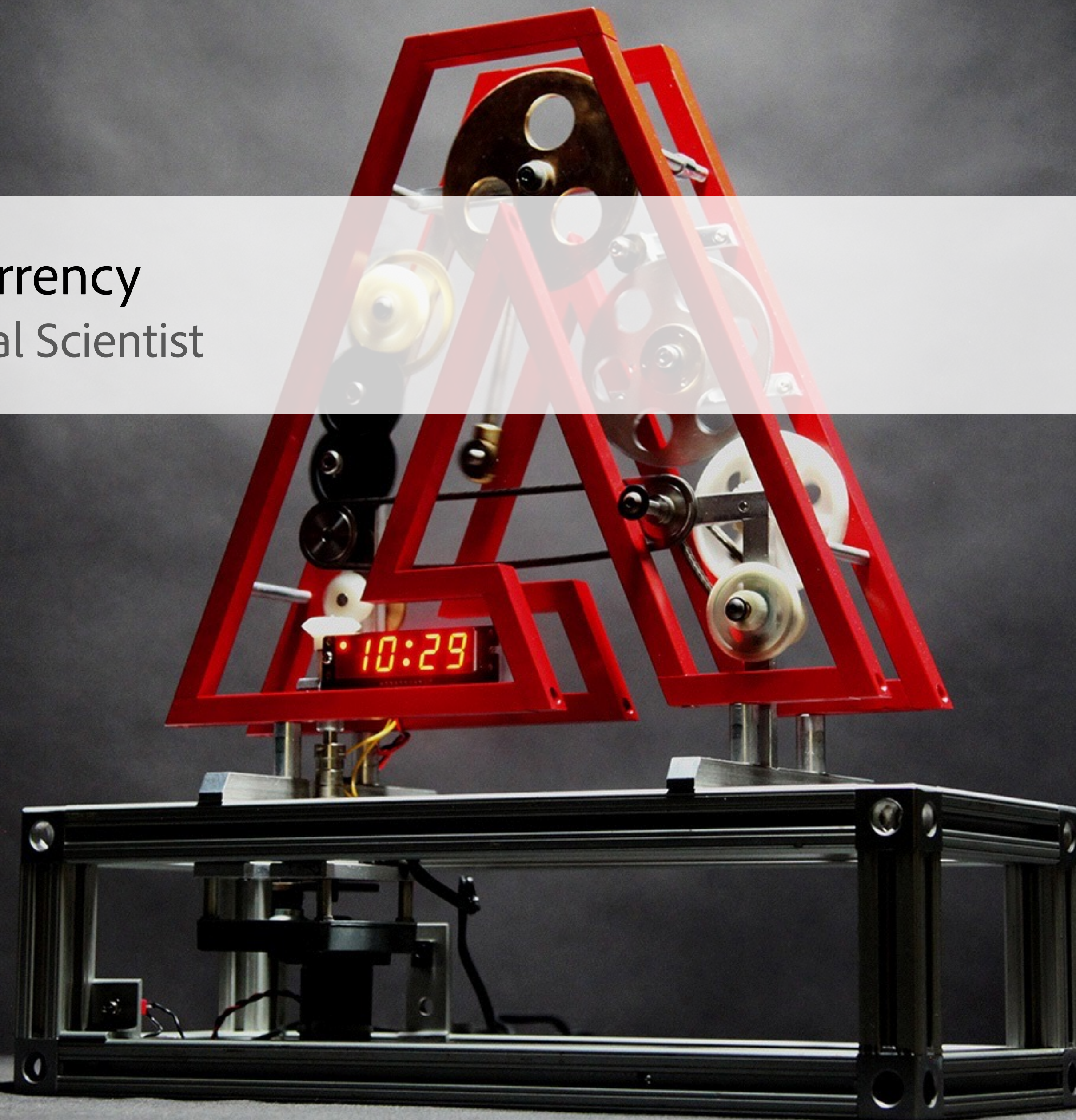




# Better Code: Concurrency

Sean Parent | Principal Scientist





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

# Common Themes

- Manage Relationships
- Understand the Fundamentals
- Code Simply

# Demo

- 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

# No Raw Synchronization Primitives

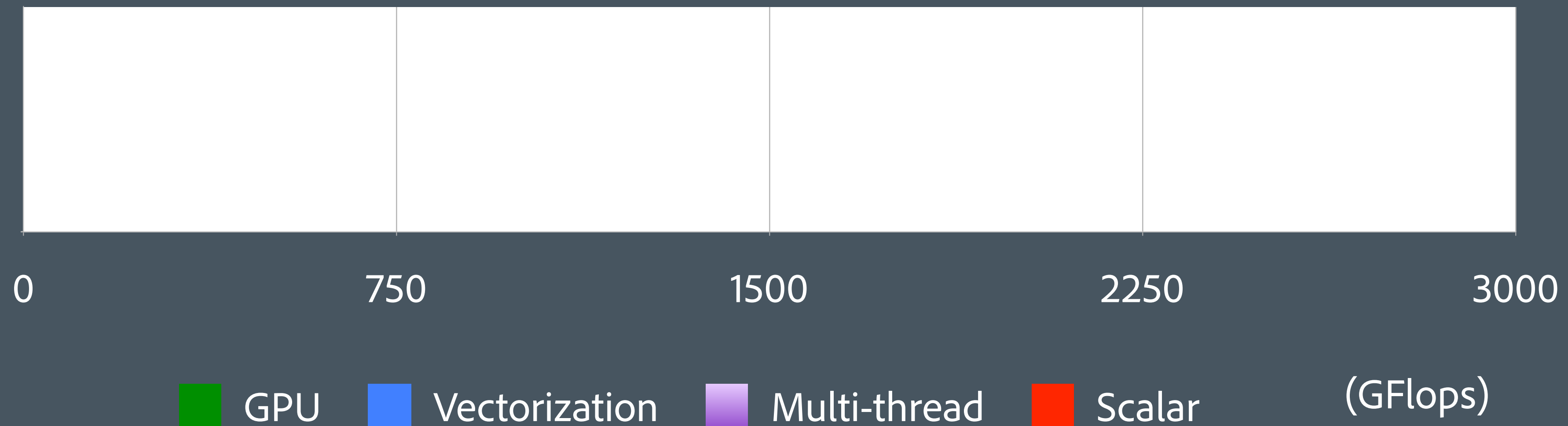
# What are raw synchronization primitives?

- Synchronization primitives are basic constructs such as:
  - Mutex
  - Atomic
  - Semaphore
  - Memory Fence
  - Condition Variable

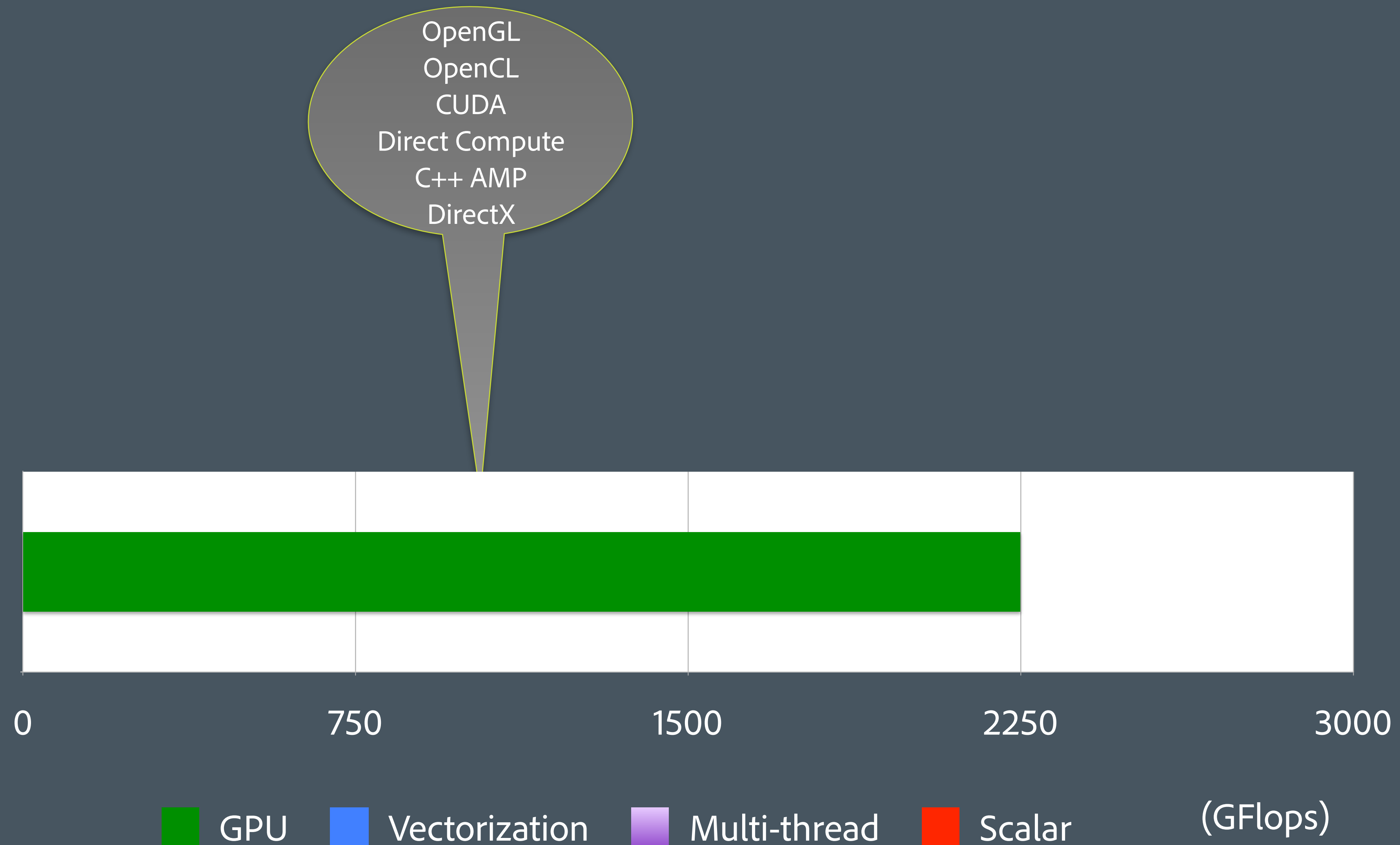
# Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



