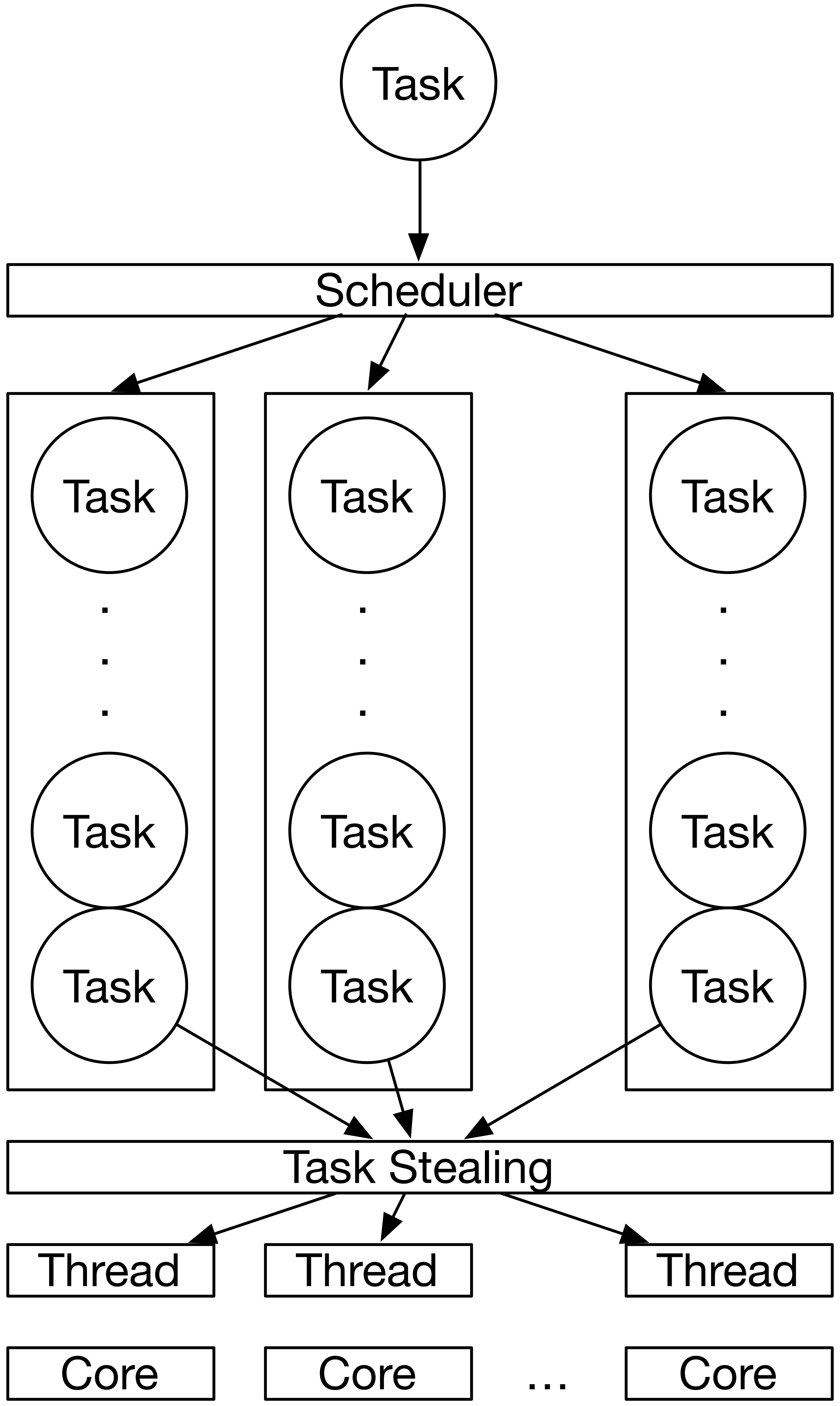
Building a Task System

```
    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));
    }
};
```

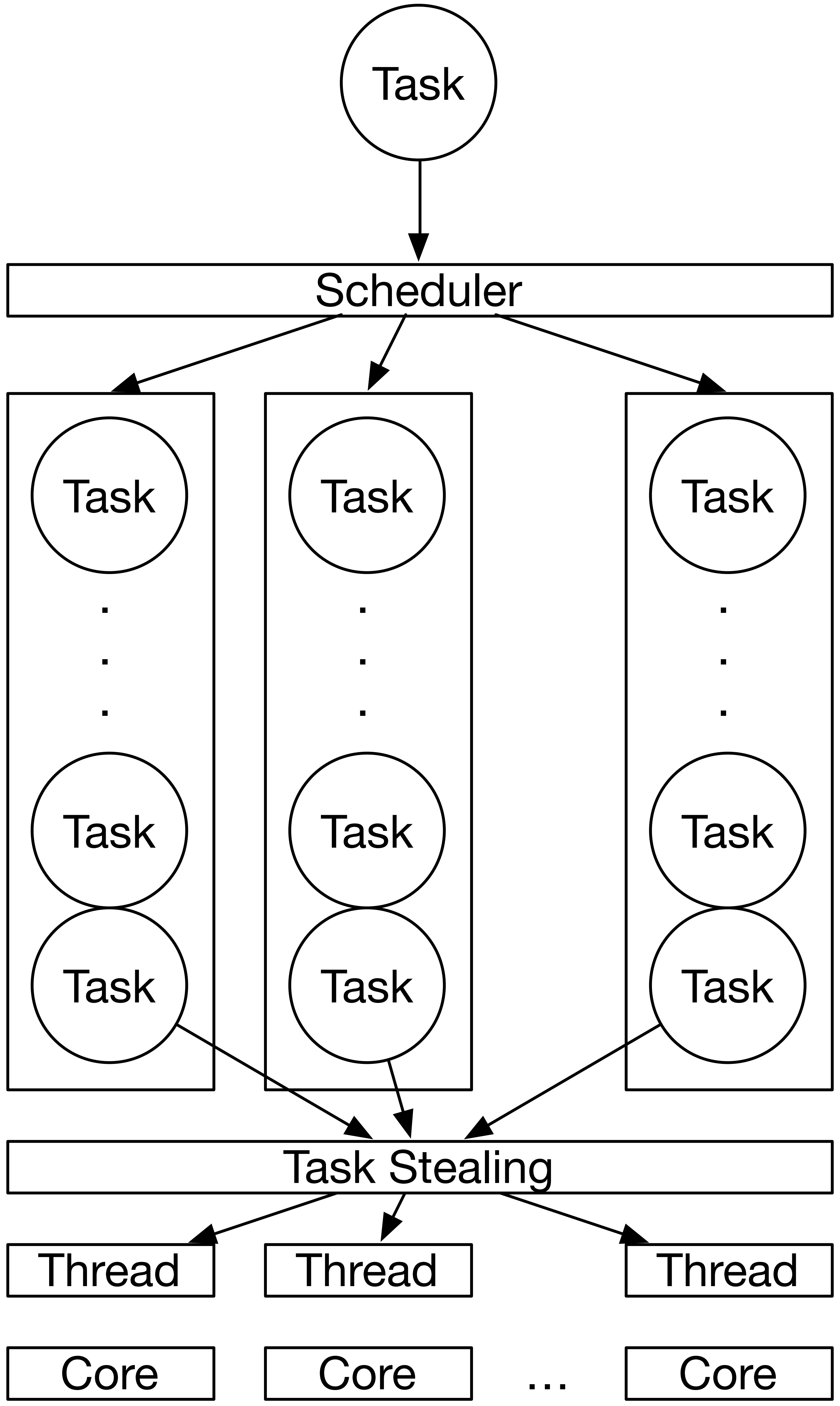
Building a Task System

```
    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



Building a Task System



Task System

- Compared to Apple's Grand Central Dispatch (libdispatch)

Task System

- Compared to Apple's Grand Central Dispatch (libdispatch)

C++14 compatible async with 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)

```
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));
    }
};
```

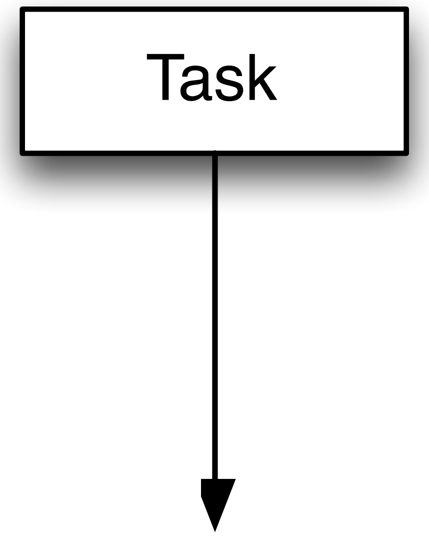
Task System

- Written with ASIO (Boost 1.62.0)

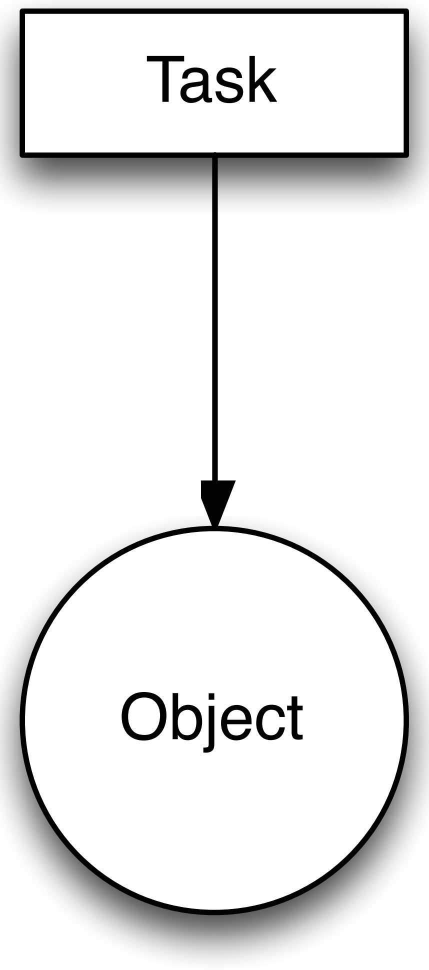
```
class task_system {
    io_service                            _service;
    vector<thread>                        _threads;
    unique_ptr<io_service::work>          _work{make_unique<io_service::work>(_service)};