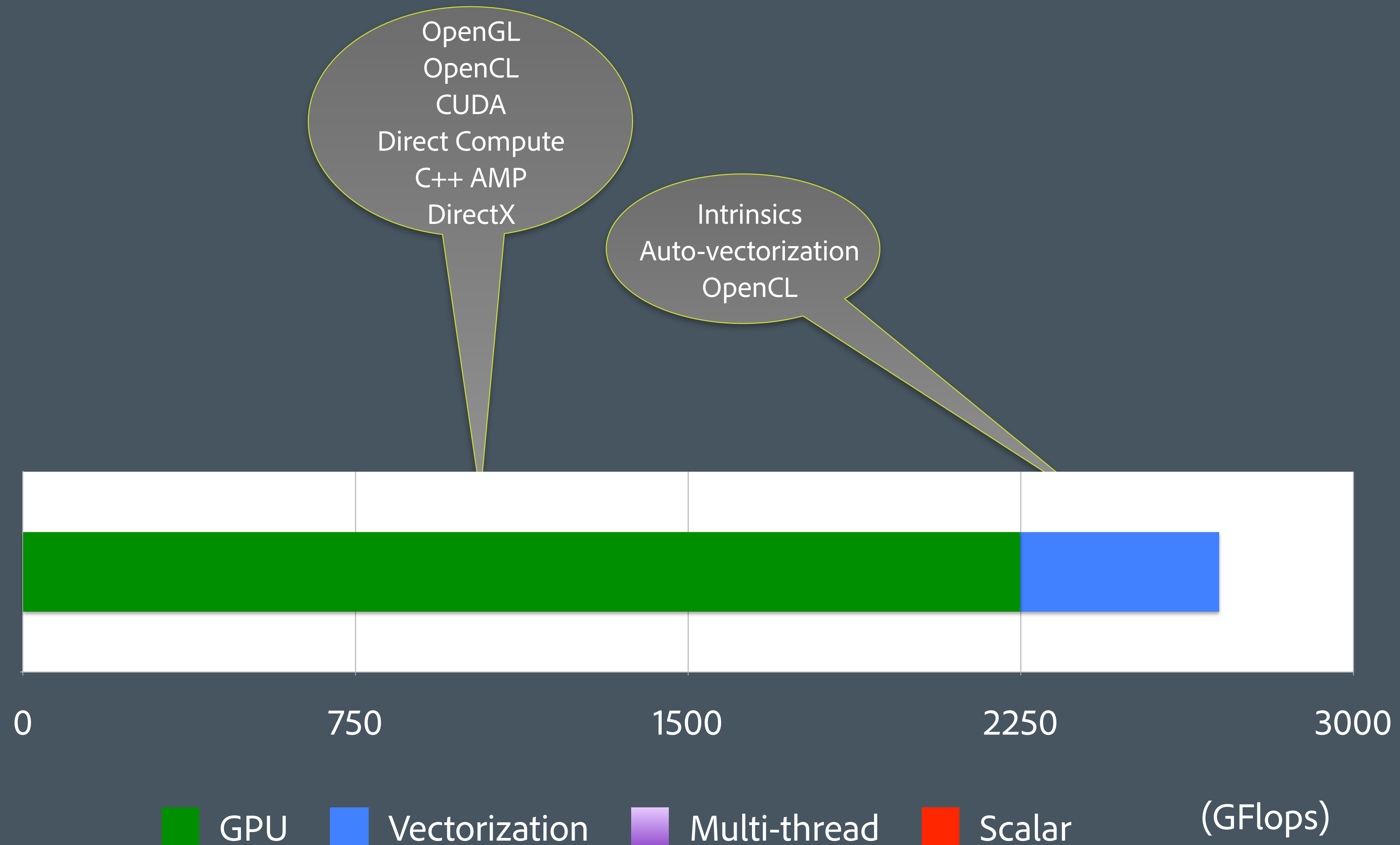
# Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



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

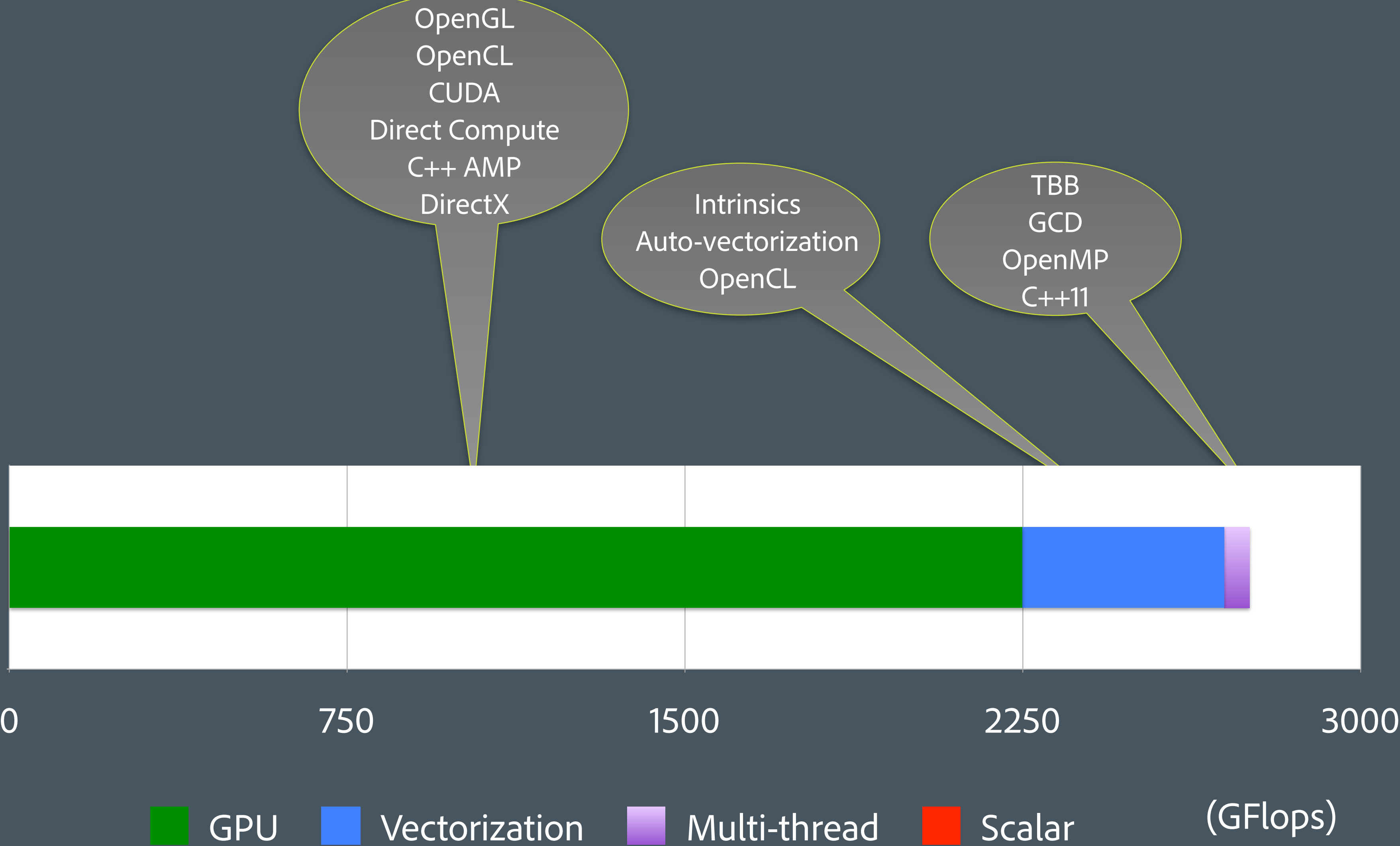


# Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)

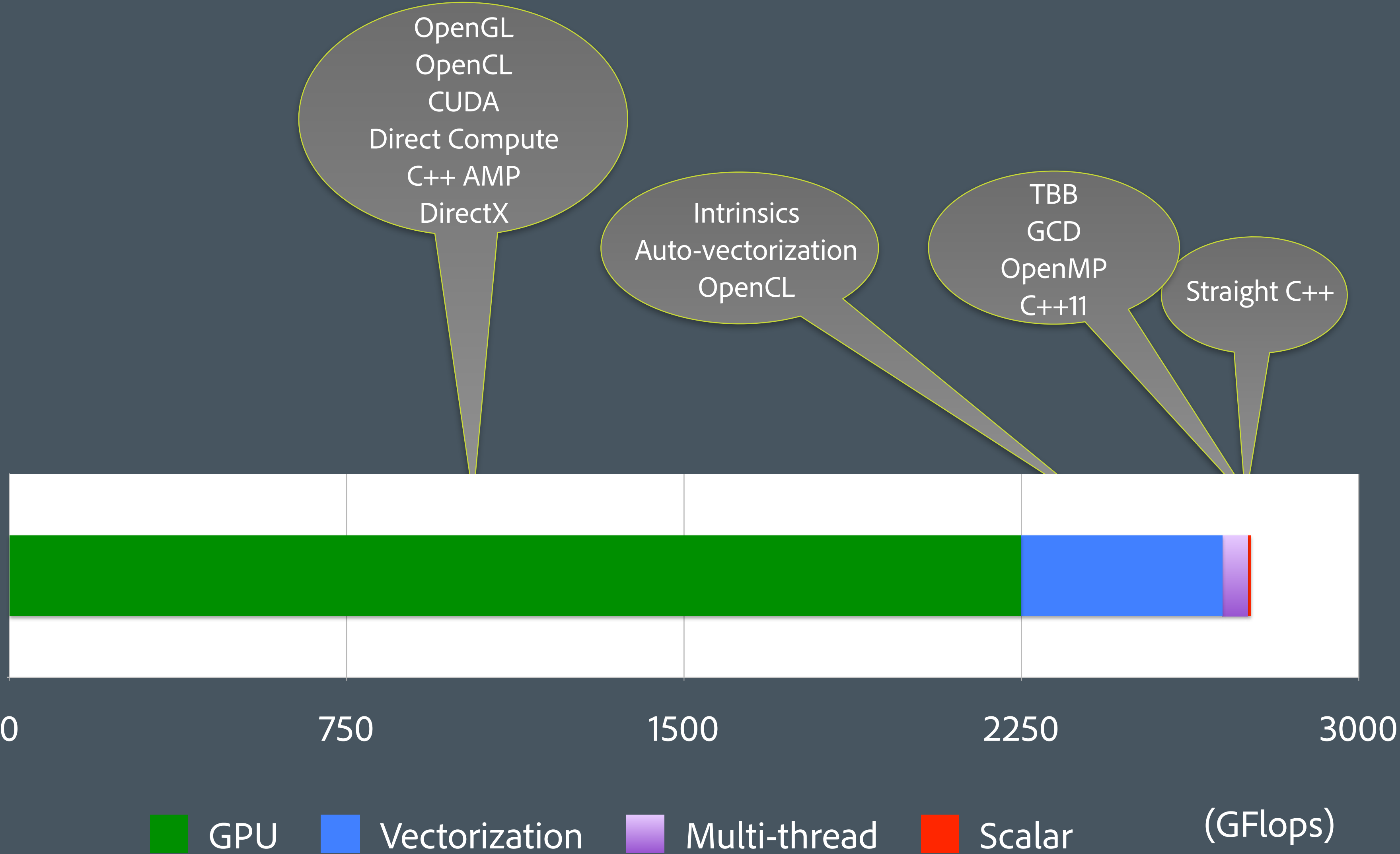




# Desktop Compute Power (8-core 3.5GHz Sandy Bridge + AMD Radeon 6950)



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



# Threads and Tasks

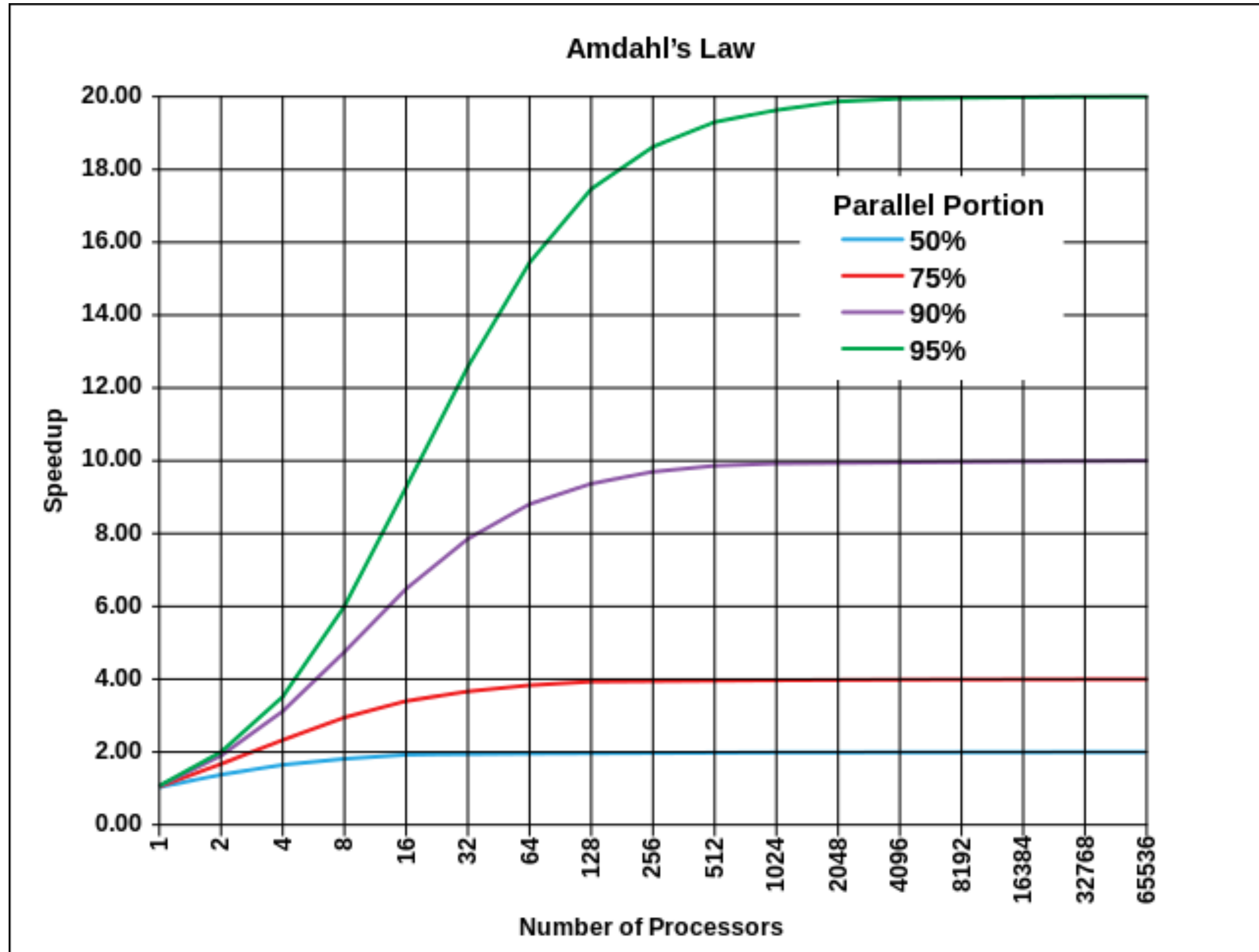
- Thread: Execution environment consisting of a stack and processor state running in parallel to other threads
- Task: A unit of work, often a function, to be executed on a thread
  
- Tasks are scheduled on a thread pool to optimize machine utilization



- C++14 does not have a task system
  - Threads
  - Futures (more on this)

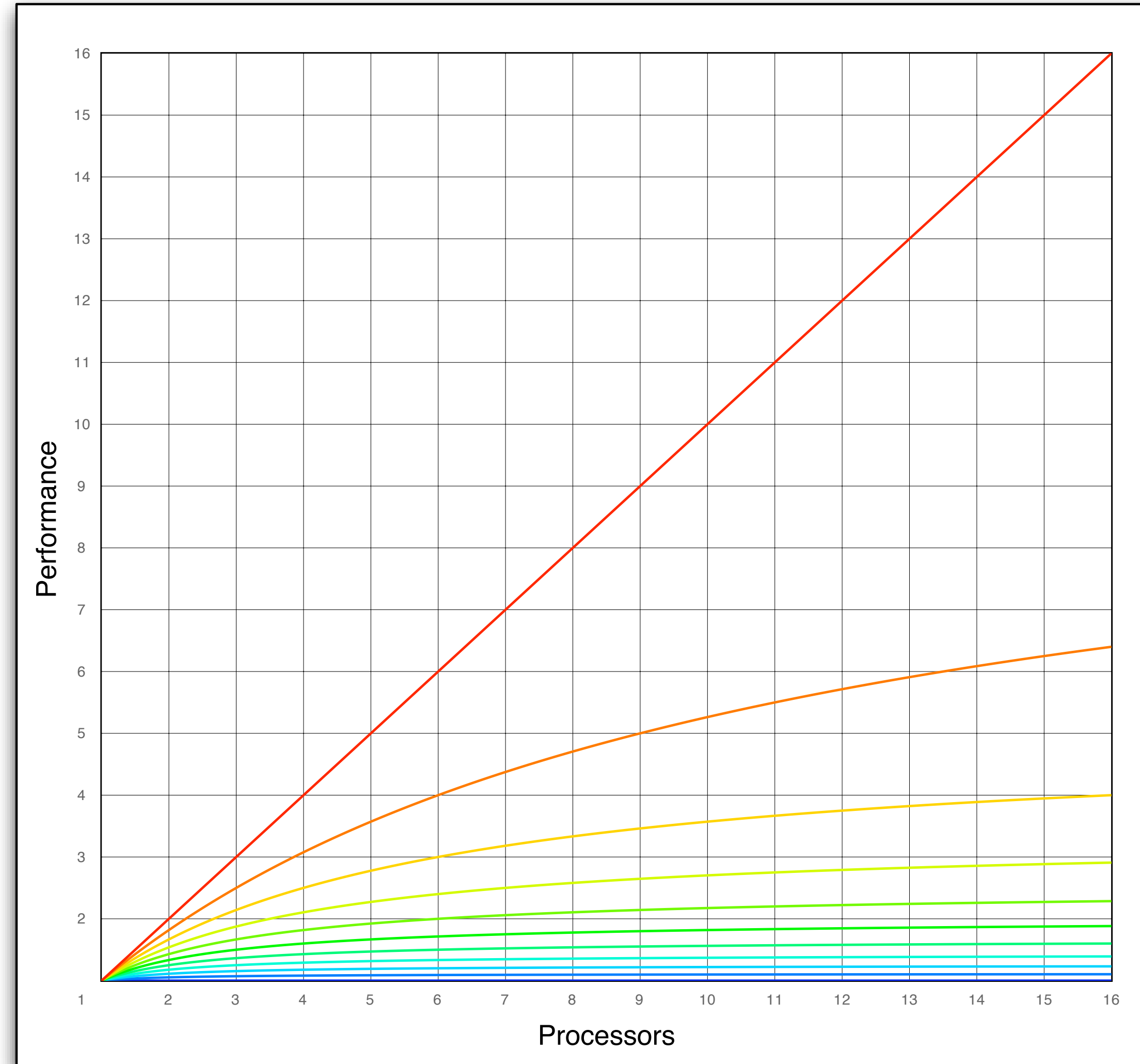
# Amdahl's Law

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



[http://en.wikipedia.org/wiki/Amdahl%27s\\_law](http://en.wikipedia.org/wiki/Amdahl%27s_law)