  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

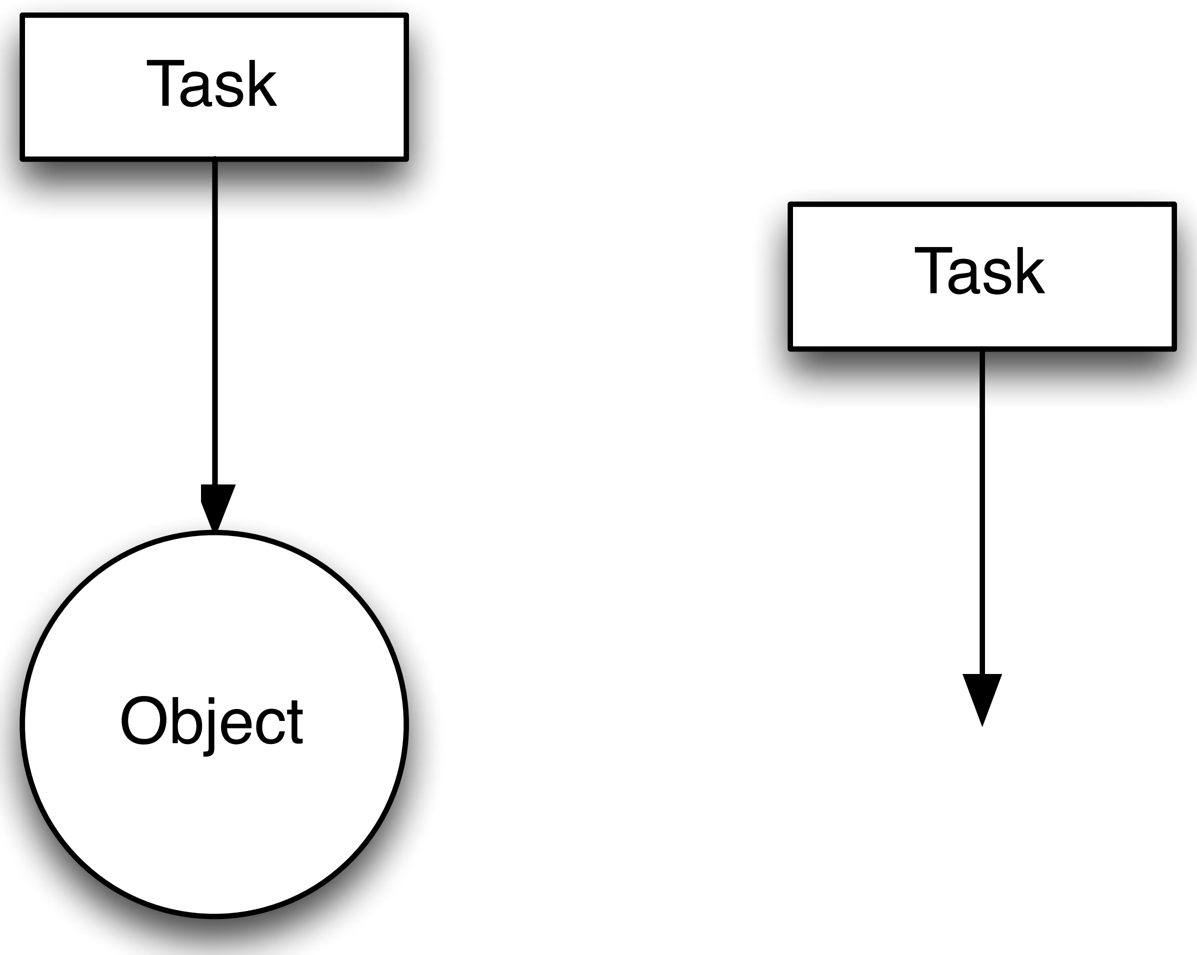
No Raw Synchronization Primitives



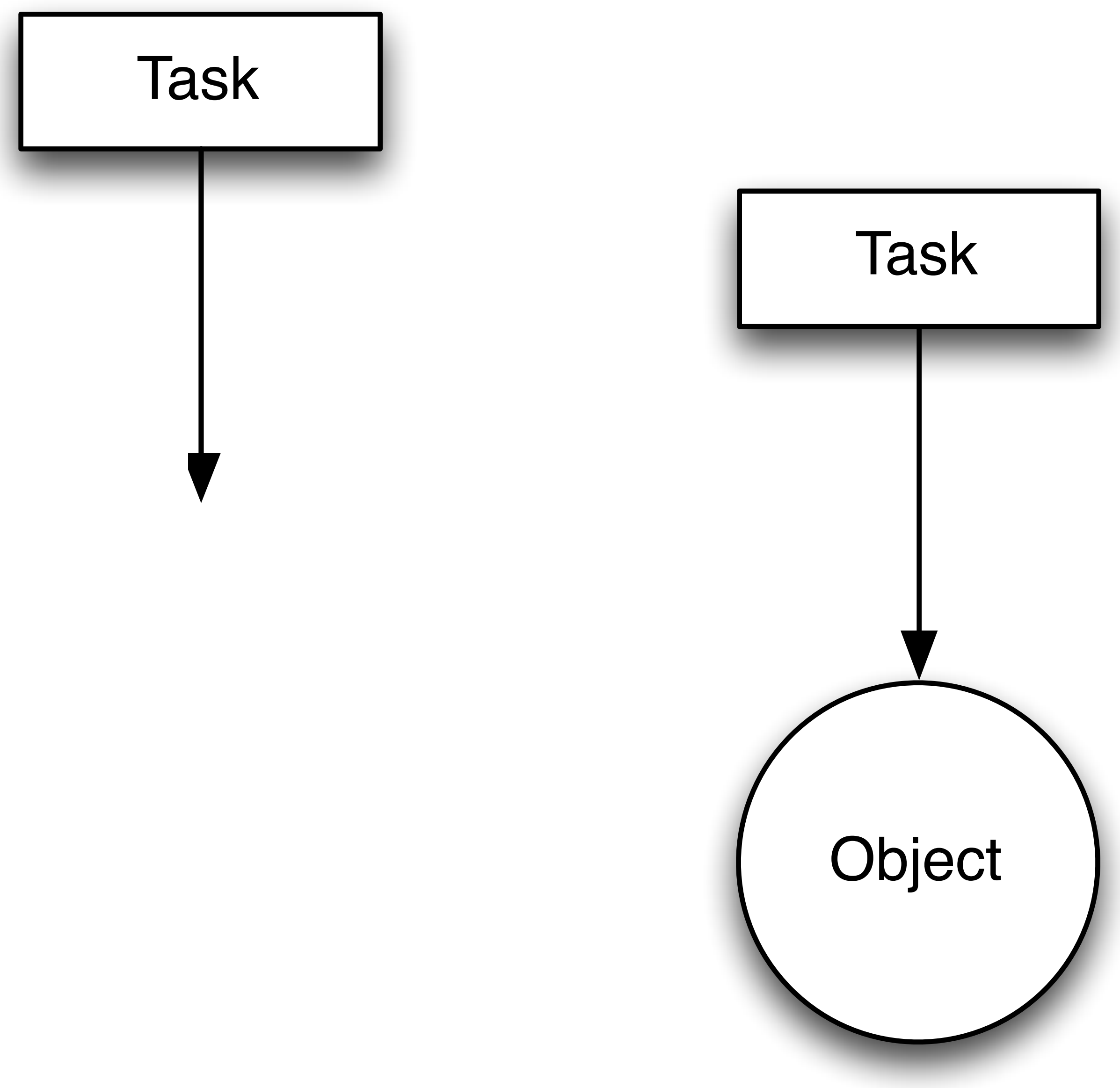
No Raw Synchronization Primitives



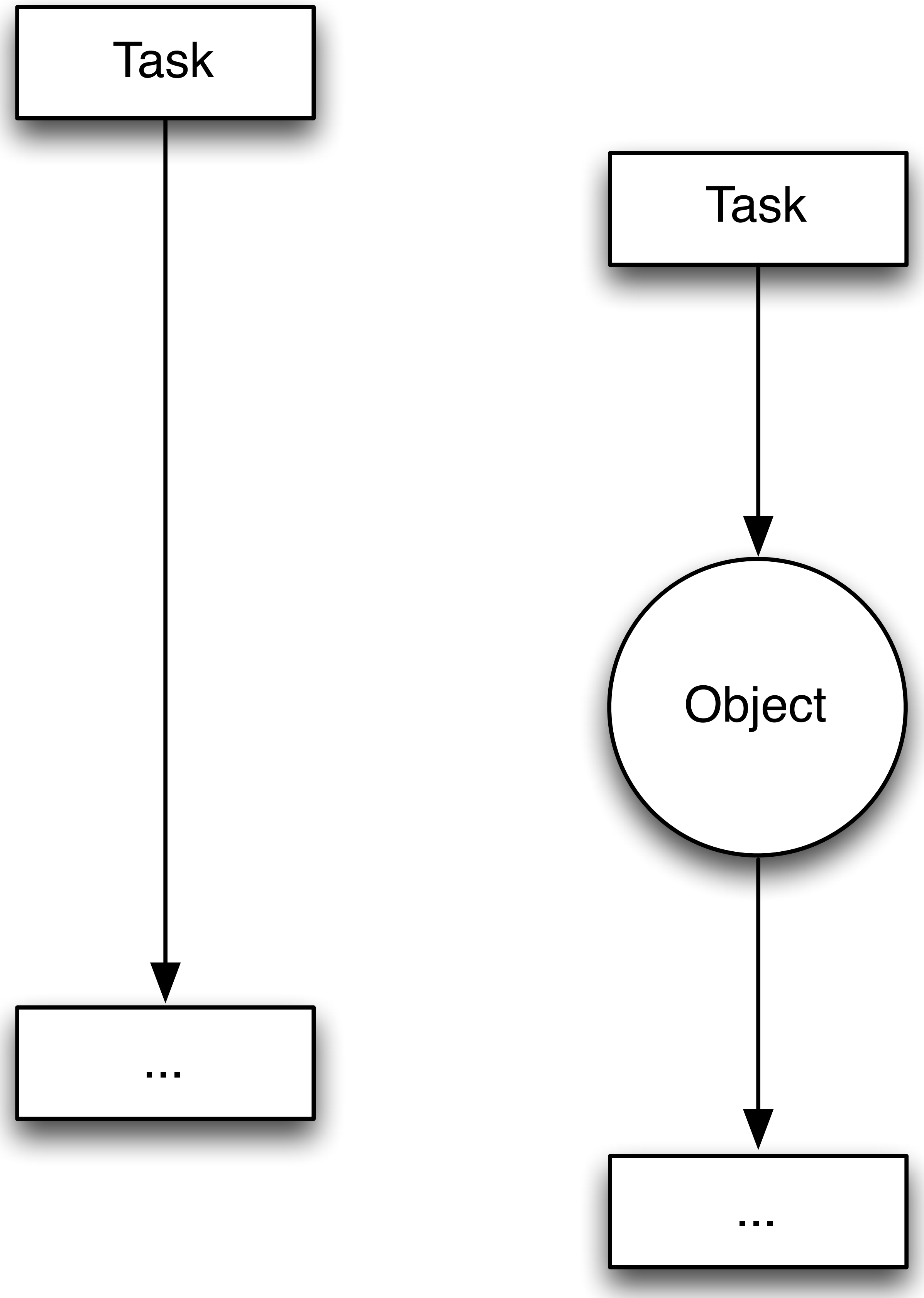
No Raw Synchronization Primitives



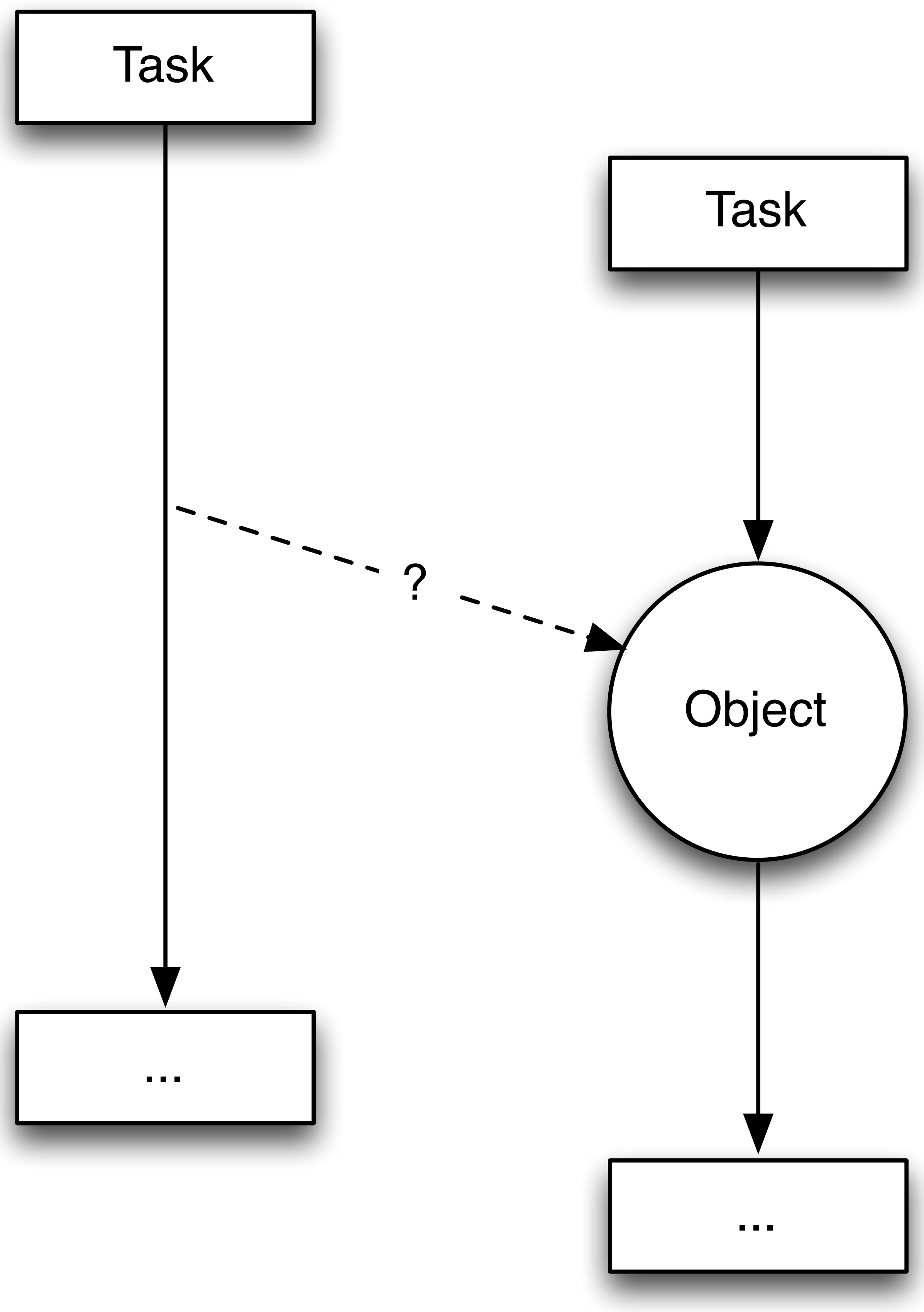
No Raw Synchronization Primitives



No Raw Synchronization Primitives



No Raw Synchronization Primitives



Futures

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

// Do Something

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

Futures

```
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…

Public Service Announcement - How to Write Fibonacci

```
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;
}
```

Public Service Announcement - How to Write Fibonacci

```
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

Public Service Announcement - How to Write Fibonacci

```
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],
                 x[2] * y[0] + x[3] * y[2], x[2] * y[1] + x[3] * 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];
}
```

Public Service Announcement - How to Write Fibonacci

```
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],
                 x[2] * y[0] + x[3] * y[2], x[2] * y[1] + x[3] * 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];
}
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Futures

1953282128707757731632014947596256332443542996591873396953405194571625257887015694766641987634150146128879 5
2433522023608462551091201956023374401543811519663615691996212564289430337011382780063800276741152792746666 9
8655783793188228320612714975832303348548934895725992307229129019282092643316275217308614600179125820426996 5
9936020959339202005184862028402447343139811367418720203868480175318538621112878108240617741383293554561687 6
0645406512595471802912654794289403698165920636101935929135213541037679908294032015570271611539503197597324 7
7821629576316296533566947776632850623452455934606475750259358134434578167676462587885901137272990737294785 1
1448089572456191503507025589529116868550008802013233458747217794781447546792016090170642585629359747546532 7
5757574007743203491342878518979535430473456030776507893876728653916679923281744936199152376814955763208537 1
0478597061884387315305823956275608790631078190049751695947097367138917457045552021351233507944033607120305 0
4144685221041565037321067932275625864751191461141736034968121738023422478608029202109319249649040983239706 6
8324705444176351252673245527541950168384520602300739495985427929829783120438211575764578769249558335140252 2
1527206624418090032593807536284917966809529711850719137983367887377045991363933395581421203699026161797211 3
2509184002305532760710431647819097430043464779336328760146999612802392582947155731668894333945542929287187 7
4877478920429616635653661079602391970210972847296670942733458634479804863394463521165497150726134276820547 9
3209317507988801013041602798250635418234403455874223670128266635693461129461312312838906003654732766024569 3
1515185001832848315064548002997893598516123707404615822935444070174833951457586954749175026454212636426222 4
7206004885546258996119047589210122428054289862159464666247856437357221777554987608768591203011855163566890 2
0103446399839773266388890365078416180709154525299275973521395741547772914600879431433915606044582510782351 1
6627189263792331301464388059787946844487906057678629746098962742666356968247429338674020743655942605794479 0
7119305225893159071938654552588042913974714018184916973383813844615484306312364929083558427807845613193645 7
5591172213694633818031160030789621166865289595377843646440238251636244971819738544414956313171400285033892 8
2227413460301809422483721632185471727045281382407842563874736524914111808078386650633994537623920670051339 1
8733310713606969818962828476324542329930627287045799129324574116753390227449996309656668092226251646858254 4
5578513498241441272612401581575381809846666714500698883917855180089437018902572199248520874291556026191775 2
2812466062899678716652967848726848490504132849729771268801163997837643428020245225155010224035416988518537 5
0158467388119404761972061960312653449675991789324447817070290444658957195022880915779389764242375181402099 8
9995816123147790229578110016867018673861986179713813985466628196954855374070735622861616553942807641840809 2
1204793281668300598450478792940635631809747975515203509468276591874161090763750690276529436756153980326138 8

Futures

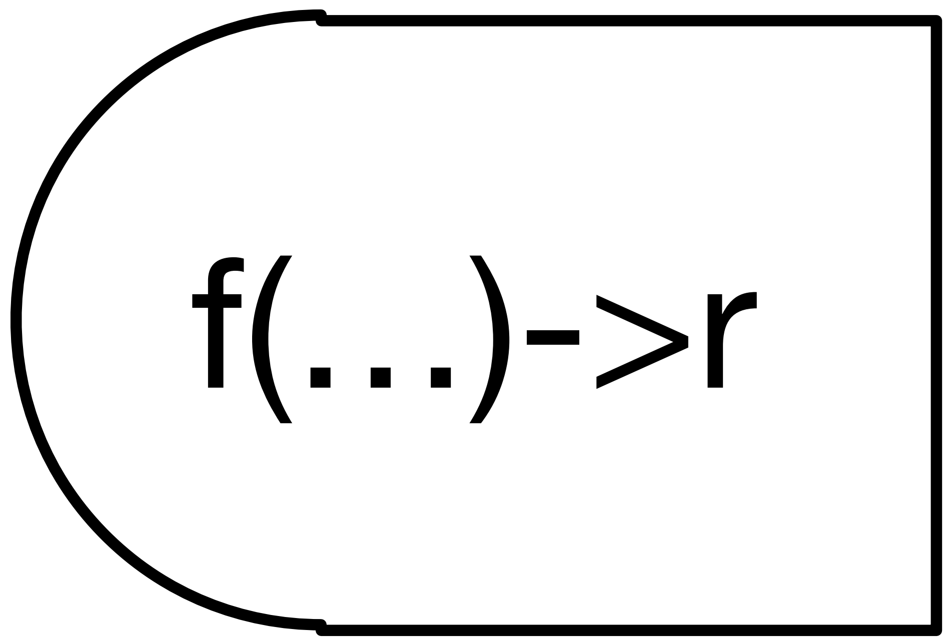
```
15790155892833100345673846243104676900000936756893803676769777642059716492347060997973282994459039755683869
10568541105888505197986232161807165960864316652383369579251545877324797429523572491518310013505994095431367
23454418539676396422570487868443336735568511535850565172490141772333018072390350689838662532338266203548476
87722321662223383305226882245421258277211223435986491973881404168406609216954760818955479619408040043497601
35646408461148077885537891122888139618703907906033147416881433658136276942006644505679690480702792206520855
12245086839375655196861305232092138041808273198852928058246964575561801618520046644949262341864859342928965
21378574554544426221453176445385228867960454072522804961741905198550911362542849130027243353553345377968558
49780195976636516290598457219043489821358221206856924121139313137132134865741440892670003665555632446499775
56853514681289887391700907057970839124191923062570547772748610990924519168225326823578140721238189631411471
29610287340041050015549547086272721534936510345705849389706515725684266079756708385889612130516276472992631
59674474594901199950849178952149715987731953191759591623424021718579696778102054496598766846143959650647332
219853235213781081870306428755069518903435871816336041263976750209091335484801511135951824112432636080497447
37395896608759569909256138919905403404664655310556021101996525724843421071082933739200159651403373870955680
75656822683537933983982488022723770319785461480932302347255796621173892988541730741484707211664044157057536
04582256143224299859780683239696543855523783781413866750792868372058020433472254190336846843017198934115689
96526838242546875
```

0.72s to calculate
208,988 digits

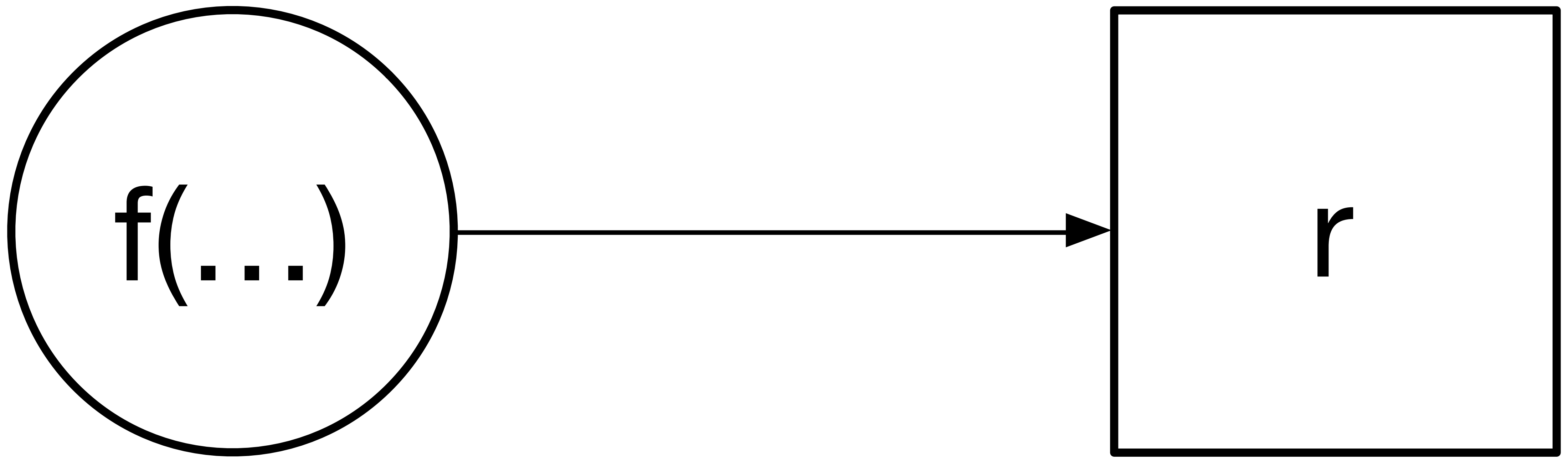
Futures

```
future<cpp_int> x = async([]{ return fibonacci<cpp_int>(1'000'000); });
// Do Something
cout << x.get() << endl;
```

Futures



Futures



Futures

- Futures allow minimal code transformations to express dependencies

Exception Marshalling

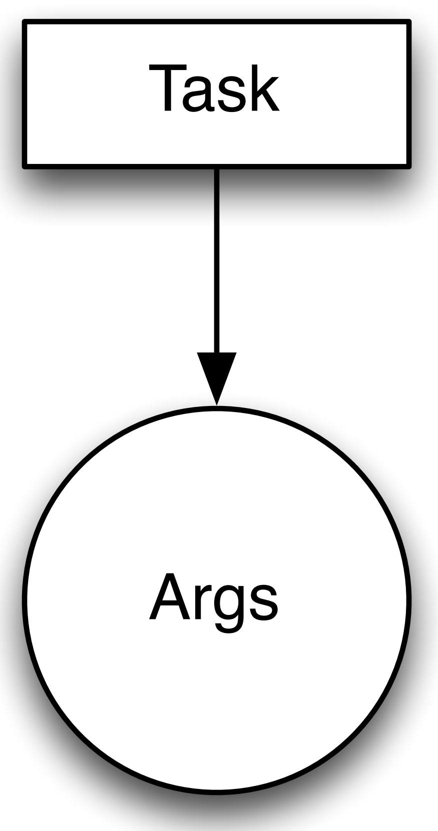
```
future<cpp_int> x = async([]{
    throw runtime_error("failure");
    return fibonacci<cpp_int>(1'000'000);
});