# Amdahl's Law

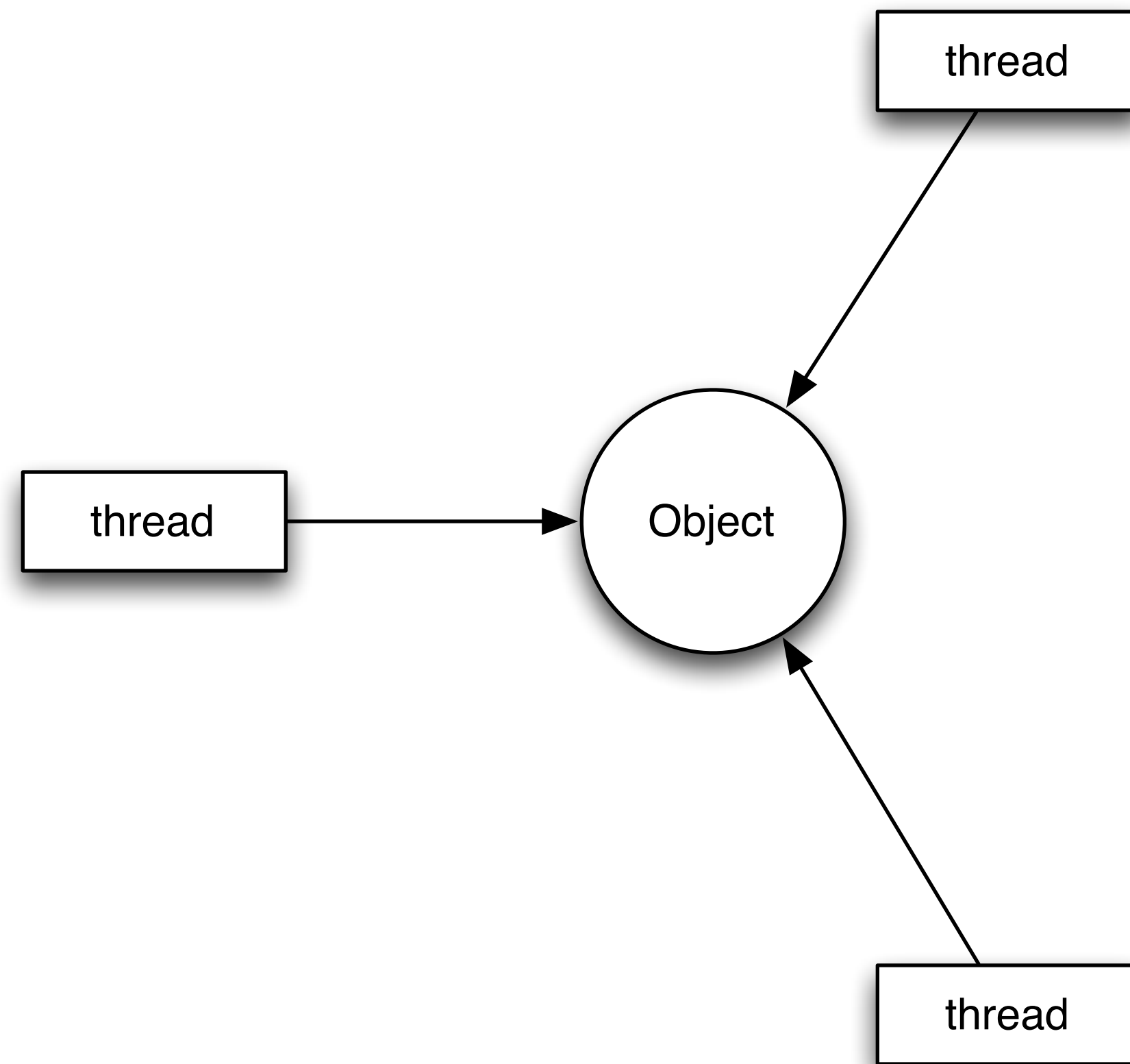




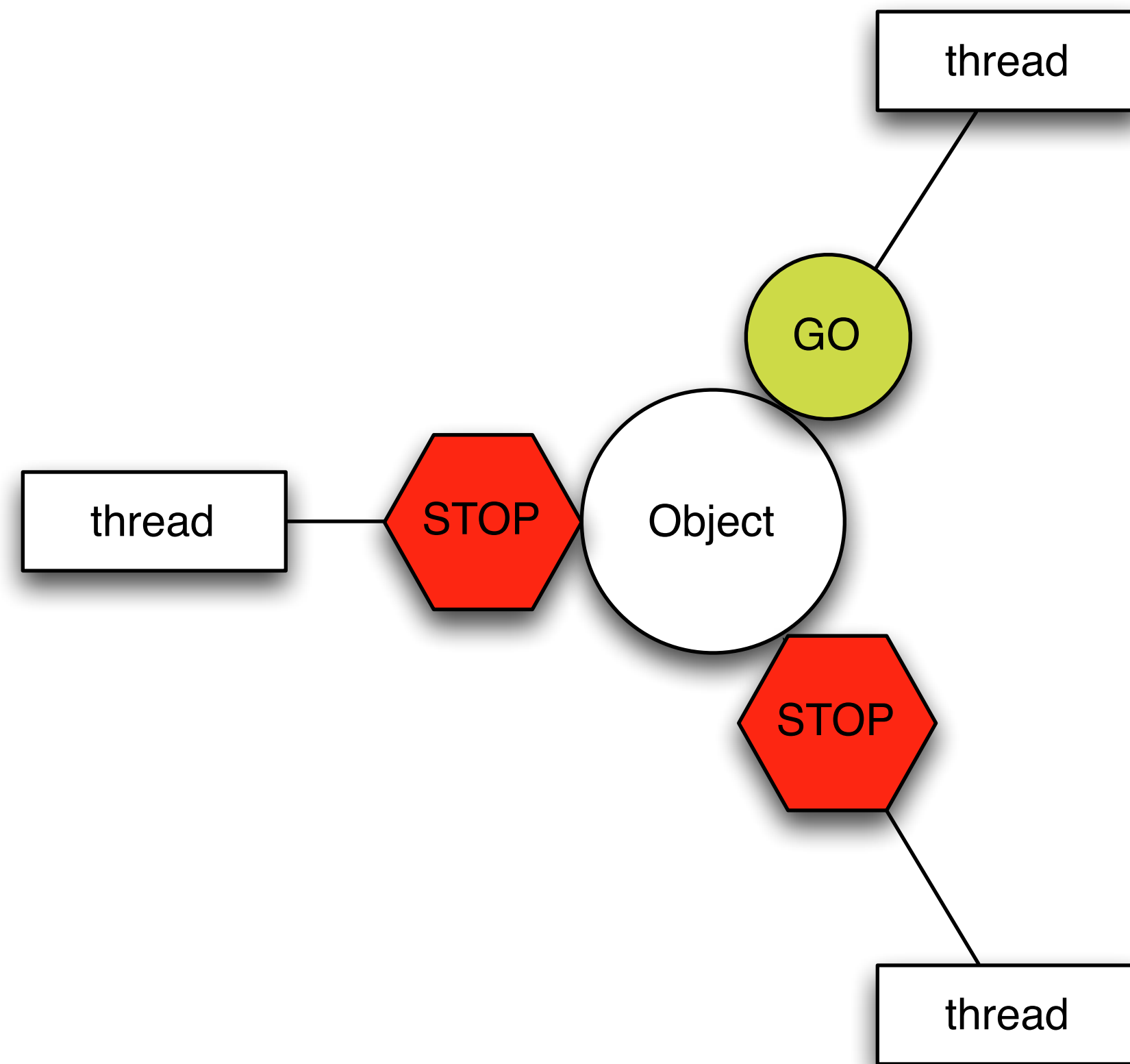
# What Makes It Slow

- Starvation
- Latency
- Overhead
- Wait

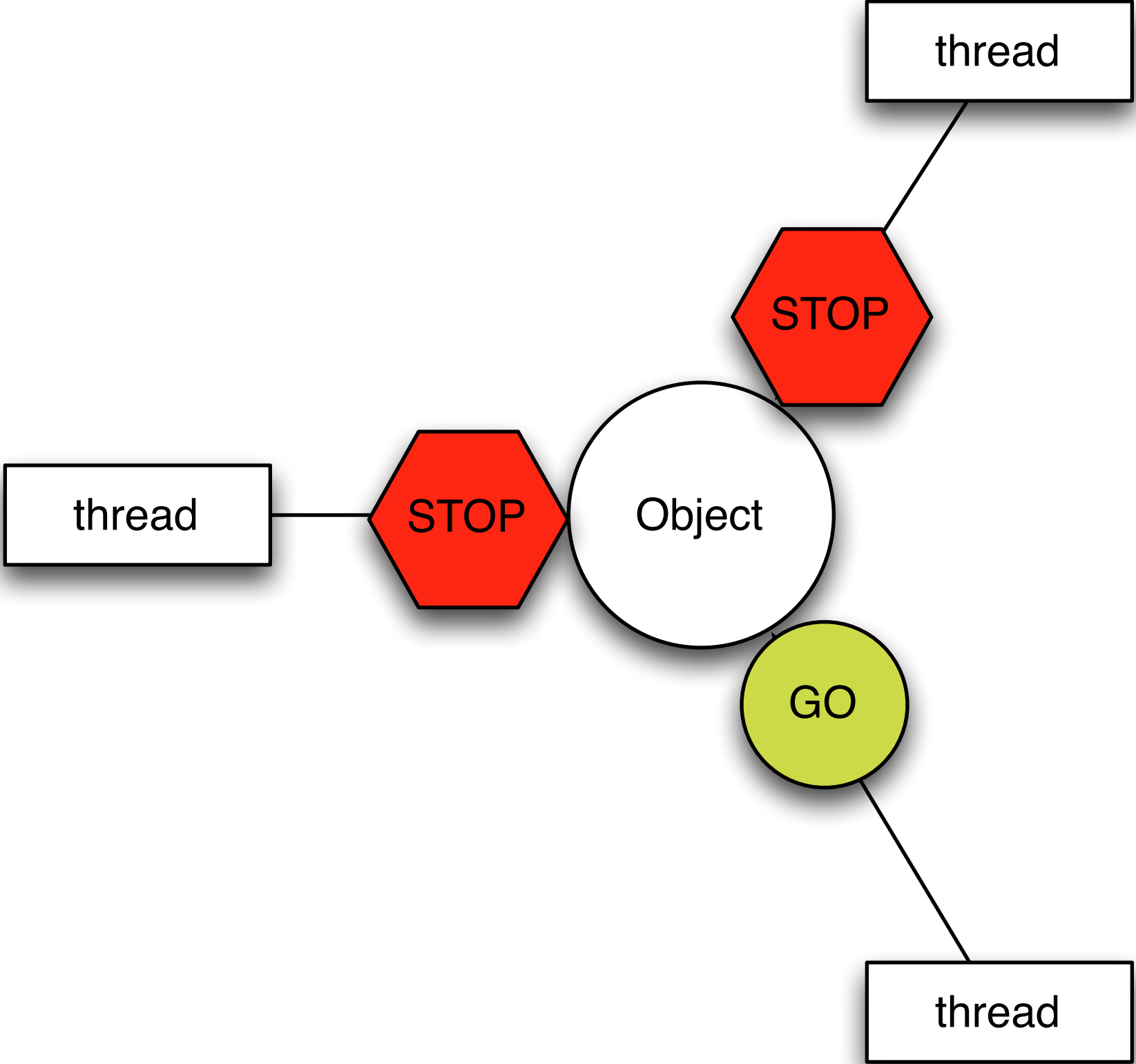
# Why No Raw Synchronization Primitives?