// Do Something

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

Exception Marshalling

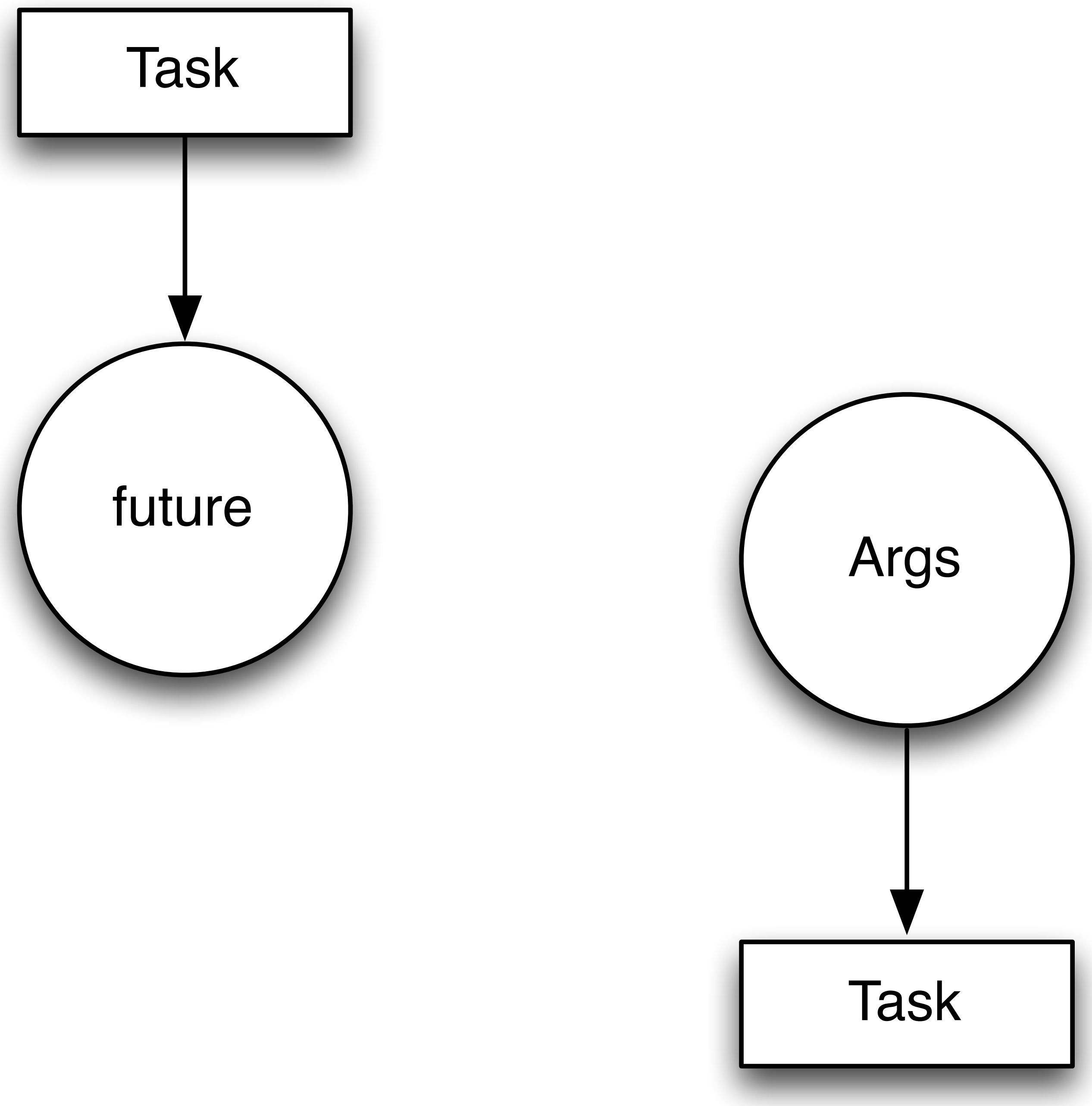
```
future<cpp_int> x = async([]{
    throw runtime_error("failure");
    return fibonacci<cpp_int>(1'000'000);
});

// Do Something

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

failure

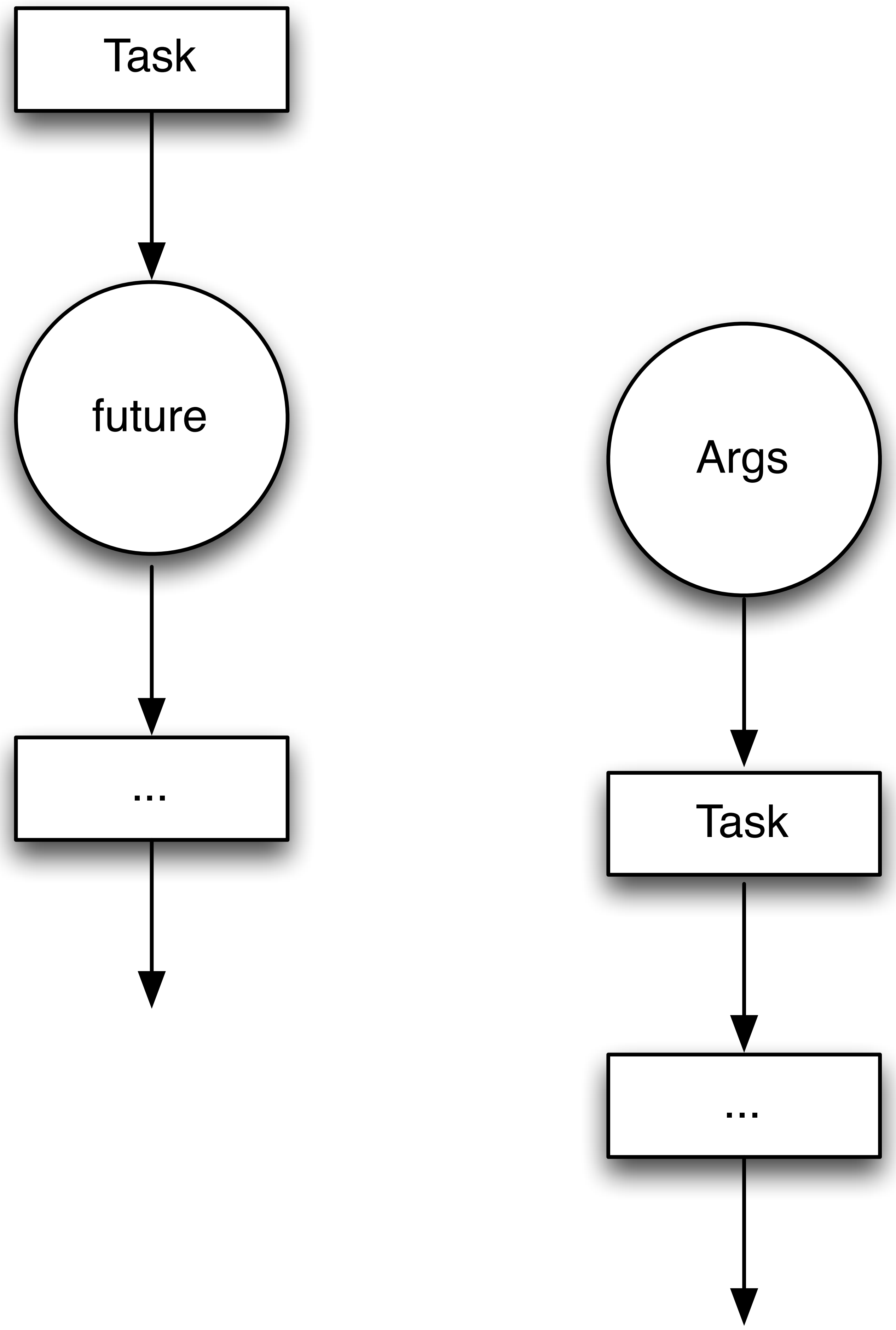
No Raw Synchronization Primitives



No Raw Synchronization Primitives

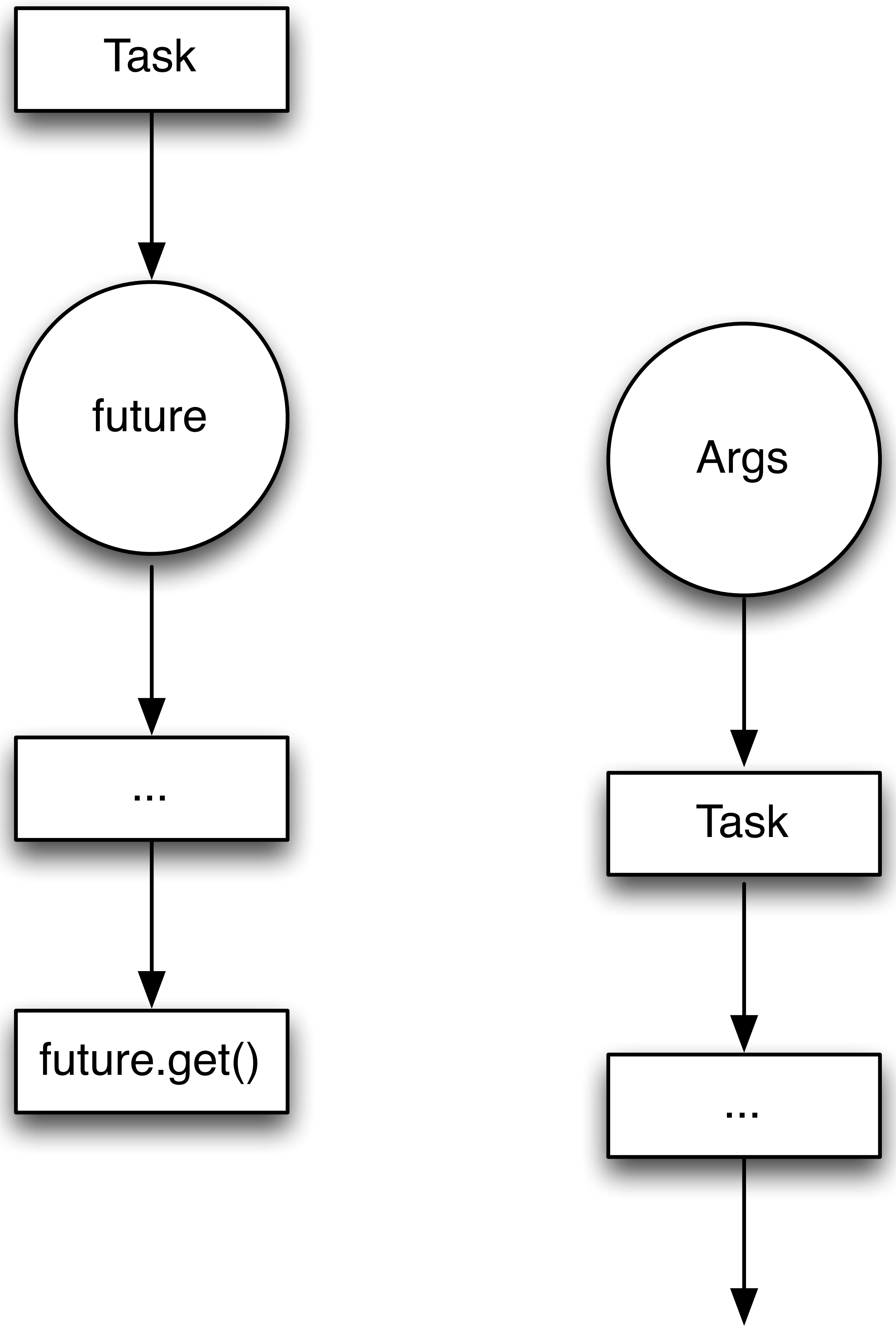


No Raw Synchronization Primitives

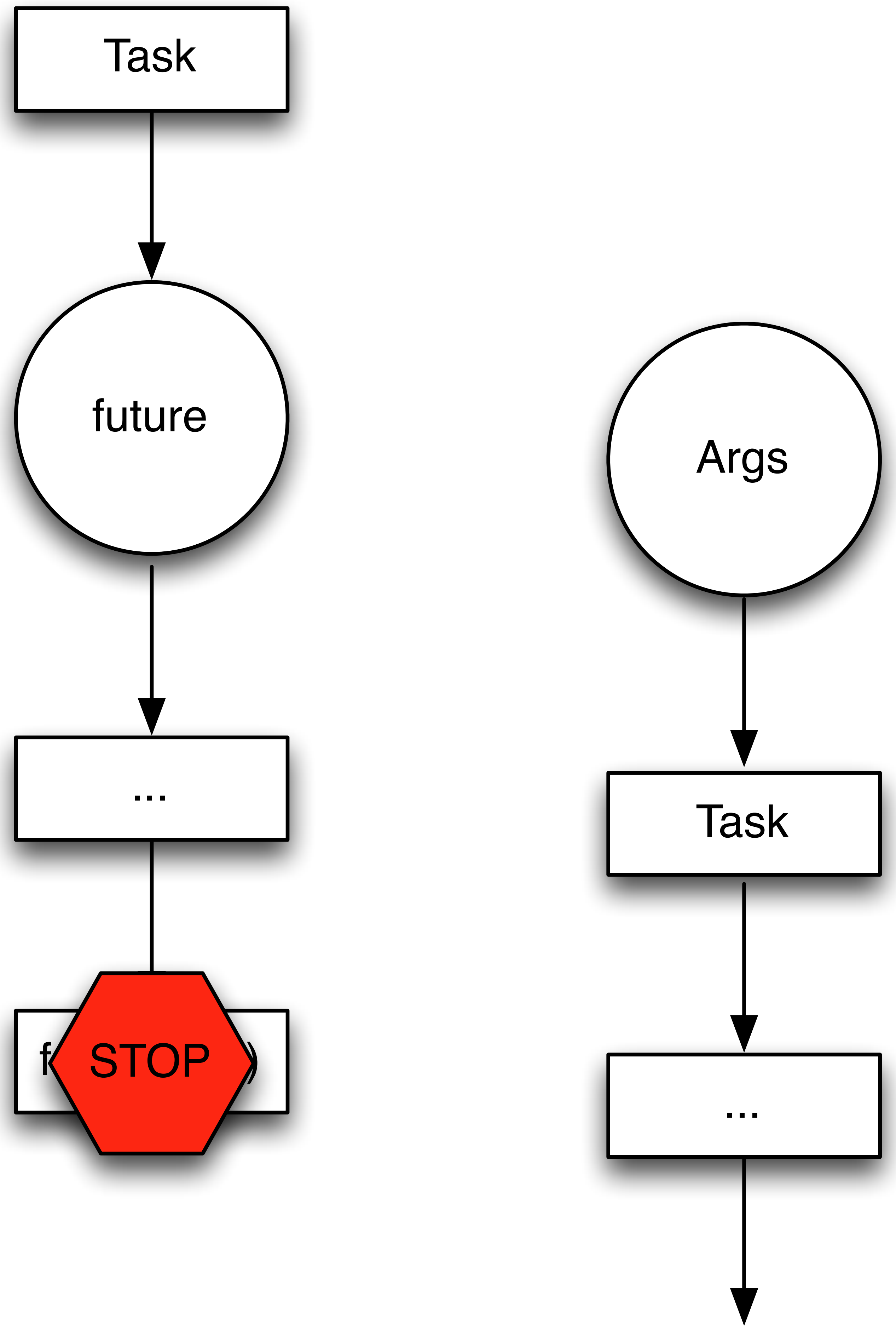


Adobe

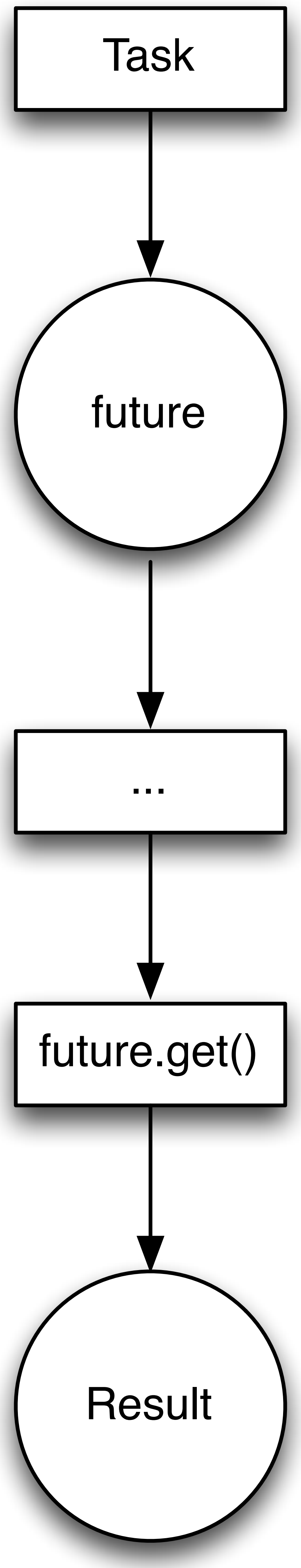
No Raw Synchronization Primitives



No Raw Synchronization Primitives



No Raw Synchronization Primitives

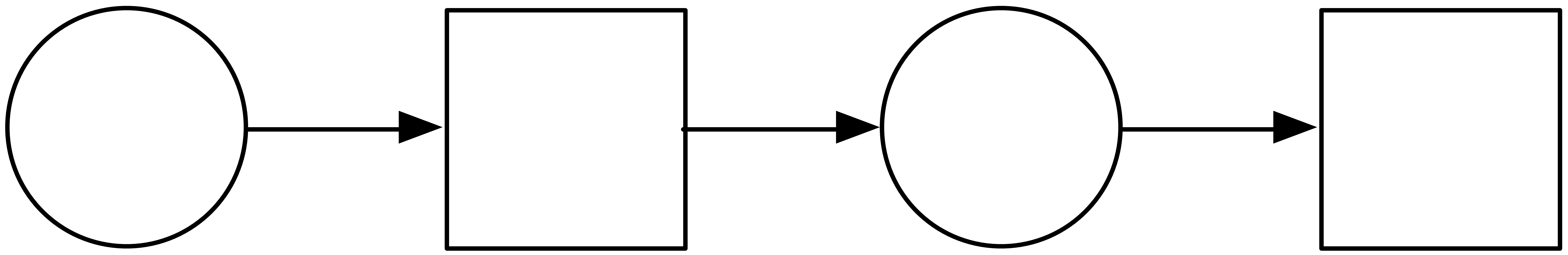


Futures: What year is this?

- 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

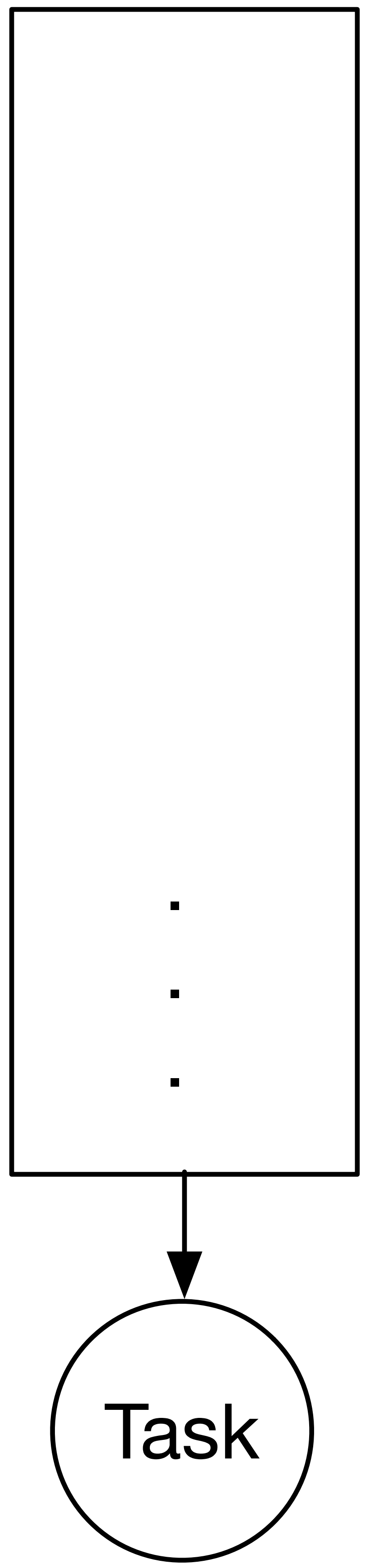


Futures: Continuations

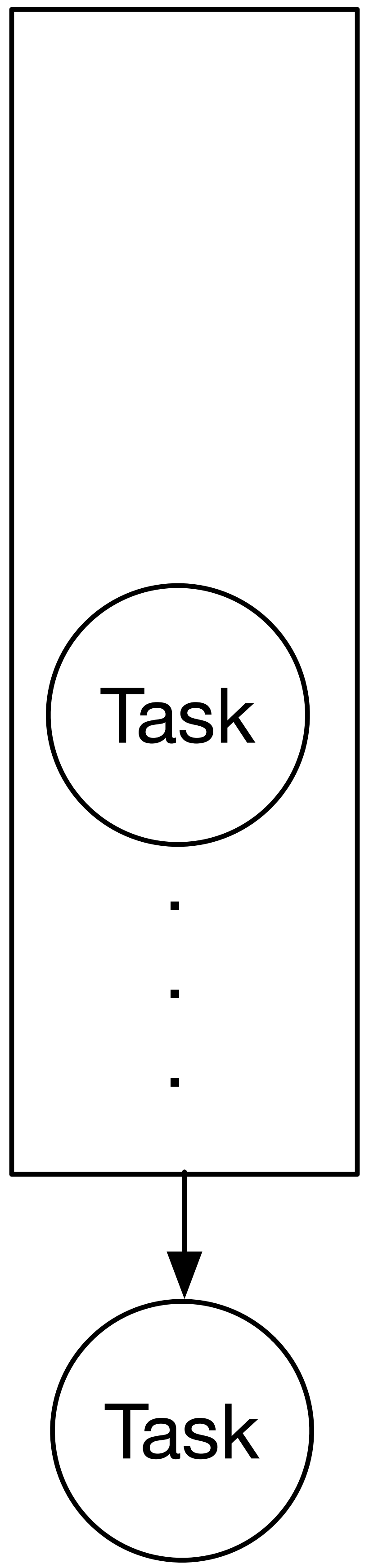