# Why No Raw Synchronization Primitives?

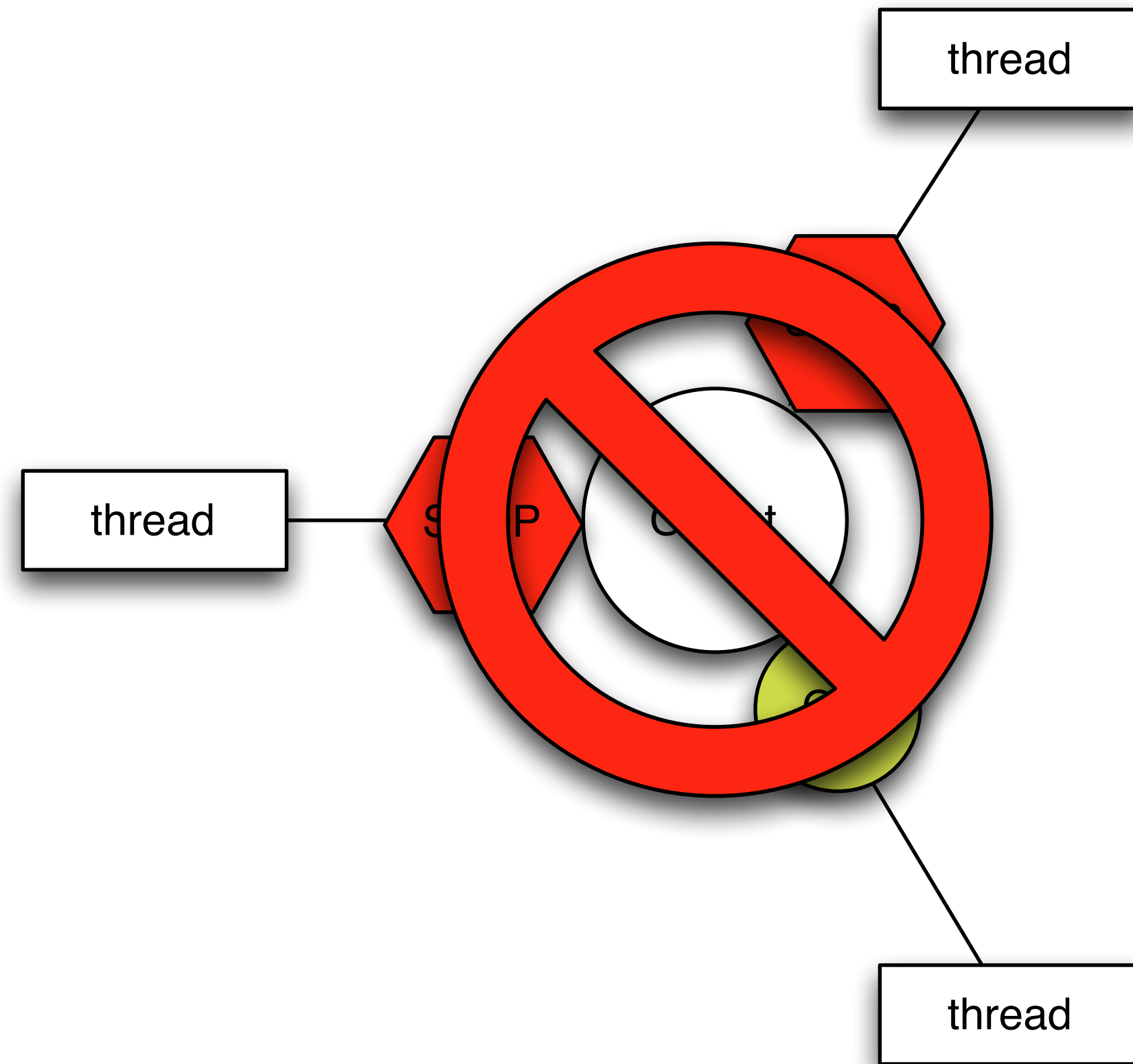


# Why No Raw Synchronization Primitives?





# Why No Raw Synchronization Primitives?



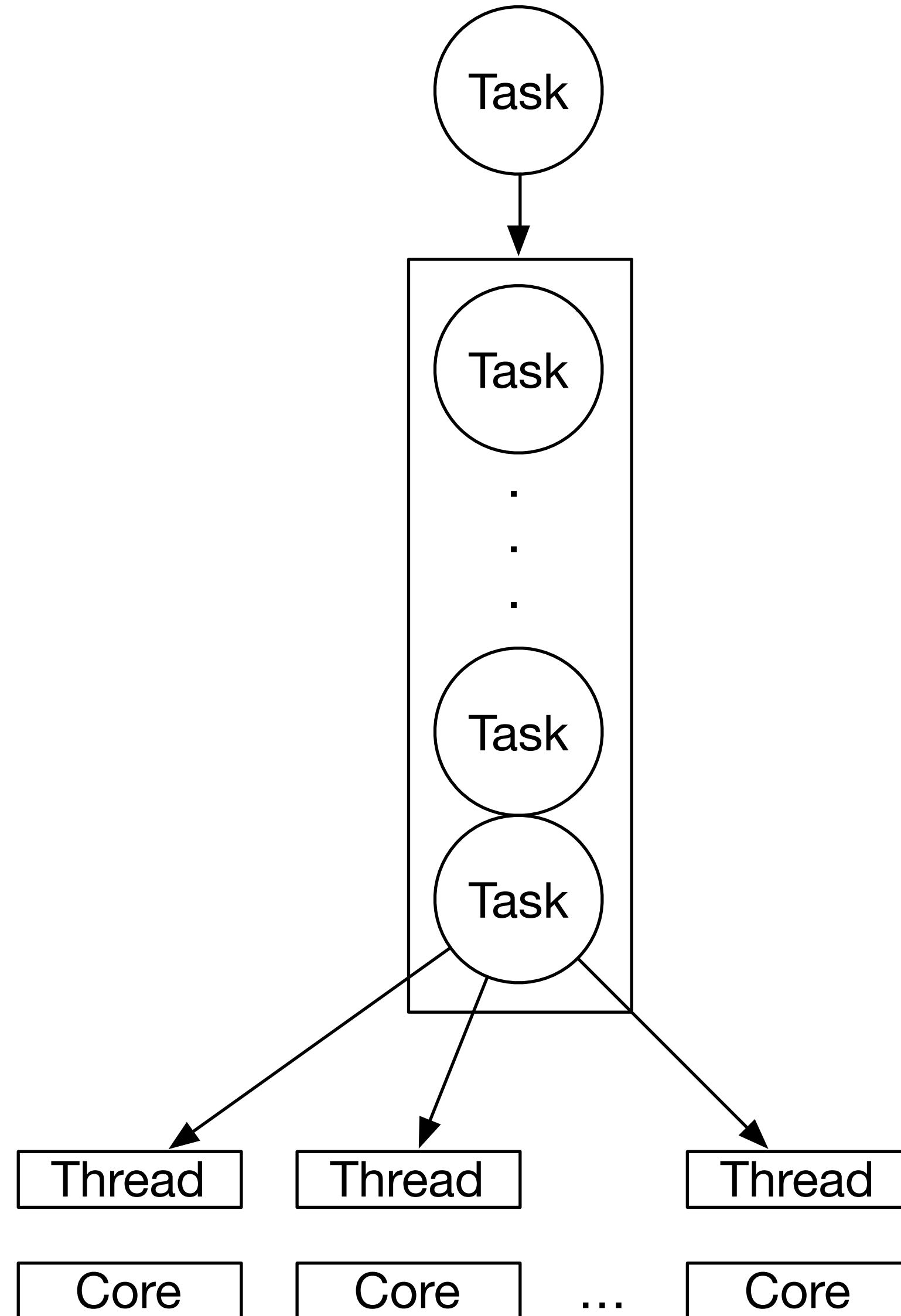




- Portable Reference Implementation in C++14
  - Windows - Window Thread Pool and PPL
  - Apple - Grand Central Dispatch (libdispatch)
    - open source, runs on Linux and Android
  - Intel TBB - many platforms
    - open source
  - HPX - many platforms
    - open source



# 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

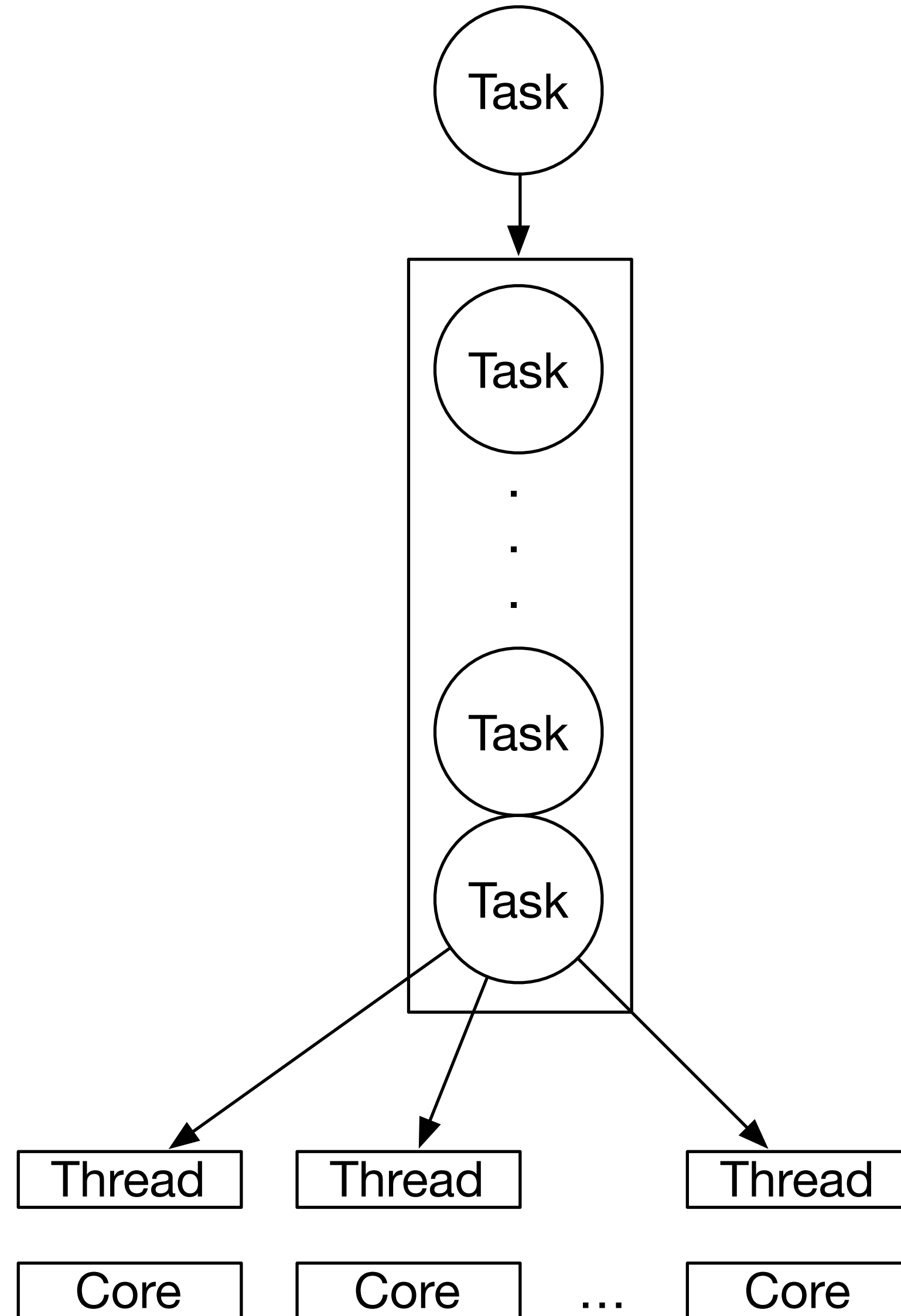
```
class notification_queue {
    deque<function<void()>> _q;
    bool _done{false};
    mutex _mutex;
    condition_variable _ready;

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

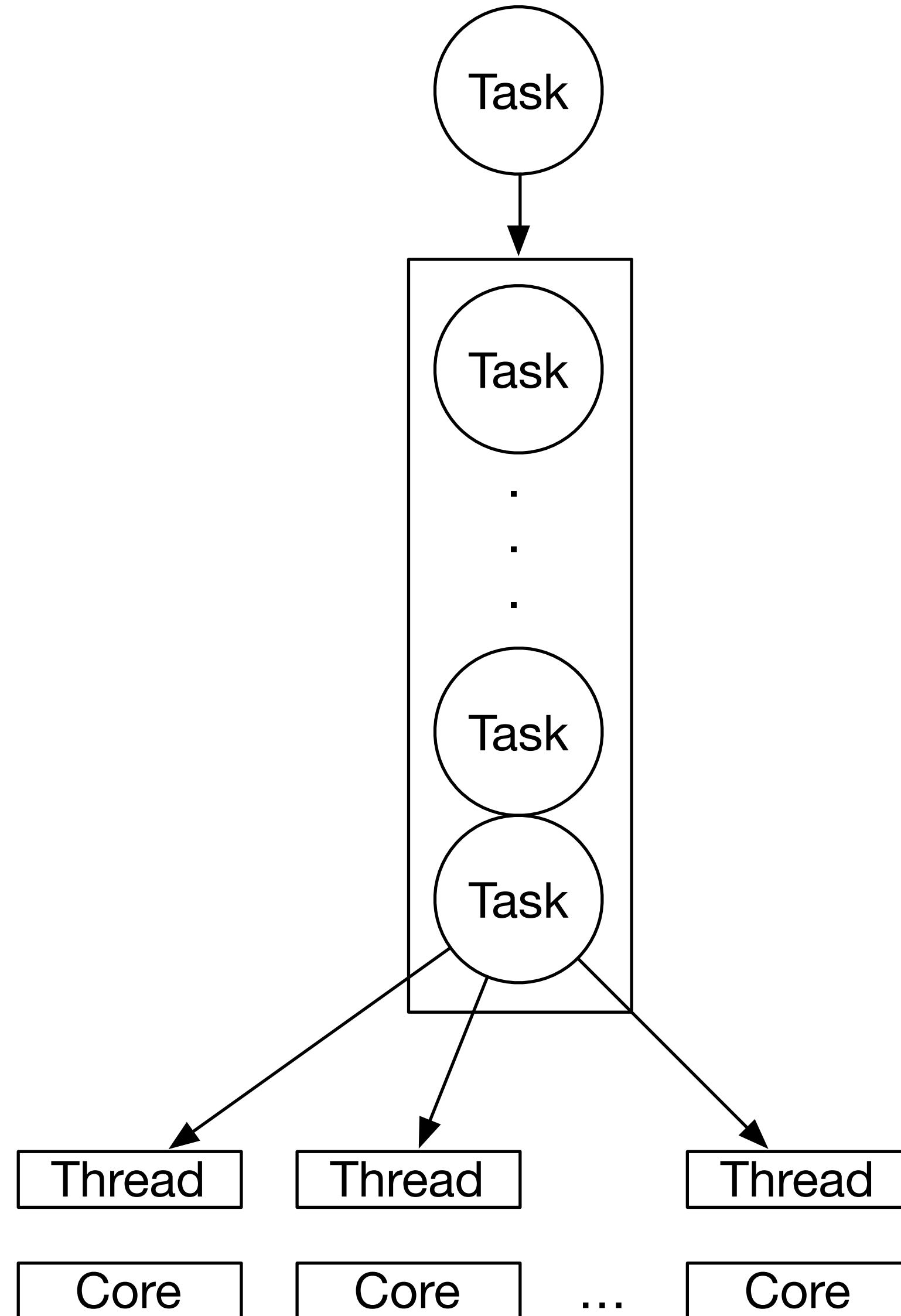
    bool pop(function<void()>& x) {
        lock_t lock{_mutex};
        while (_q.empty() && !_done) _ready.wait(lock);
        if (_q.empty()) return false;
        x = move(_q.front());
        _q.pop_front();
        return true;
    }

    template<typename F>
    void push(F&& f) {
        {
            lock_t lock{_mutex};
            q.emplace back(forward<F>(f));
        }
    }
};
```