- Blocking on `std::future.get()` has two problems
  - One thread resource is consumed, increasing contention
    - Possibly causing a deadlock in our tasking system!
  - Any subsequent non-dependent calculations on the task are also blocked

- 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

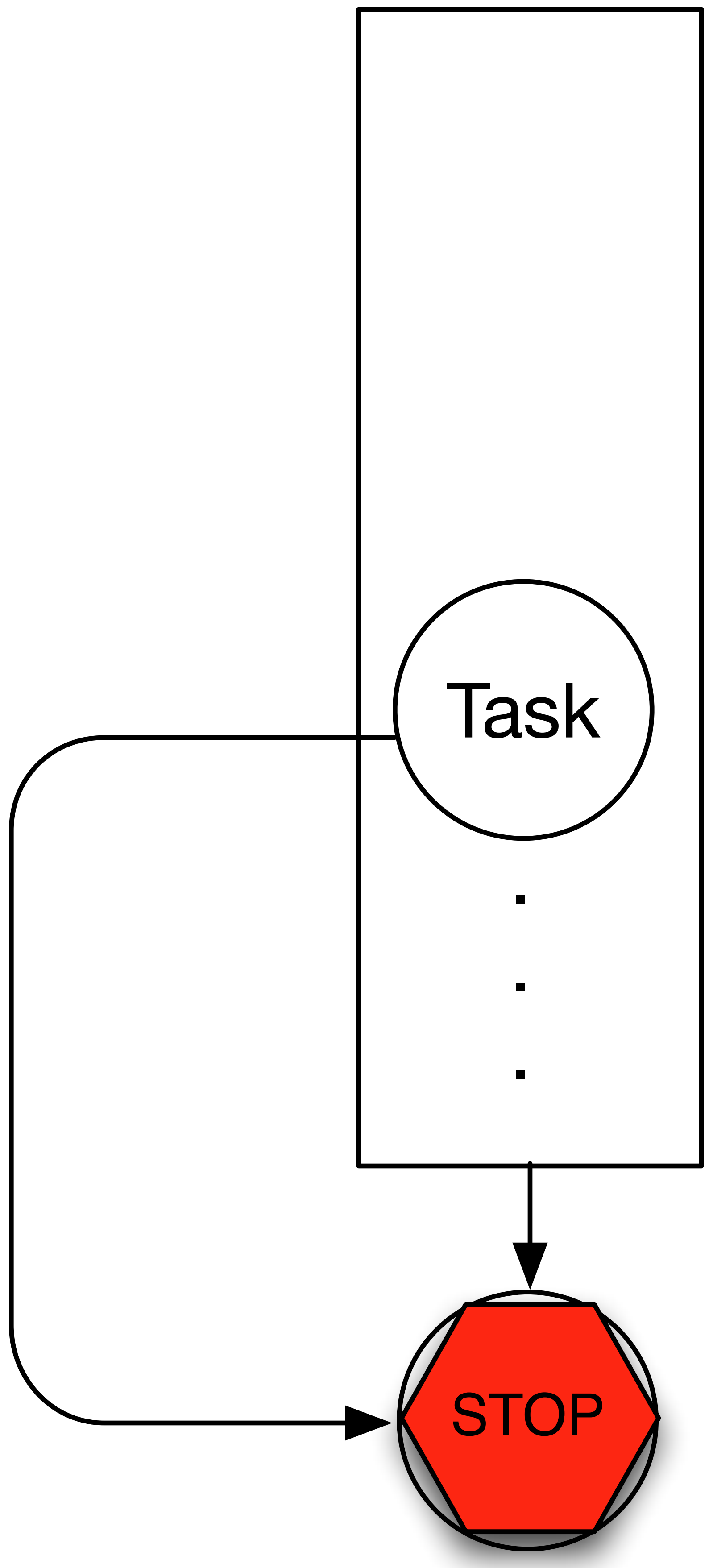
Futures: get() deadlock



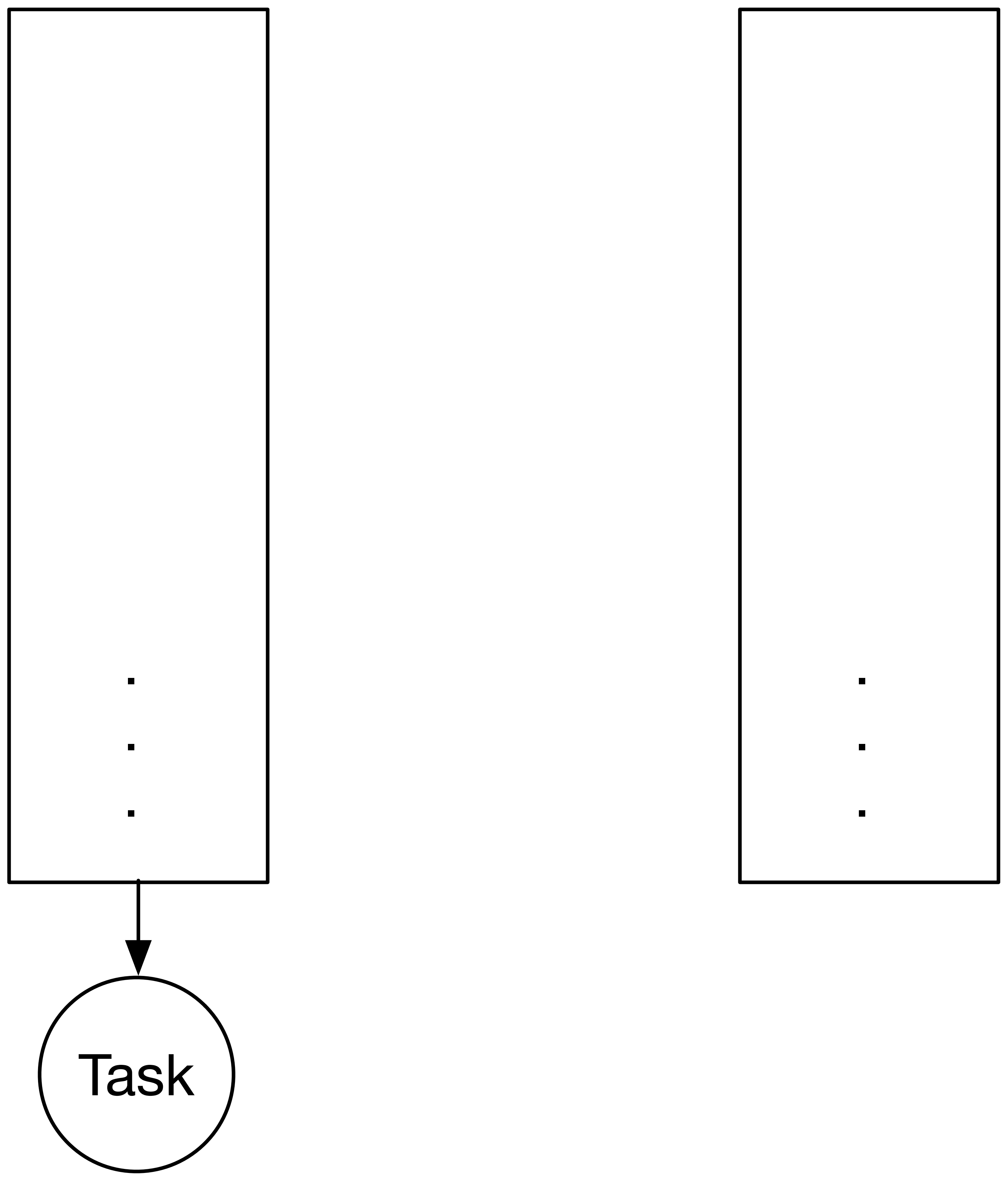
Futures: get() deadlock



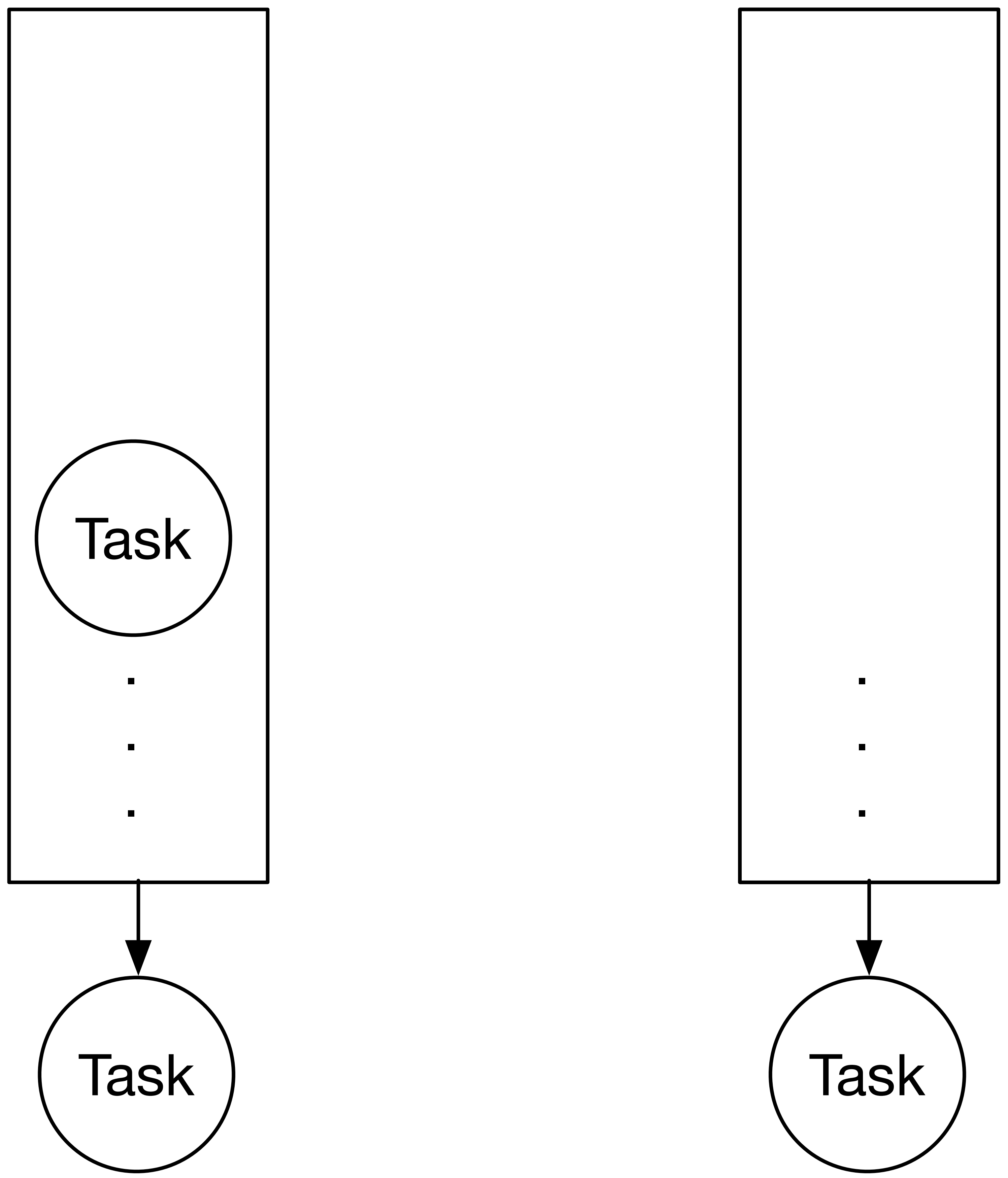
Futures: get() deadlock



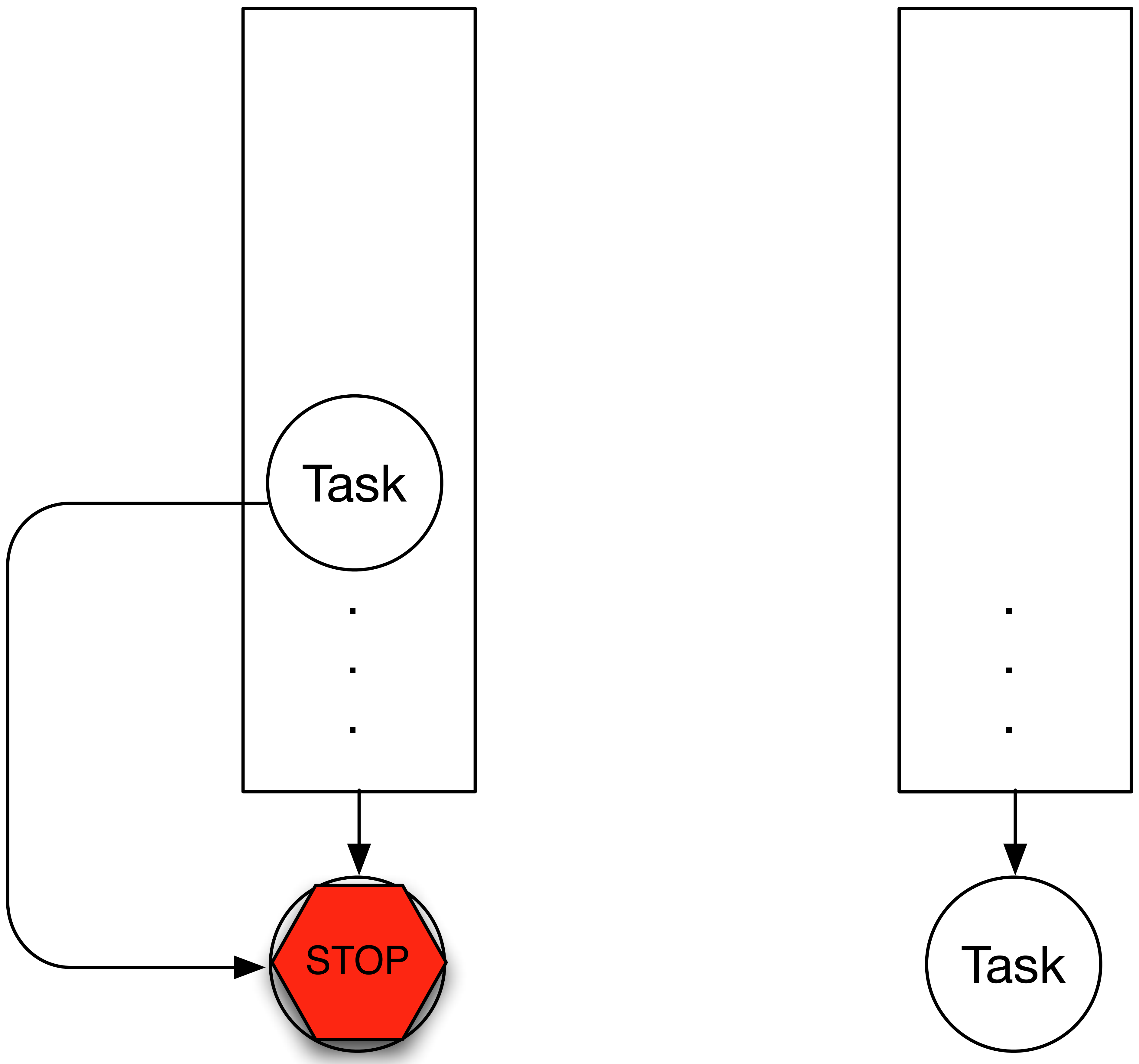
Futures: get() deadlock



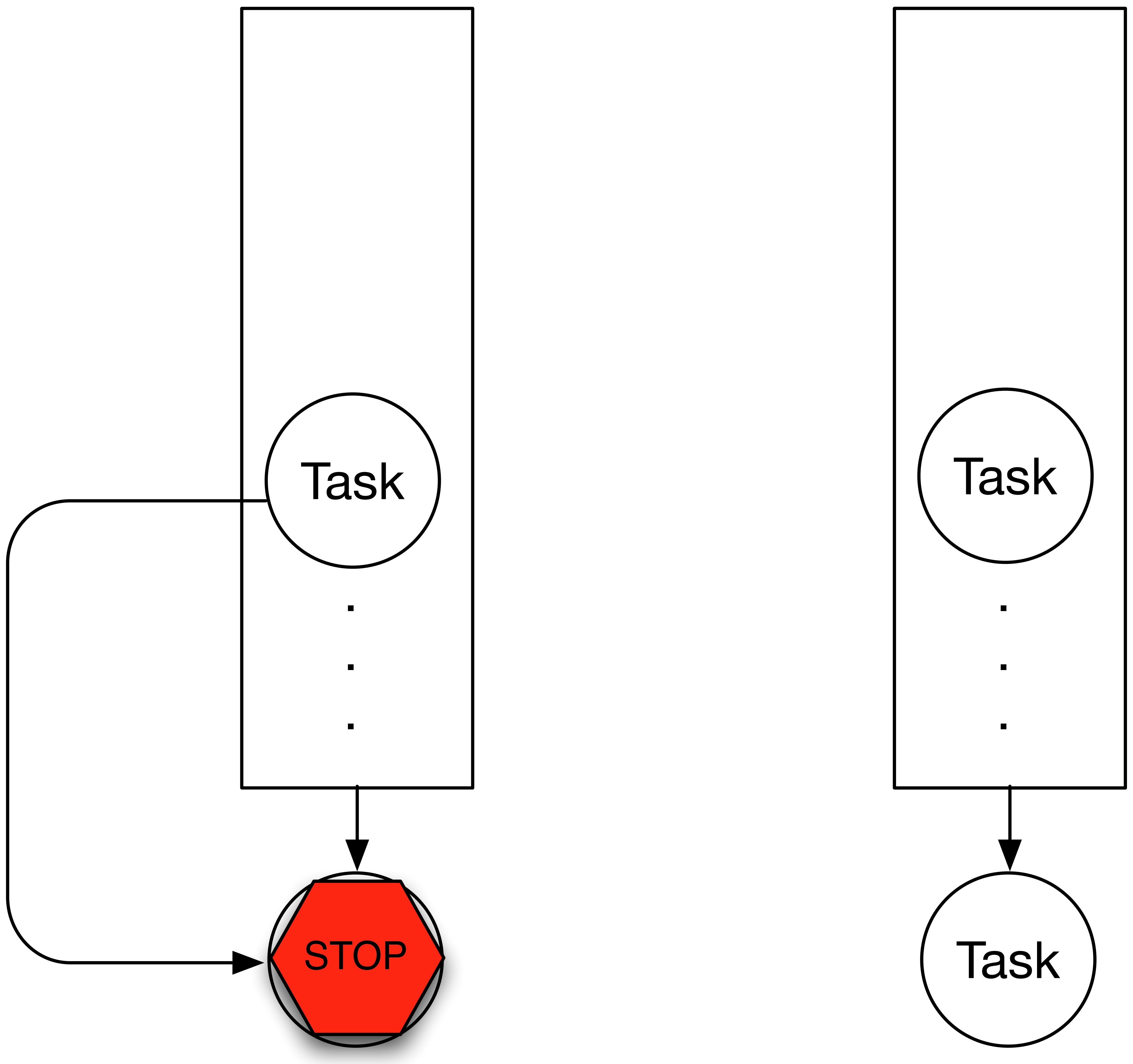
Futures: get() deadlock



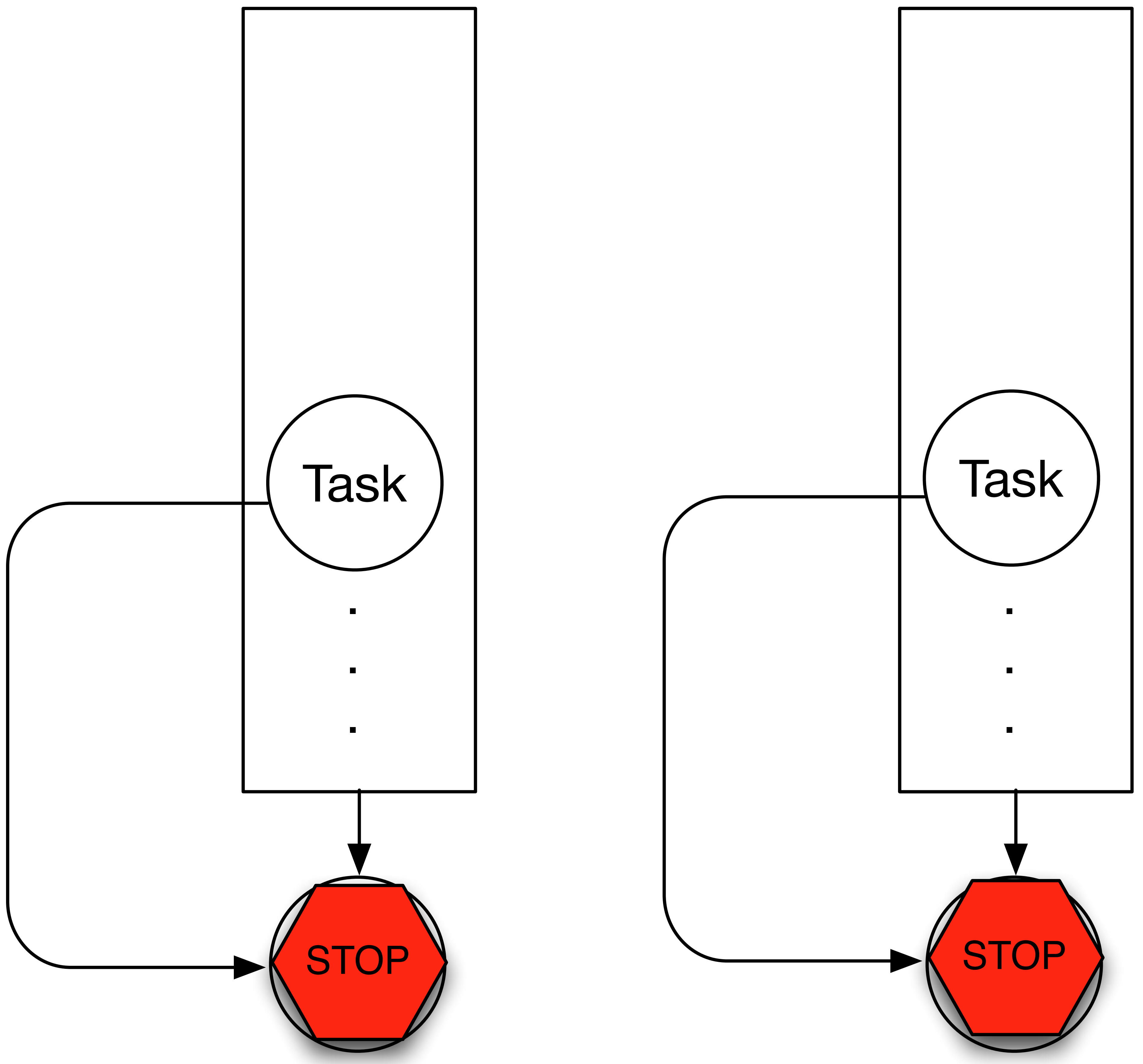
Futures: get() deadlock



Futures: get() deadlock



Futures: get() deadlock



Futures: Continuations

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).

Futures: Continuations

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.

https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html

Futures: Continuations