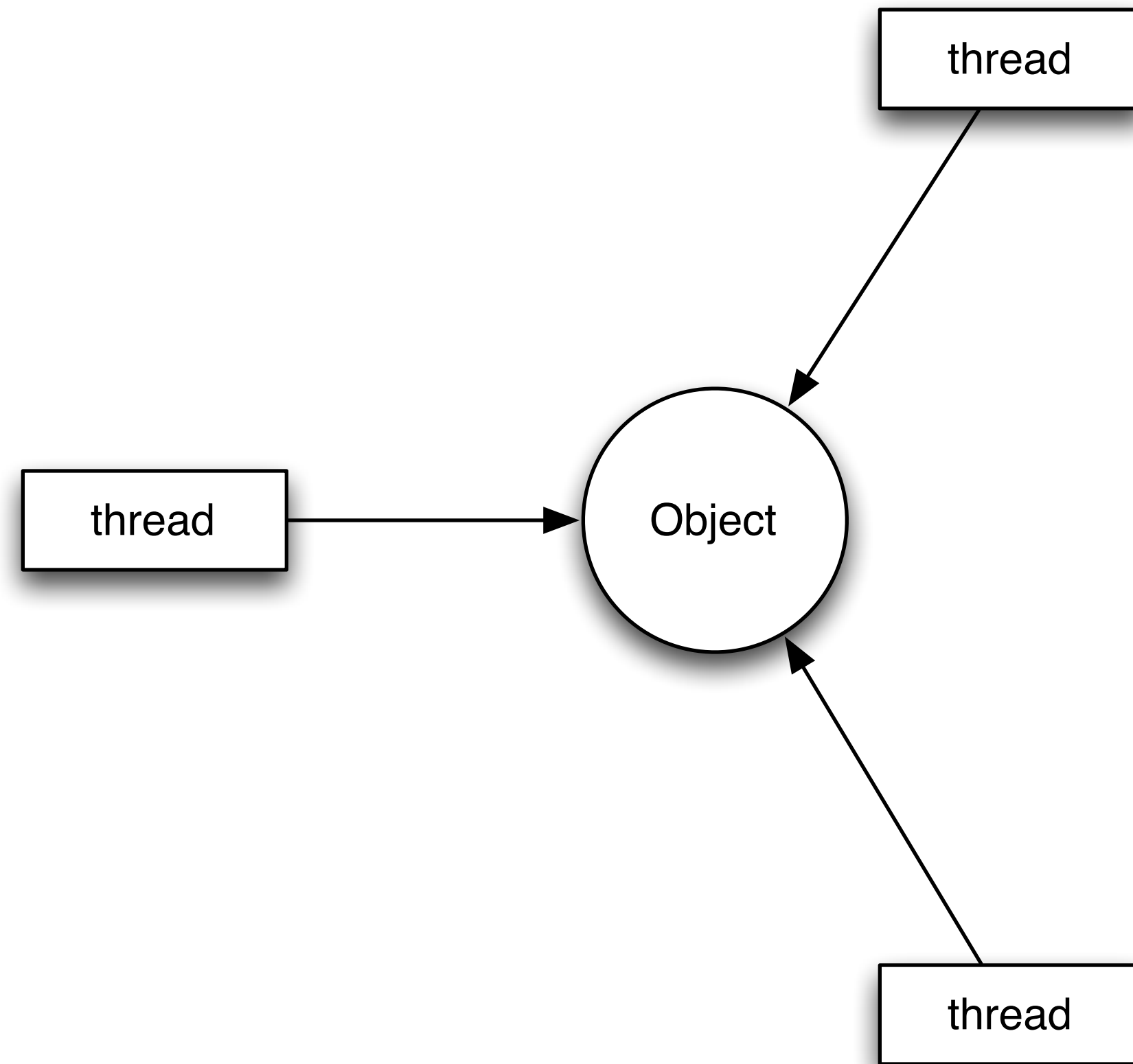
# Building a Task System



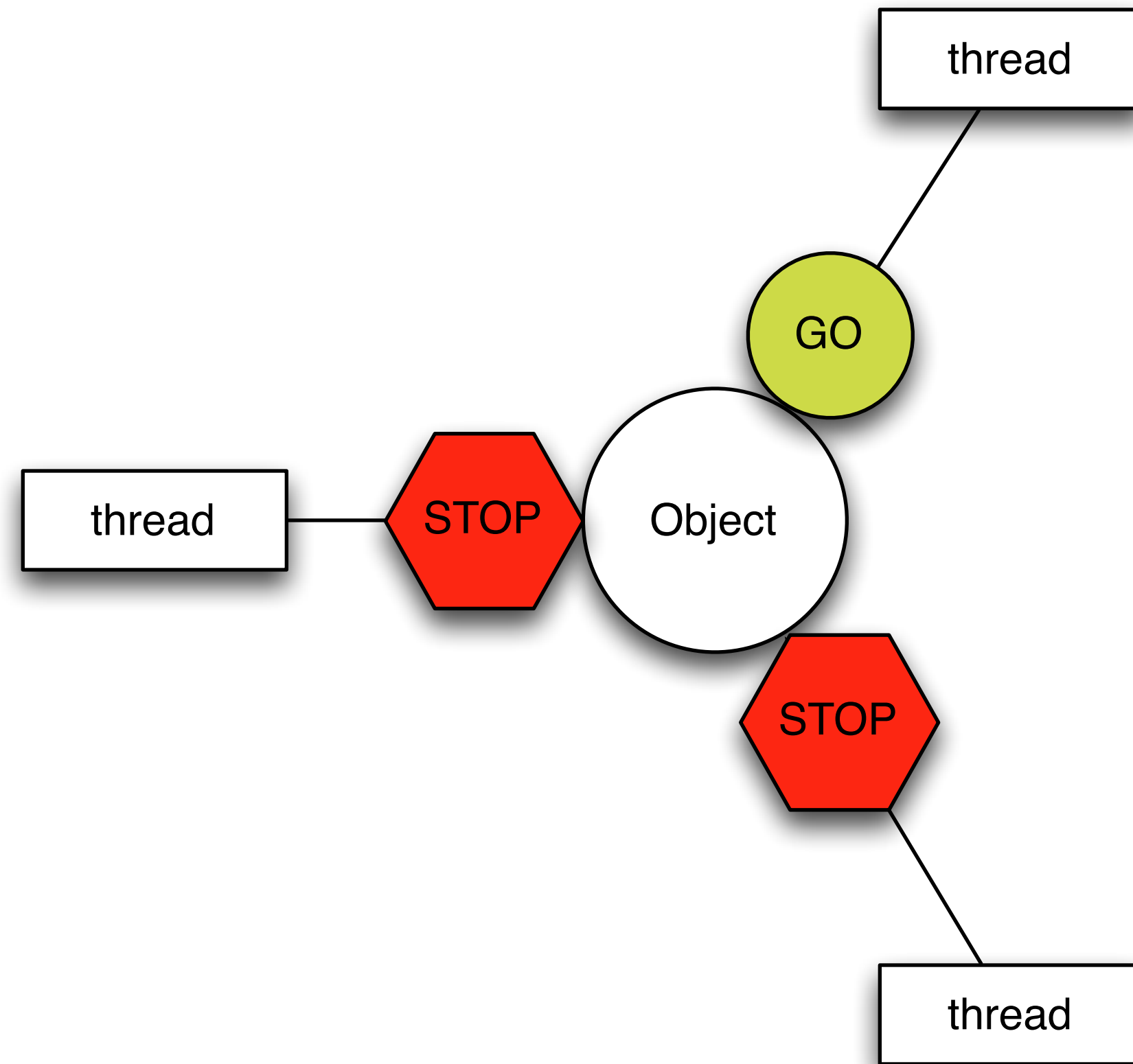
# Building a Task System



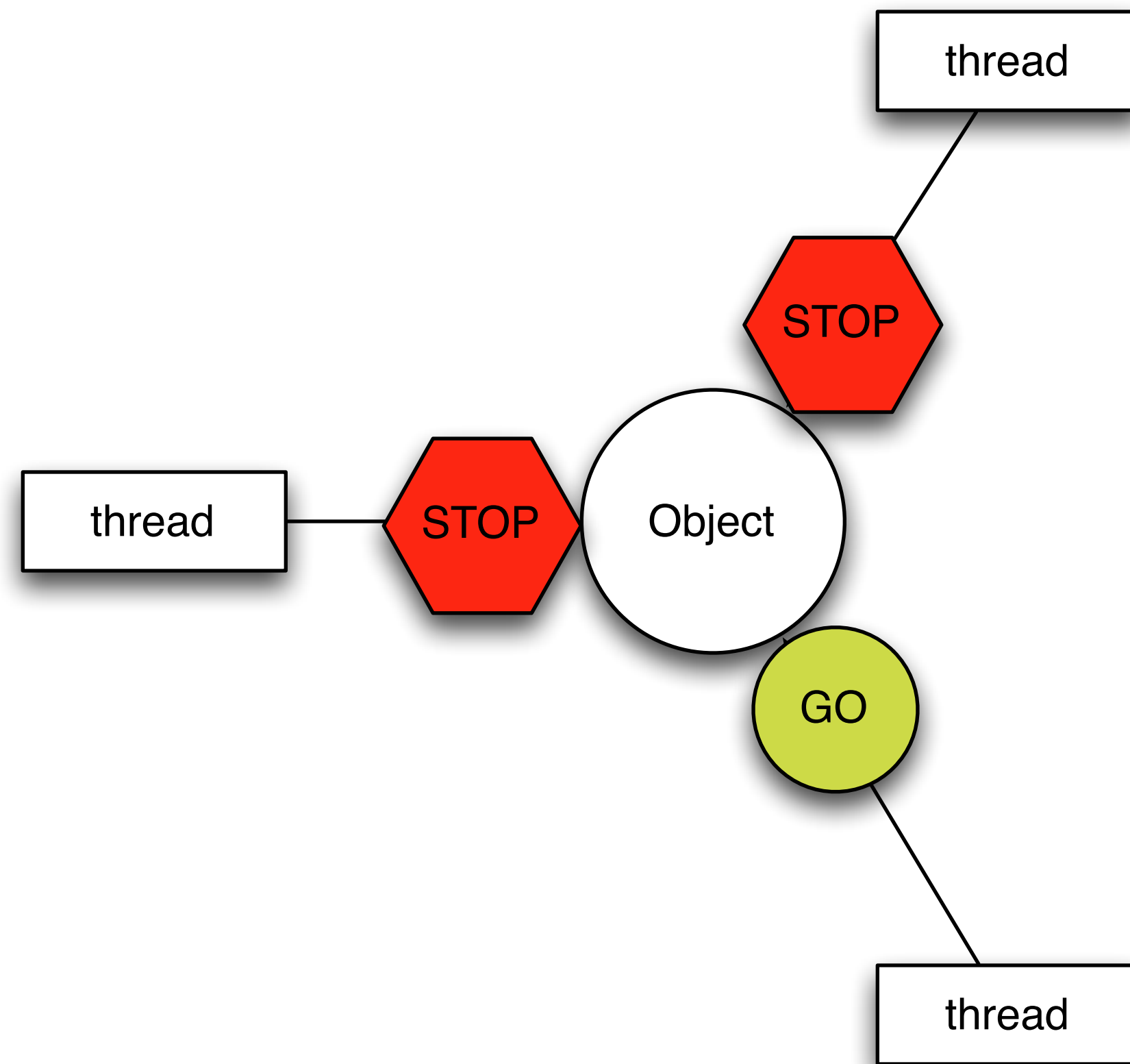
# Why No Raw Synchronization Primitives?



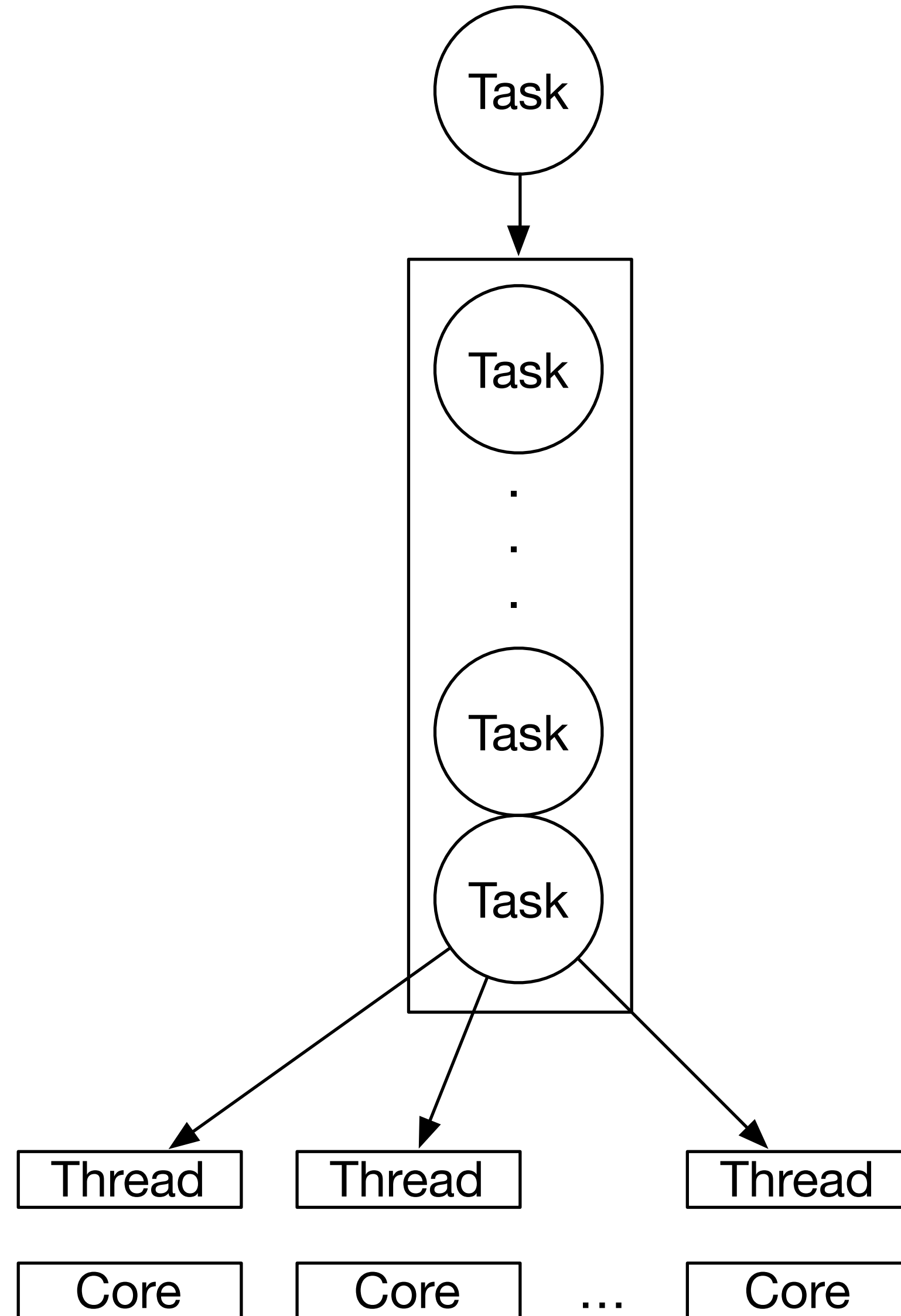
# Why No Raw Synchronization Primitives?