```
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: Continuations

```
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();
```

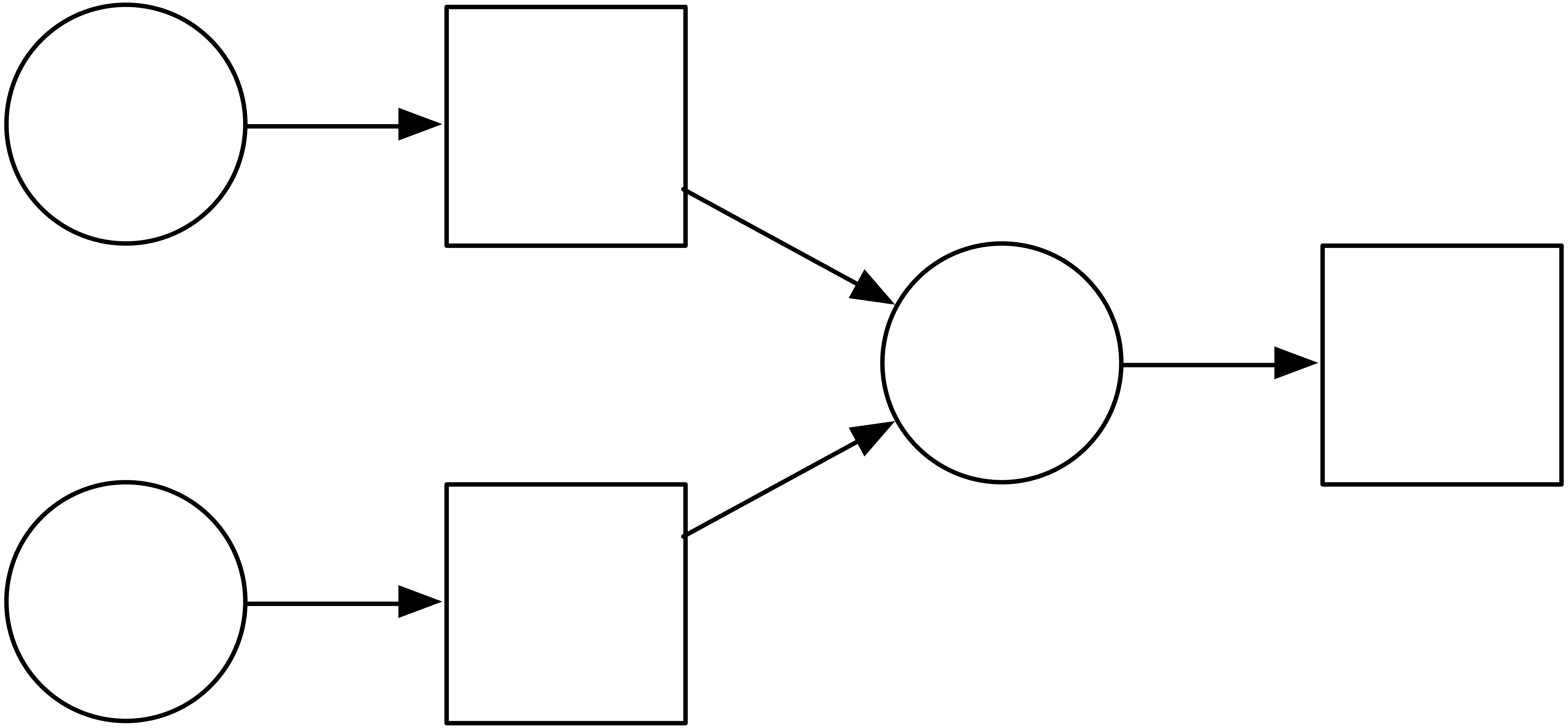
```
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209
689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

Futures vs Completion Handlers

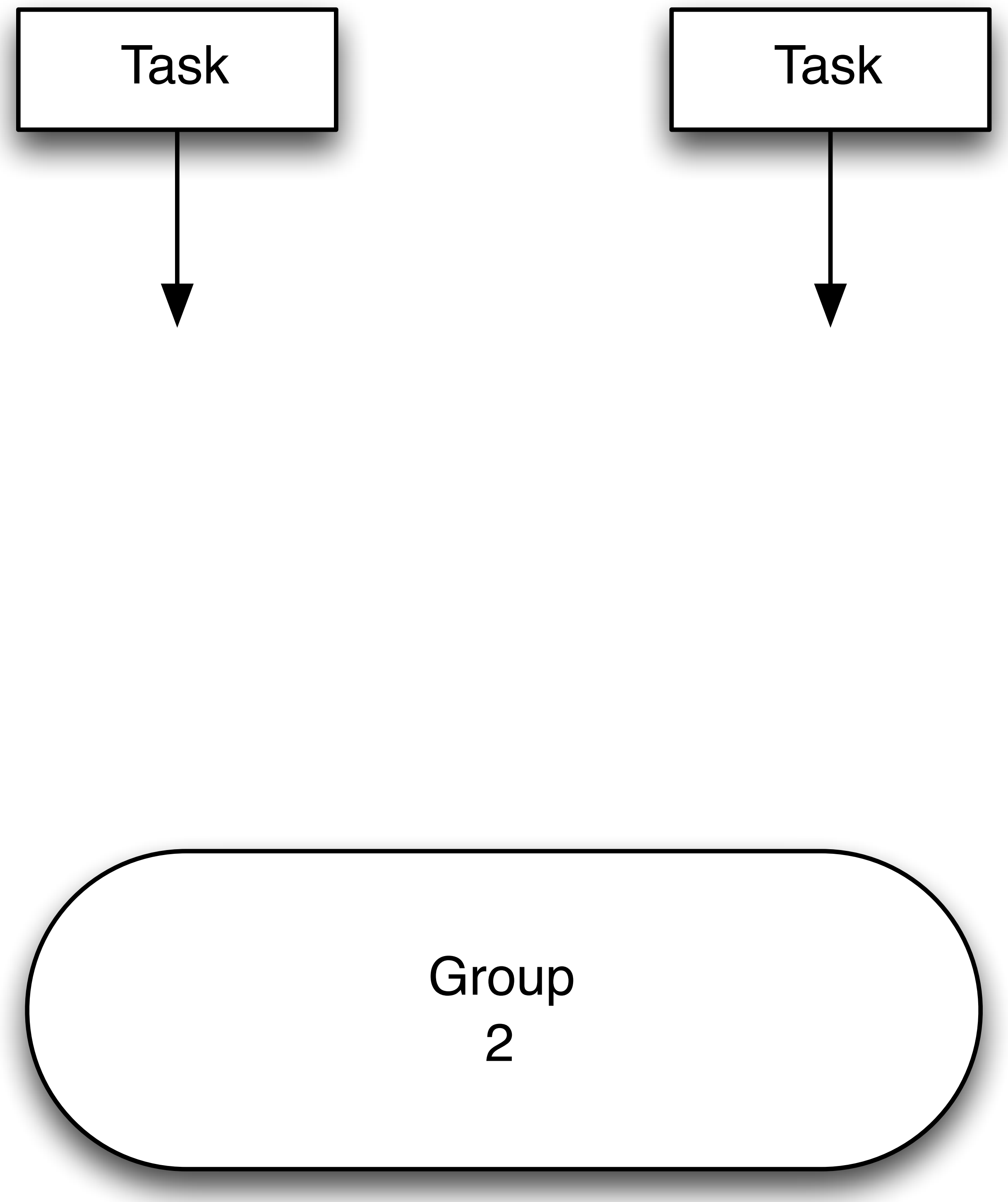
- Completion handlers are callbacks, they must be known prior to the call
  - No need to synchronize between invoking and setting the continuation

- Futures allow setting the continuation after the sending call is in flight
  - Simpler to compose
  - Require synchronization between invoking and setting the continuation

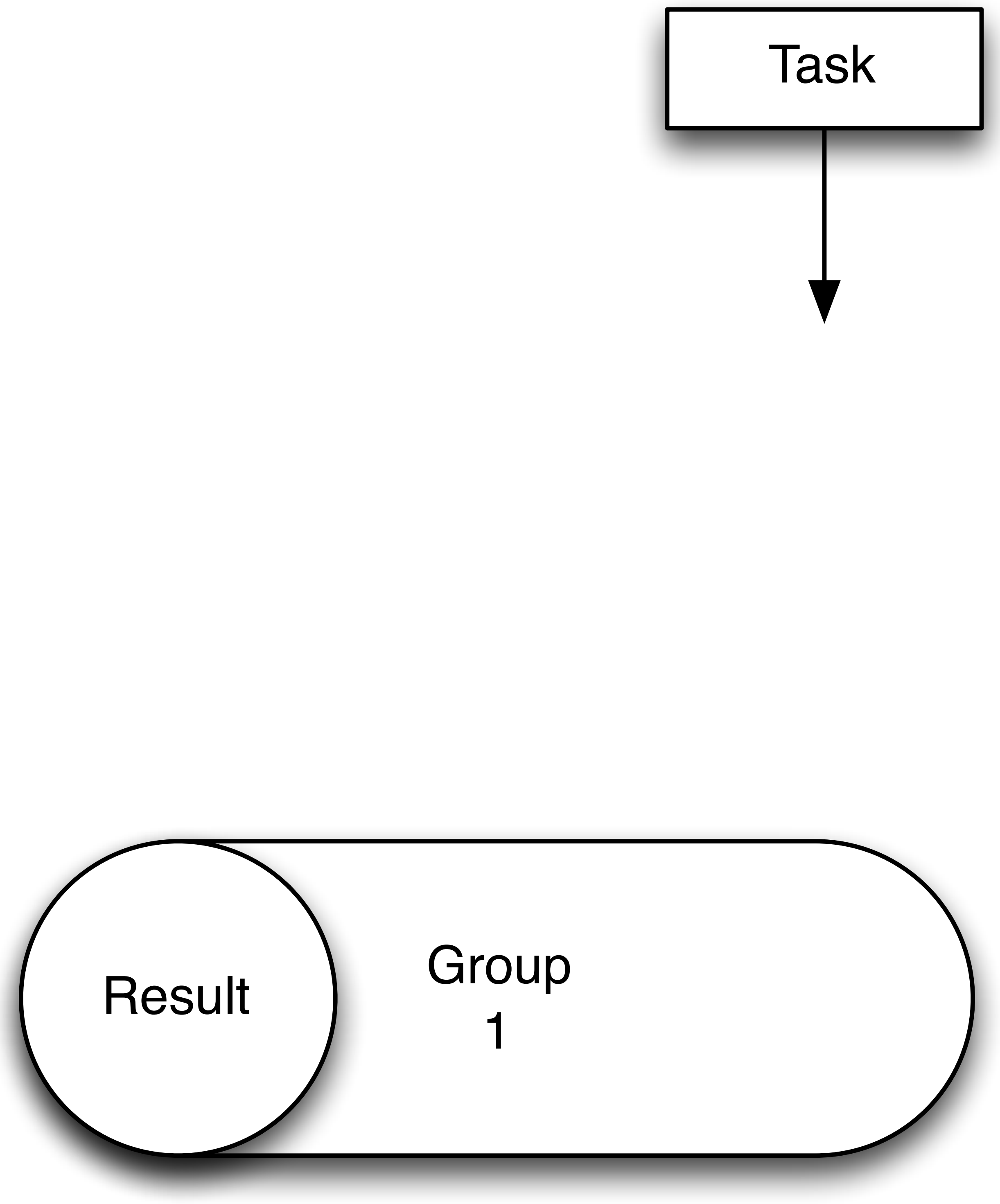
Futures: Joins



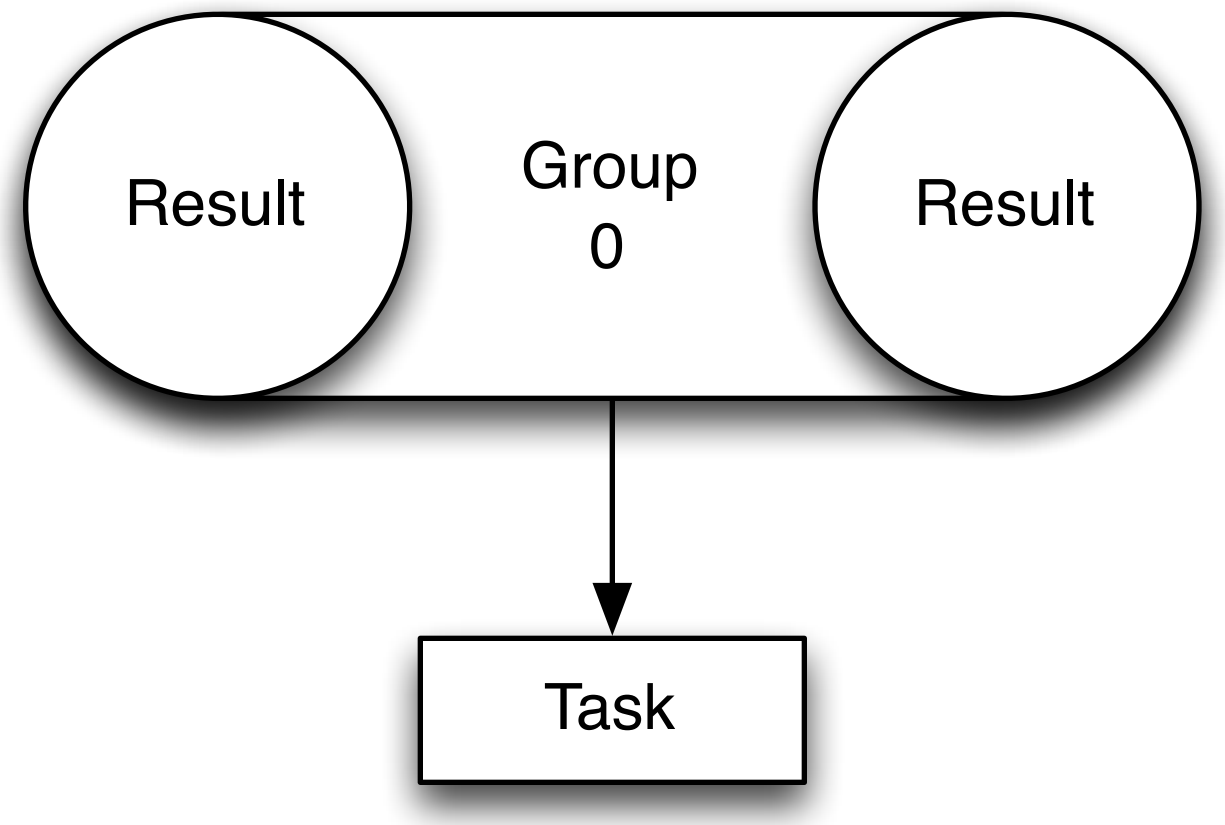
Task Systems



Task Systems



Task Systems



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;
```

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

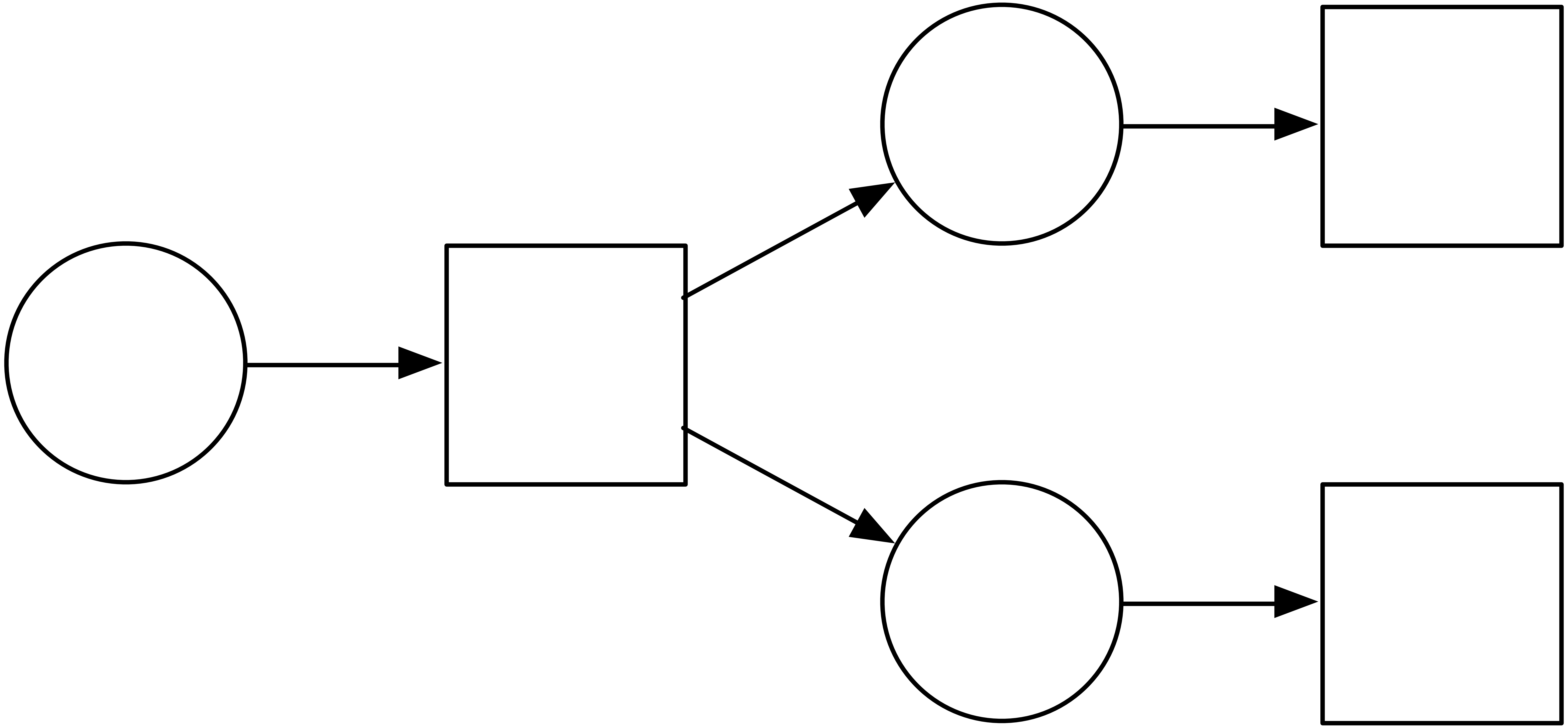
Futures: Continuations

```
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

```
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

```
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); });
```

Thread 1: signal SIGABRT

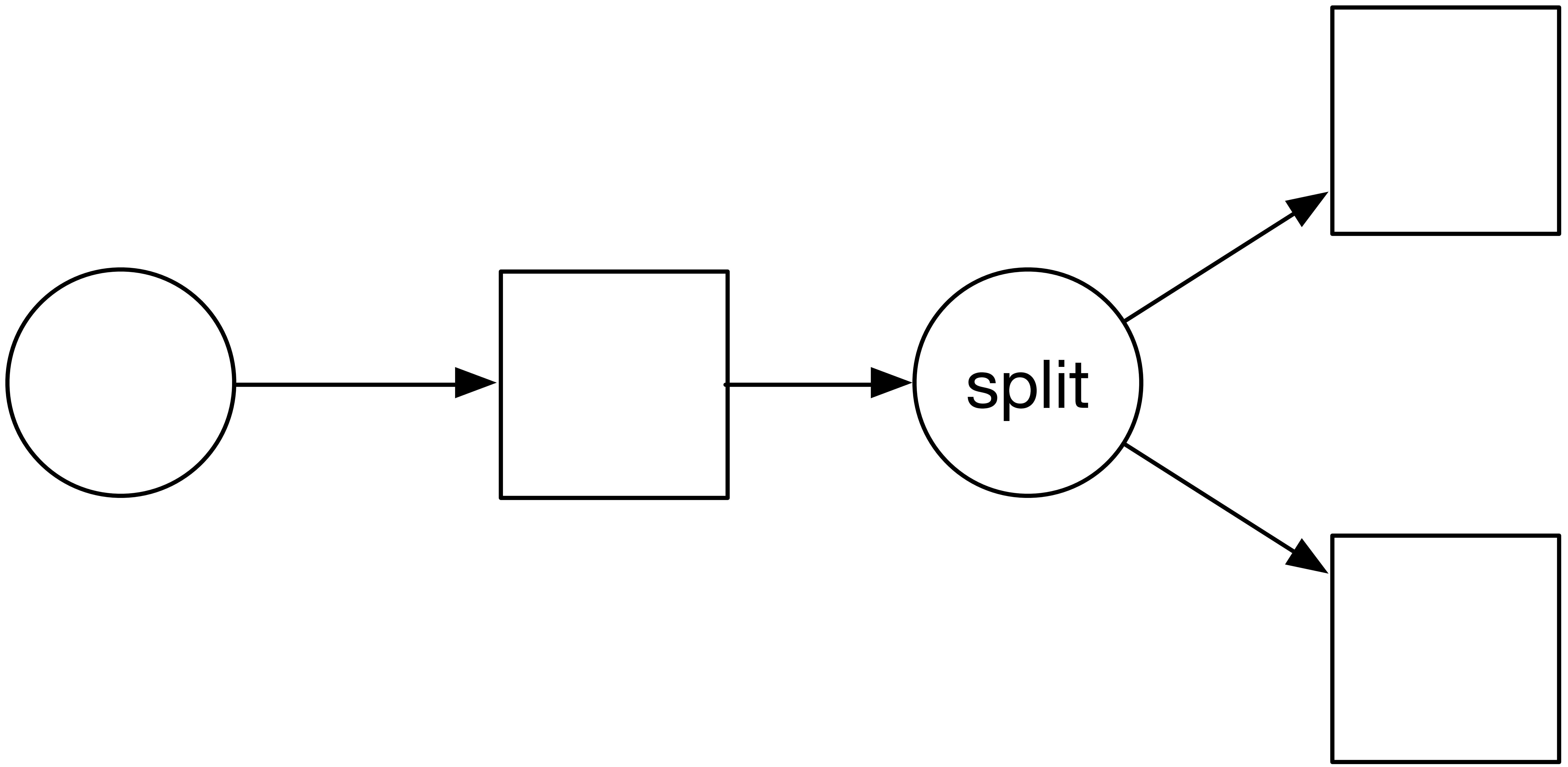
```
Assertion failed: (px != 0), function operator->, file shared_ptr.hpp, line 648.
```

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

Futures: Continuations

```
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); });
```


Adobe

Futures: Continuations

```
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

```
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 formed a packaged task
  - Packaged tasks handle the exception marshalling through a promise

Promise

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

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

Promise

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

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

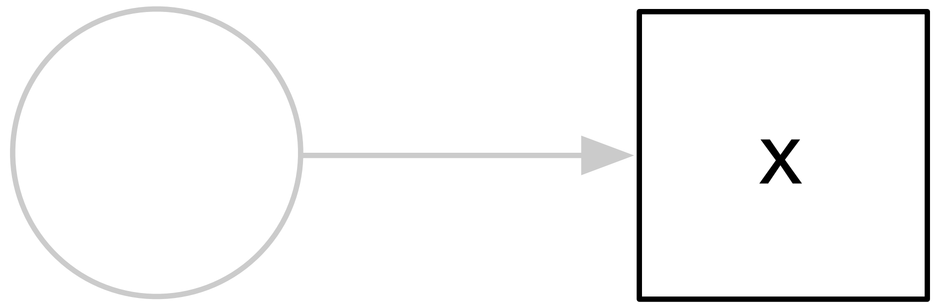
42

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
    });
}
```

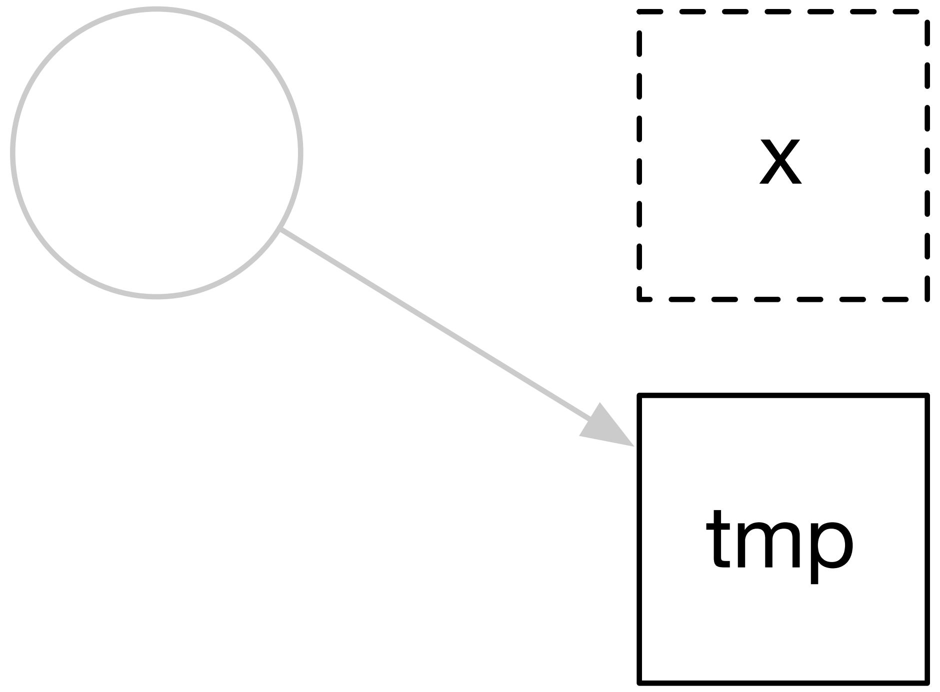
Futures: Split

```
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
    });
}
```



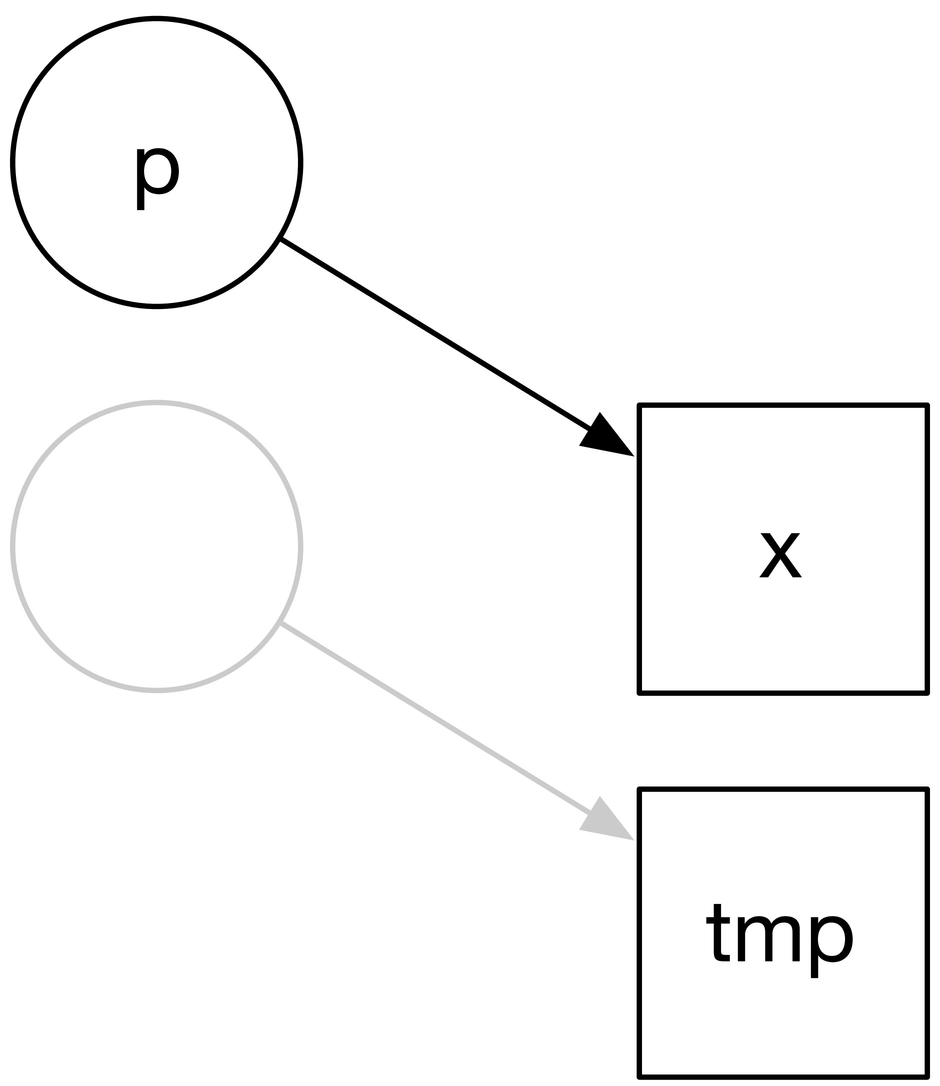
Futures: Split

```
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
    });
}
```



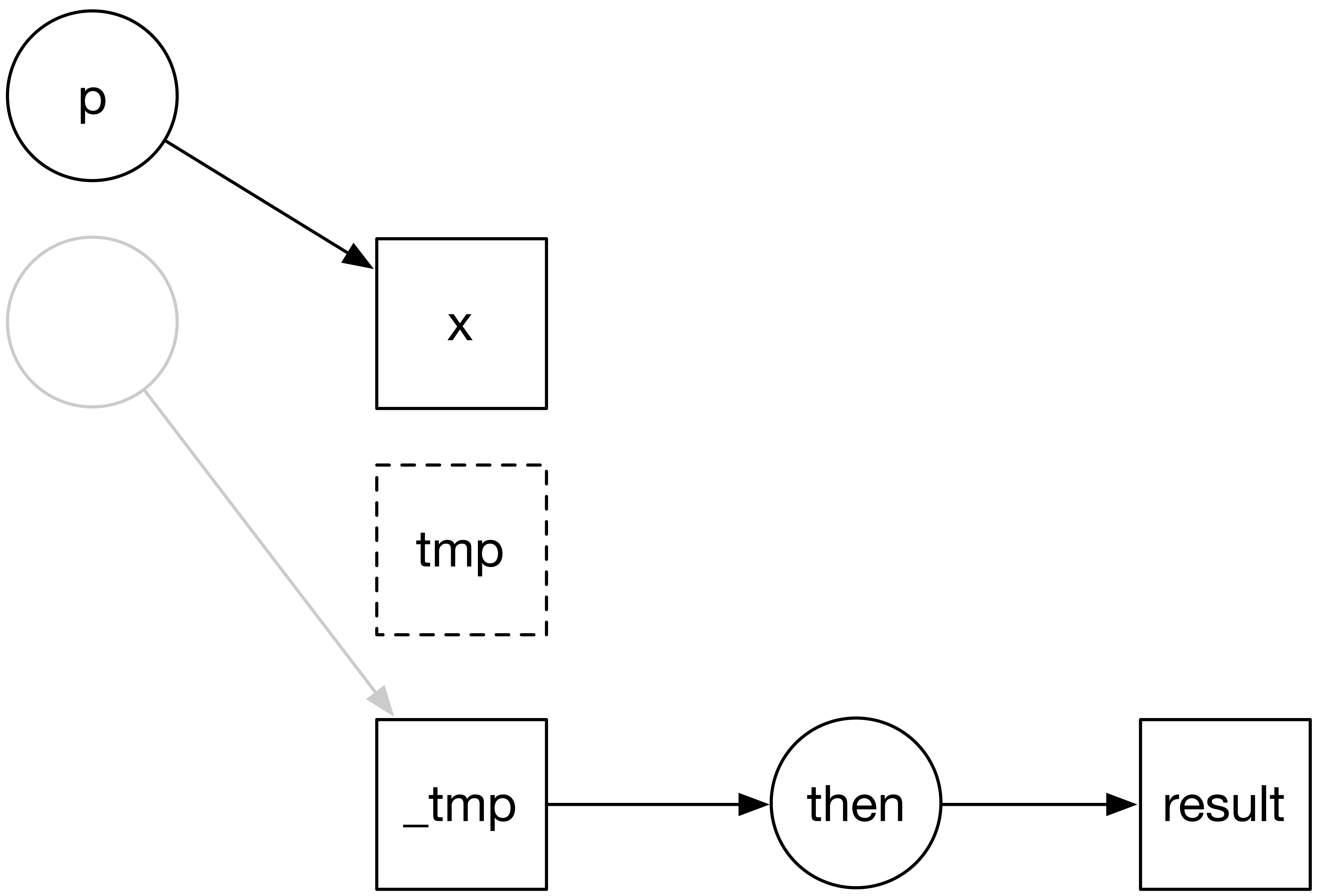
Futures: Split

```
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
    });
}
```



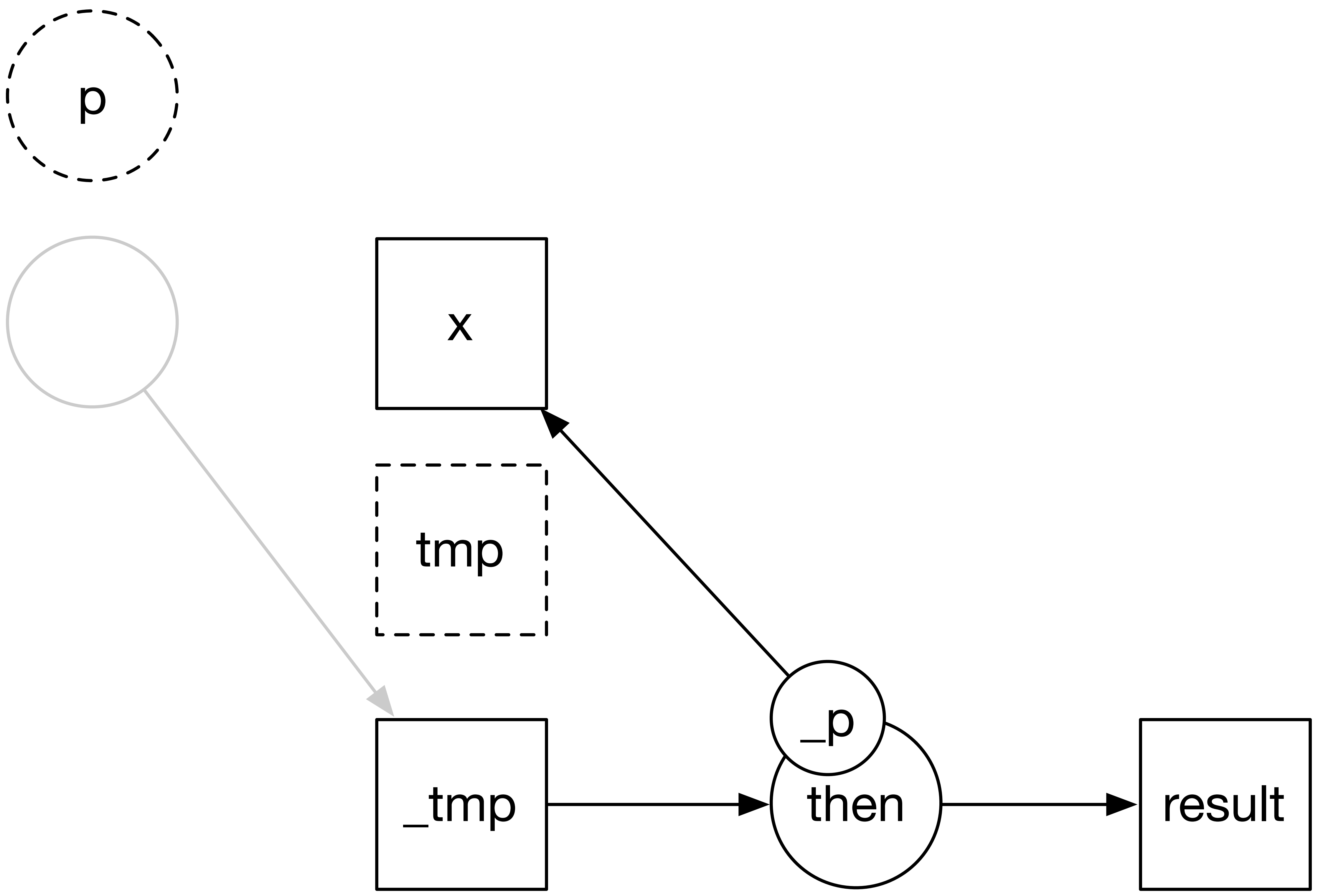
Futures: Split

```
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
    });
}
```



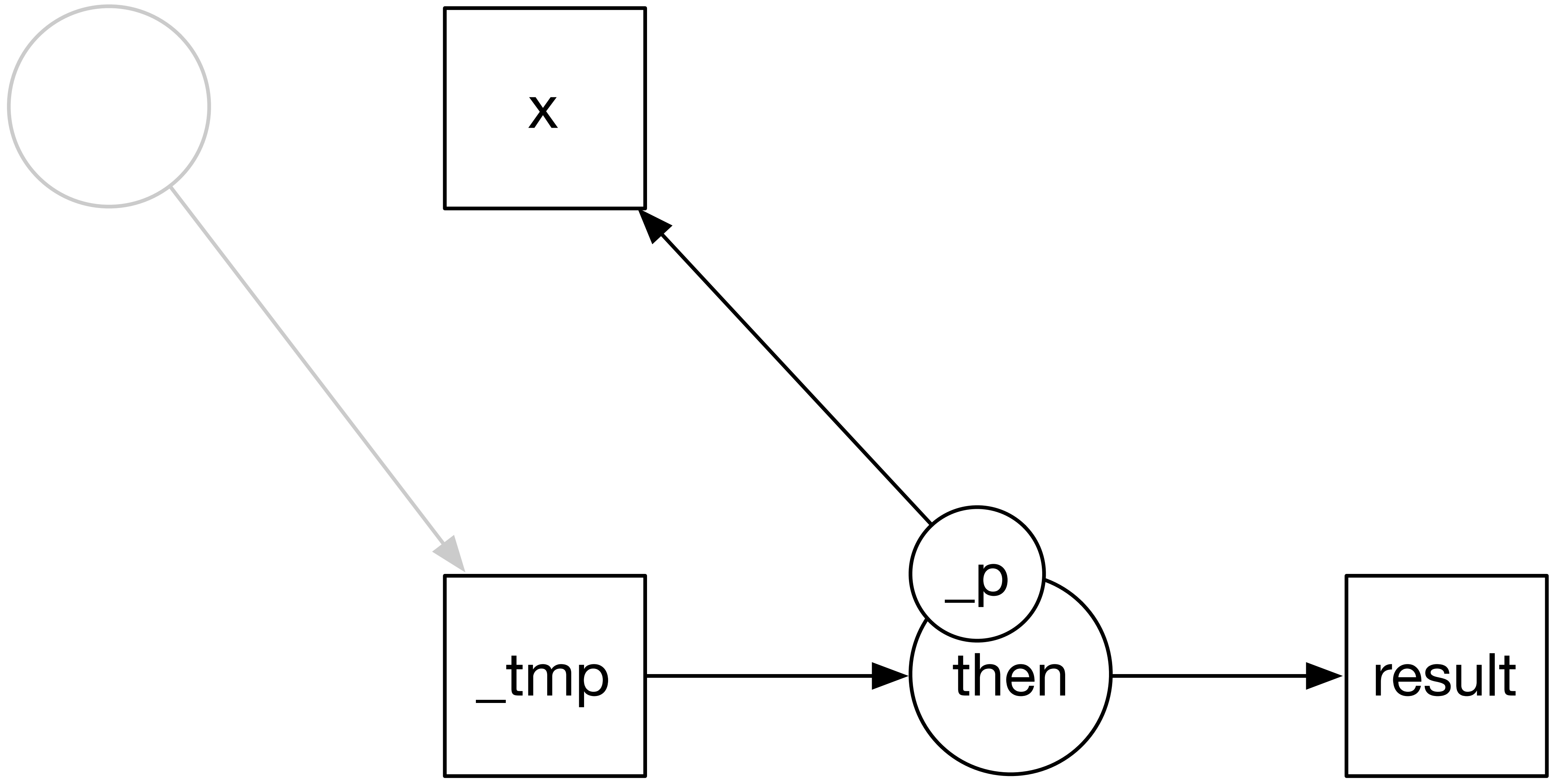
Futures: Split

```
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
    });
}
```



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 = 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
    });
}
```

Futures: Continuations

```
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

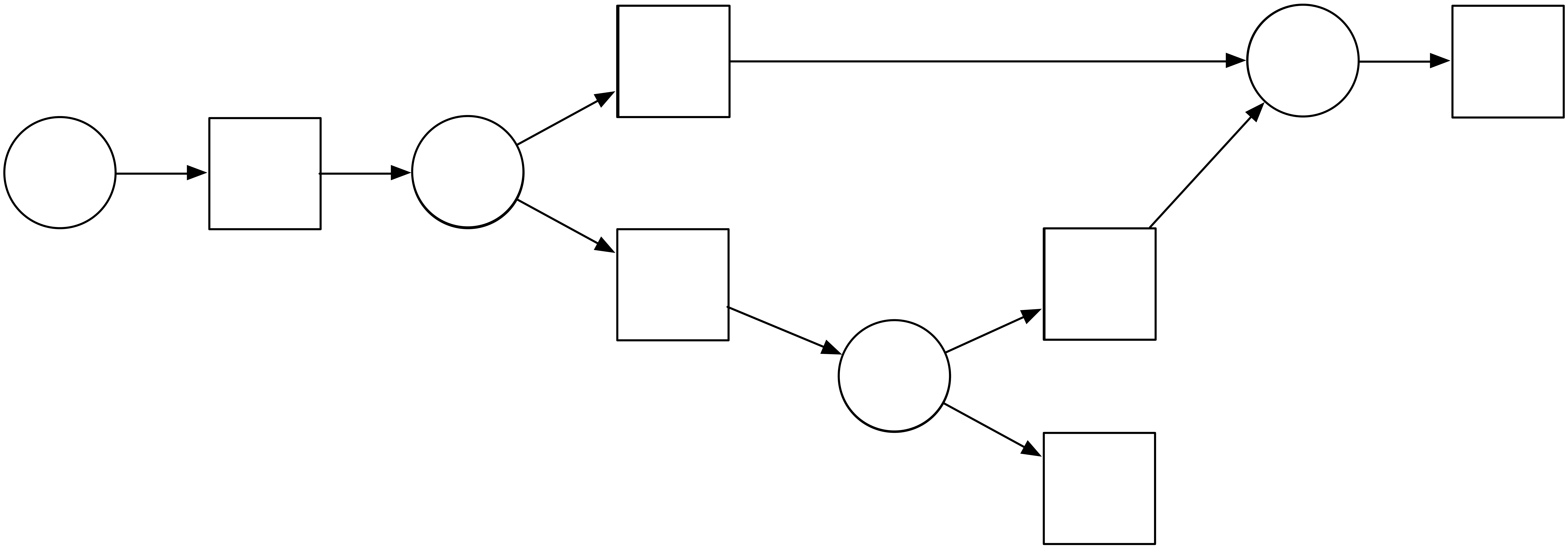
- When the (last) future destructs
  - The associated task that has not started, should not execute (NOP)
  - The resource held by that task should be released
    - Since that task may hold futures for other tasks, the system unravels

Cancelation

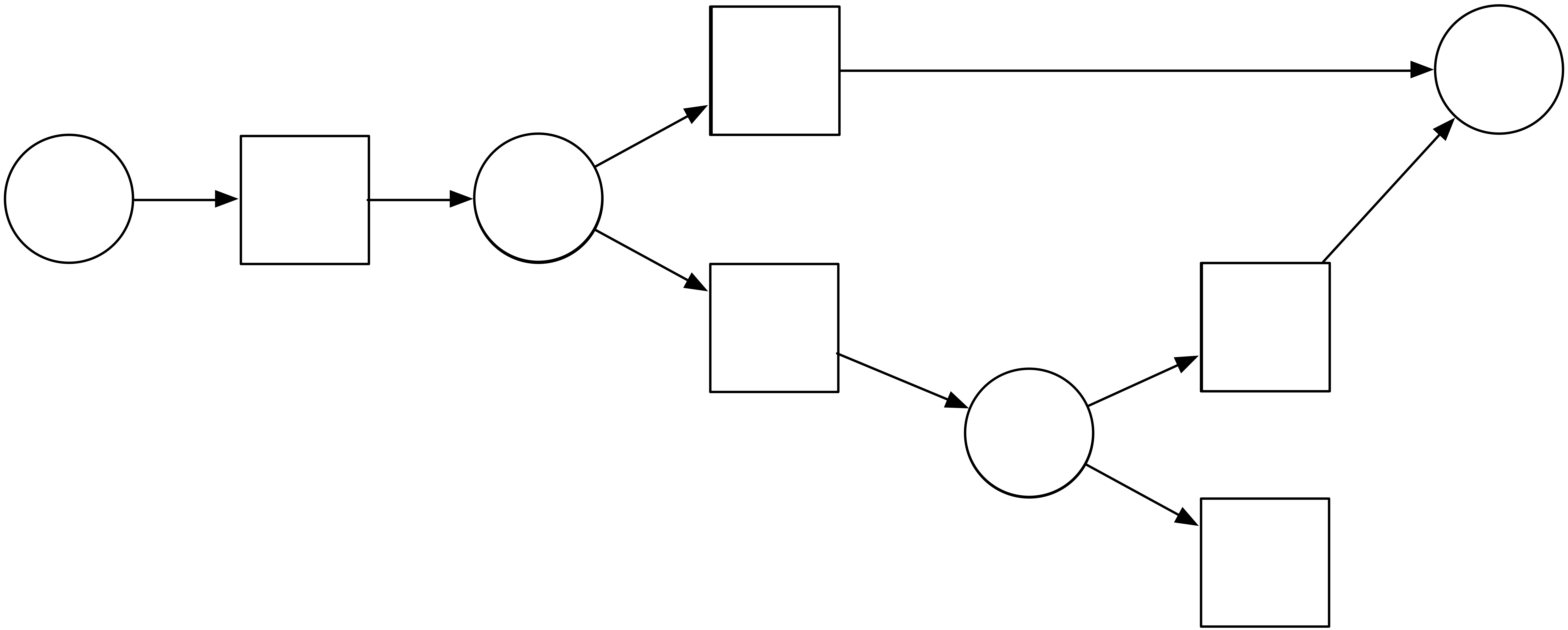
- When the (last) future destructs
  - The associated task that has not started, should not execute (NOP)
  - The resource held by that task should be released
    - Since that task may hold futures for other tasks, the system unravels

- 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

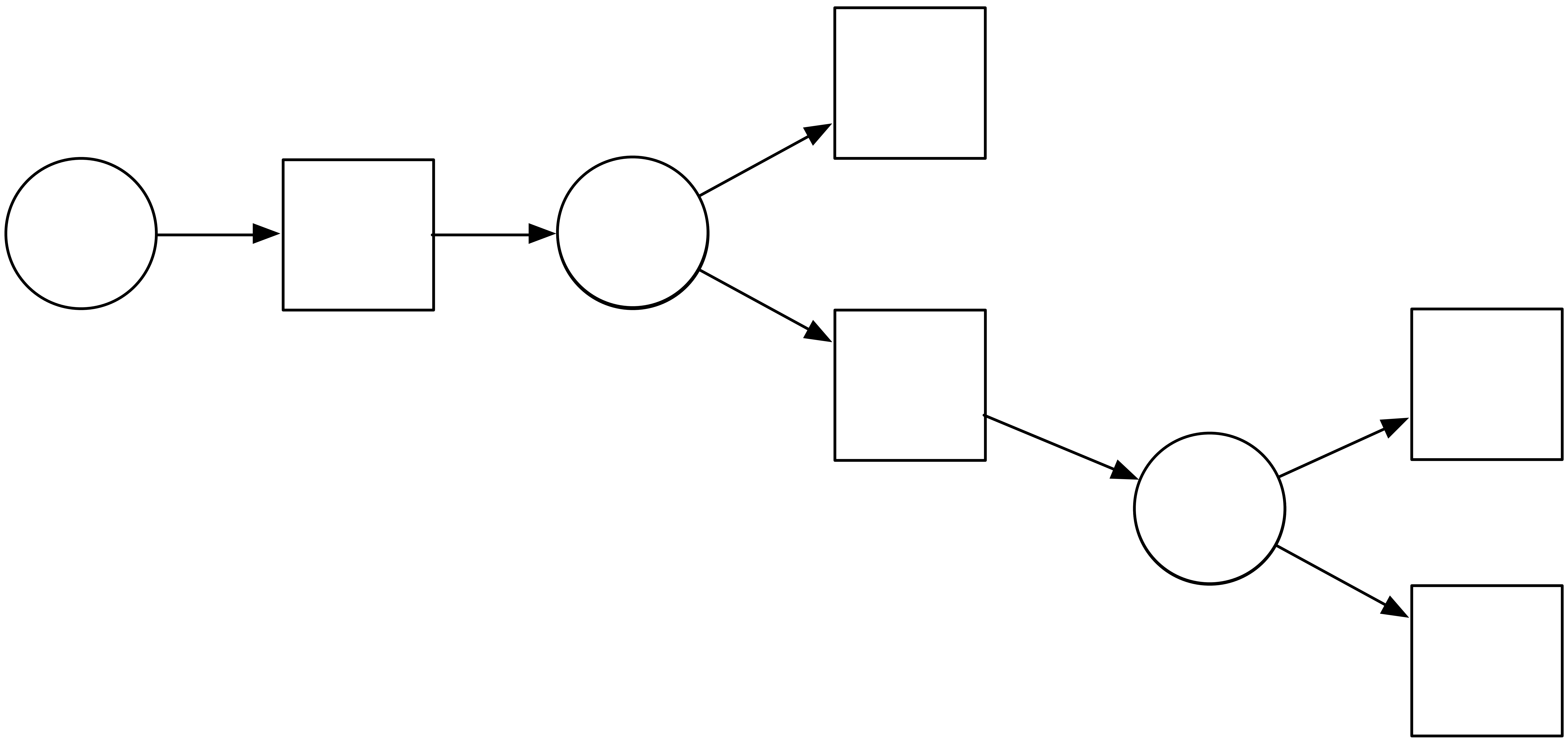
Cancelation



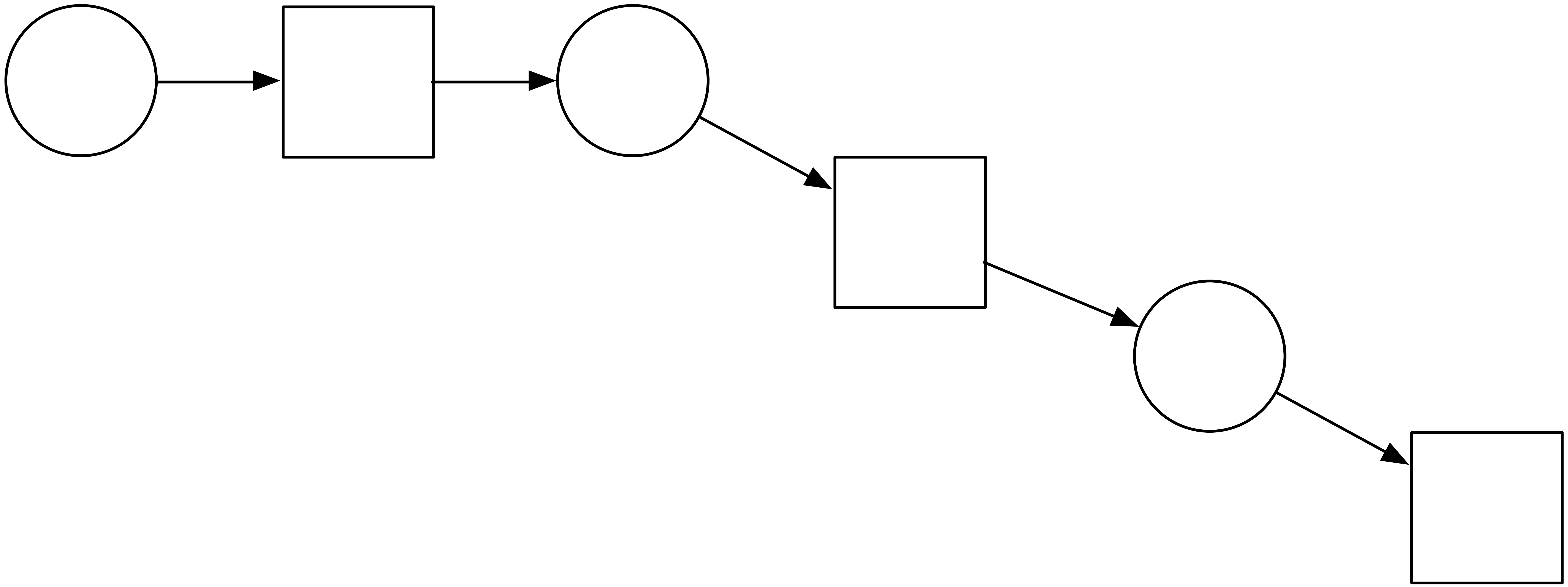
Cancelation



Cancelation



Cancelation



stlab future library

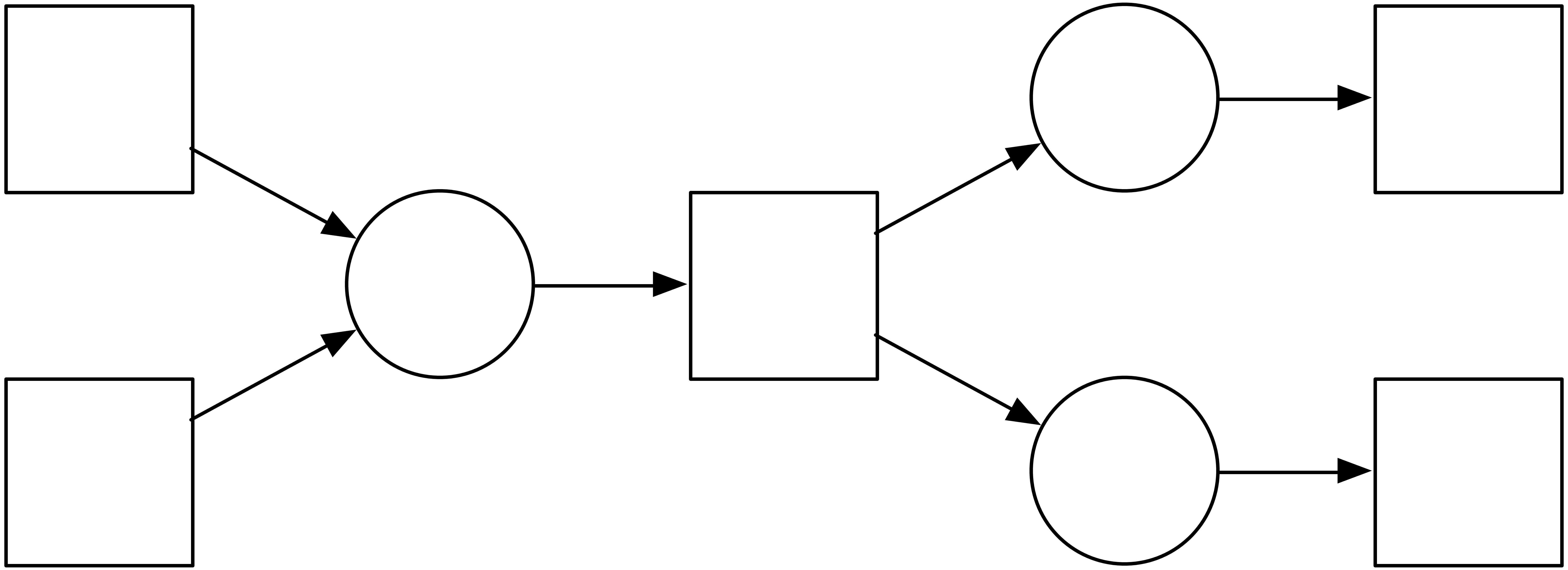
stlab future library

- 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
- 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

Channels

```
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();
```

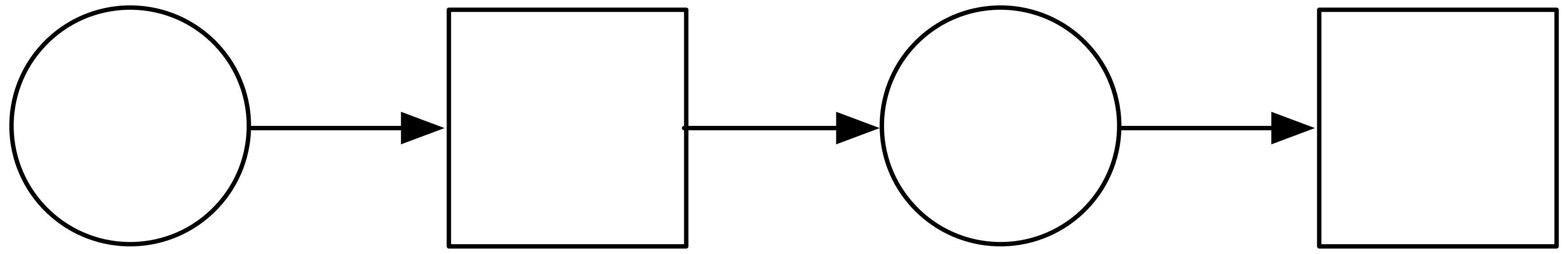
```
channel<int> send;

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();
```

6

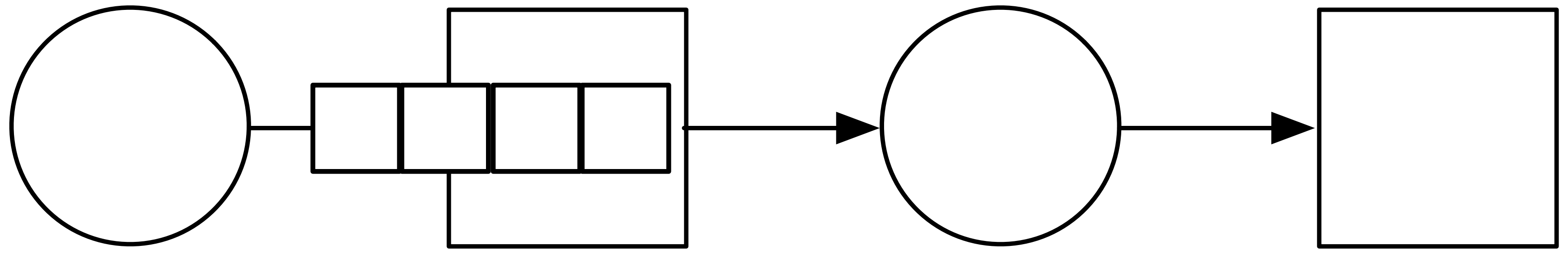
Channels

```
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; }
};
```

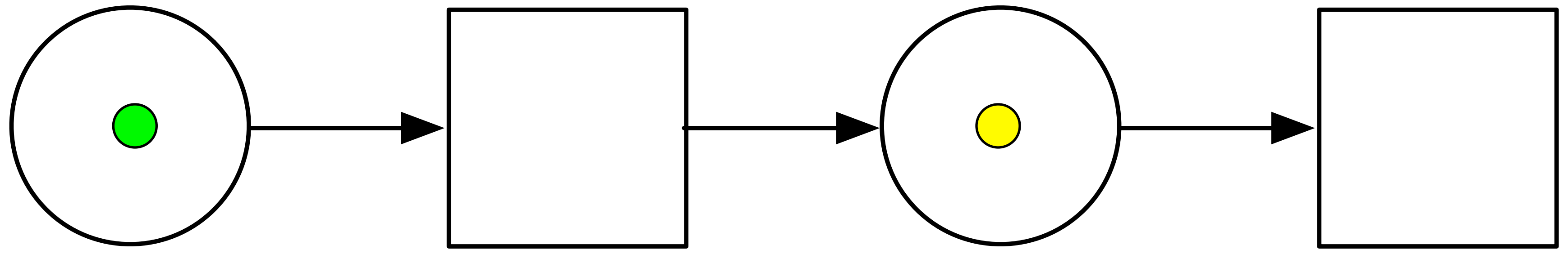
Flow Control



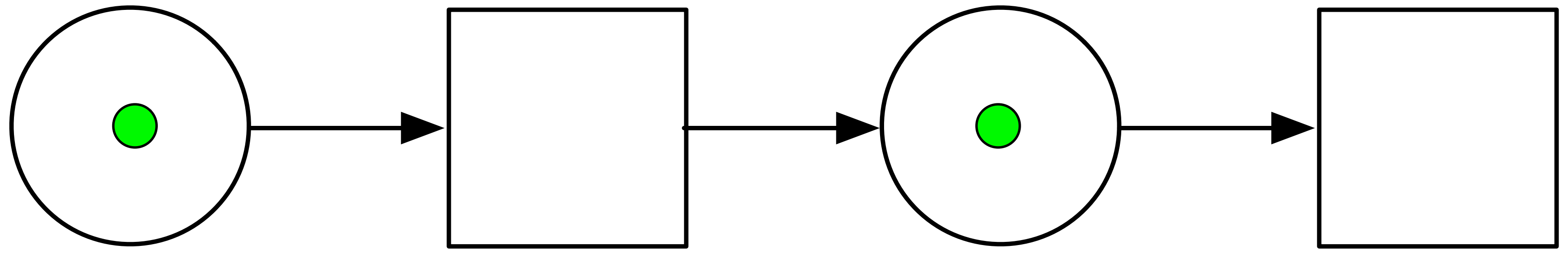
Flow Control



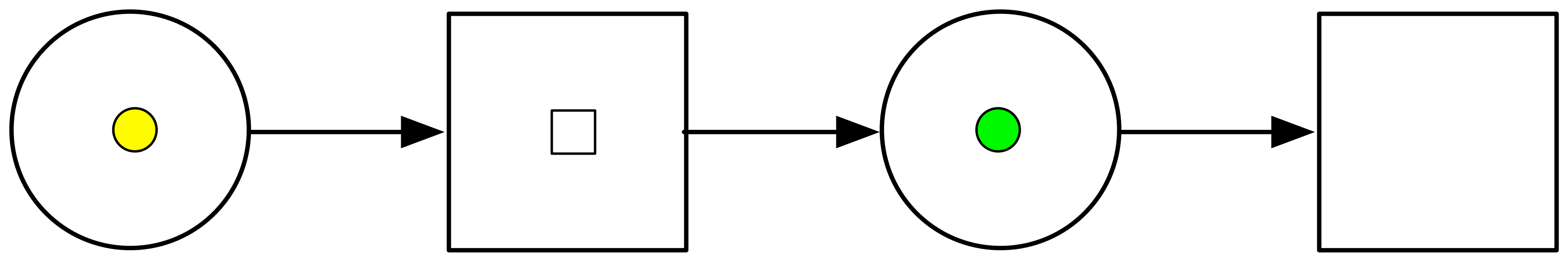
Flow Control



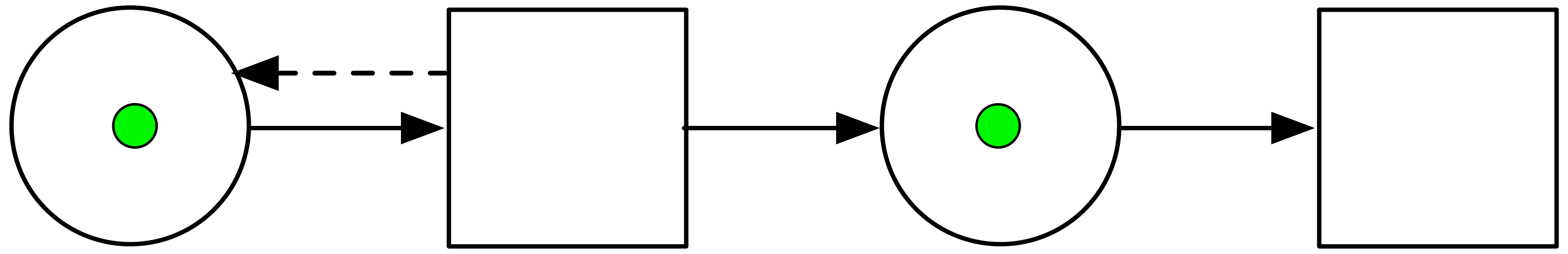
Flow Control



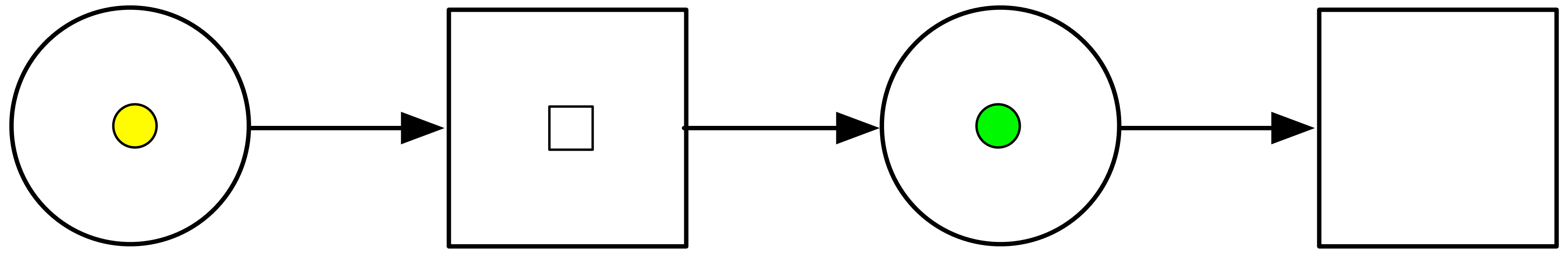
Flow Control



Flow Control



Flow Control



lightroom.adobe.com
Exit
Revert to Original
Undo Edits
CROP
PRESETS
ADJUST
HISTOGRAM
TREATMENT
Color
Black & White
WHITE BALANCE
TONE
Auto
Exposure
Contrast
Highlights
Shadows
Whites
Blacks
PRESENCE
Clarity
Vibrance
Saturation
COLOR / B&W
SPLIT TONING

Channels

```
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 where imperative code is not the correct approach
- Instead a graph description can be compiled and statically validated

- Slides and code from talk:
- http://sean-parent.stlab.cc/papers-and-presentations

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

- Communicating Sequential Processes (C. A. R. Hoare)
- http://usingcsp.com/cspbook.pdf

Adobe