# Why No Raw Synchronization Primitives?

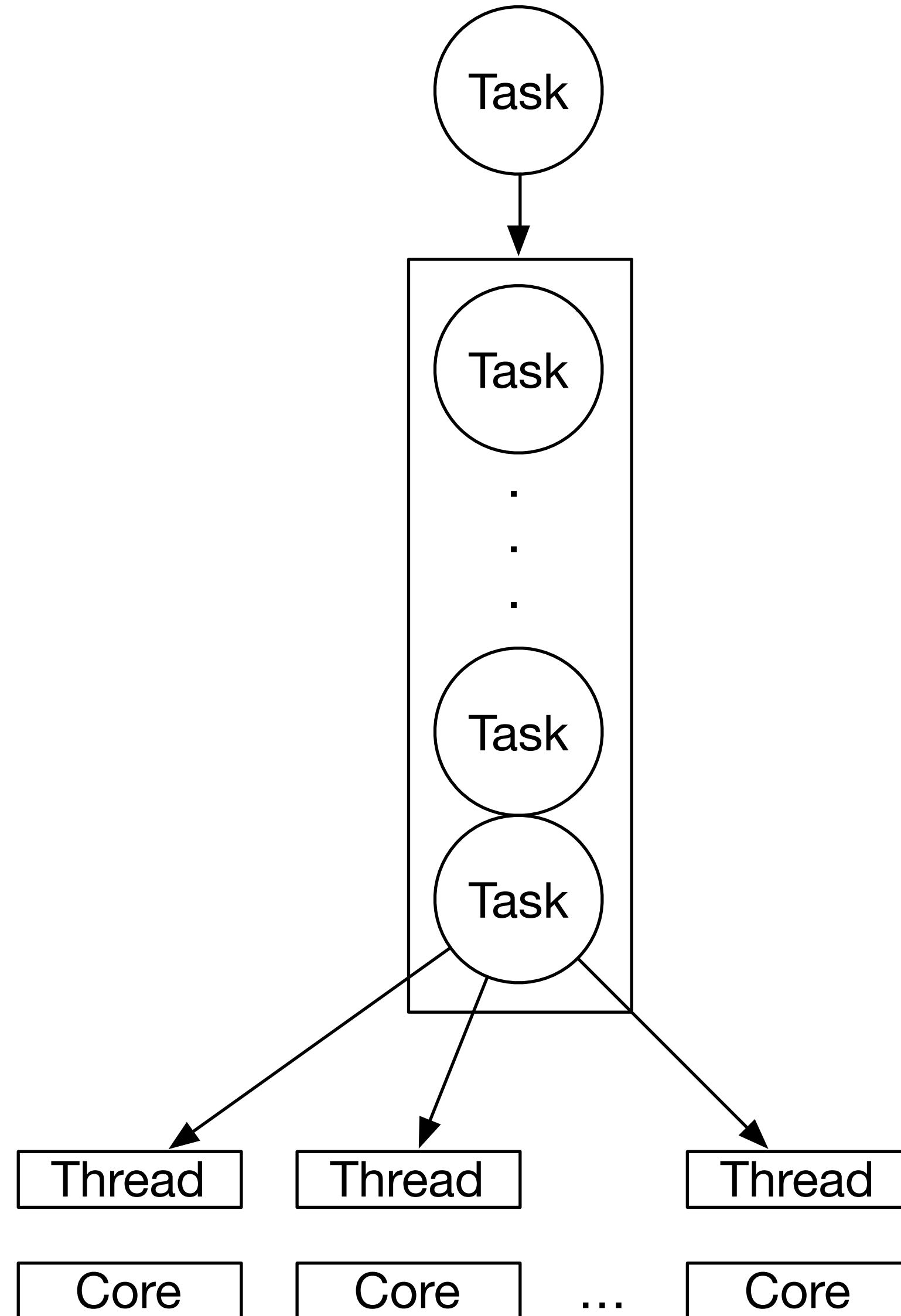


# Building a Task System

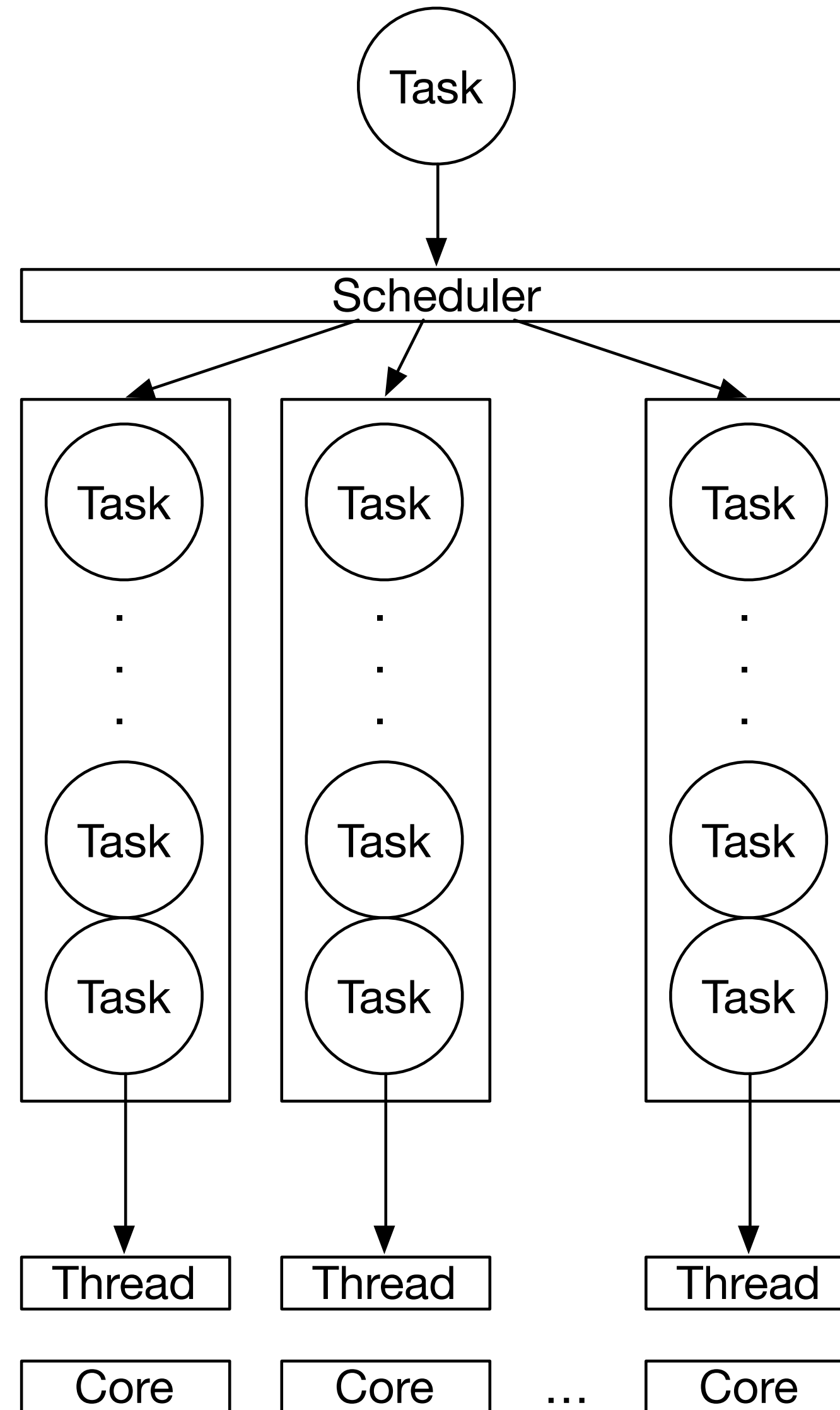




# Building a Task System



# 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();
        }
    }

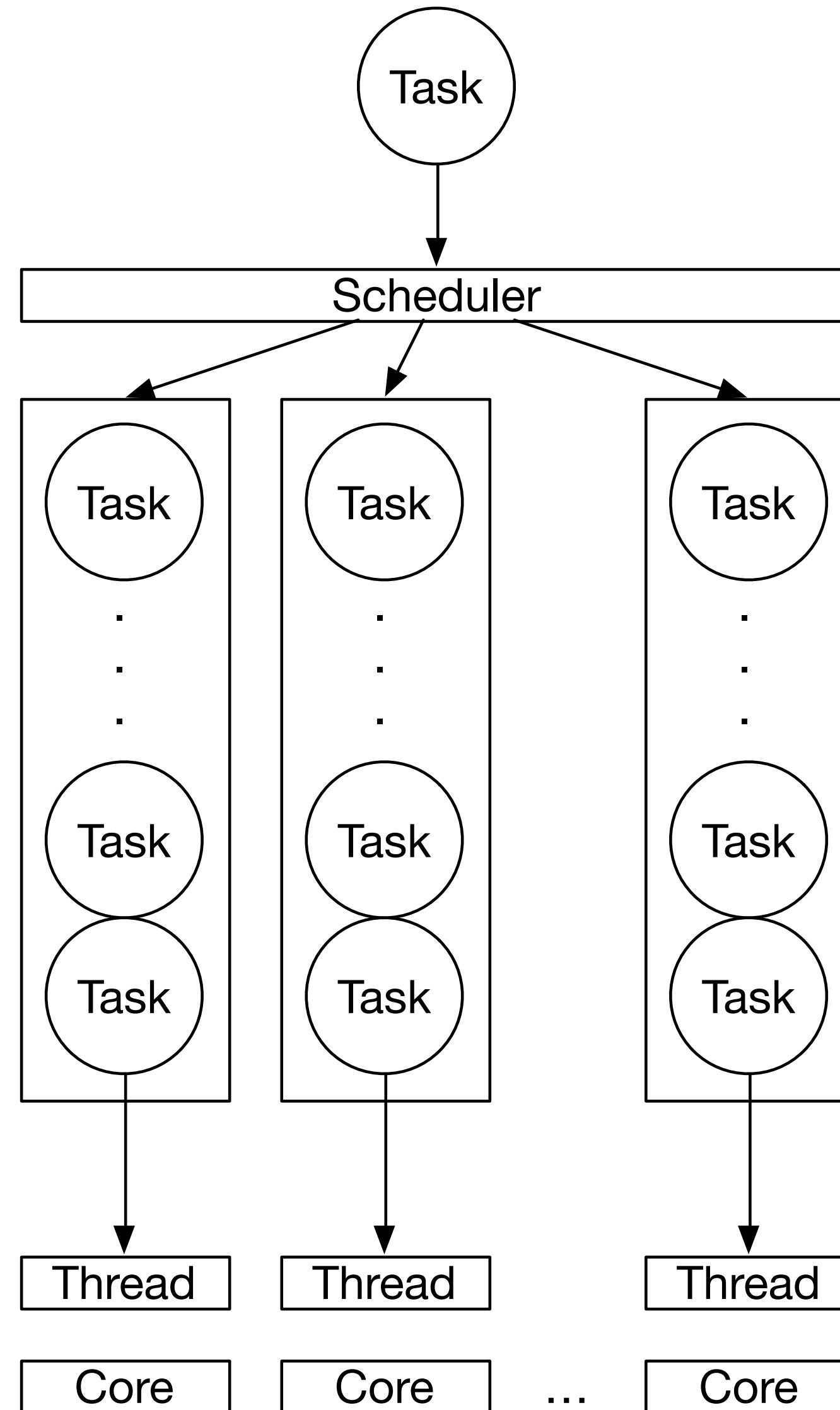
public:
    task_system() { ... }

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

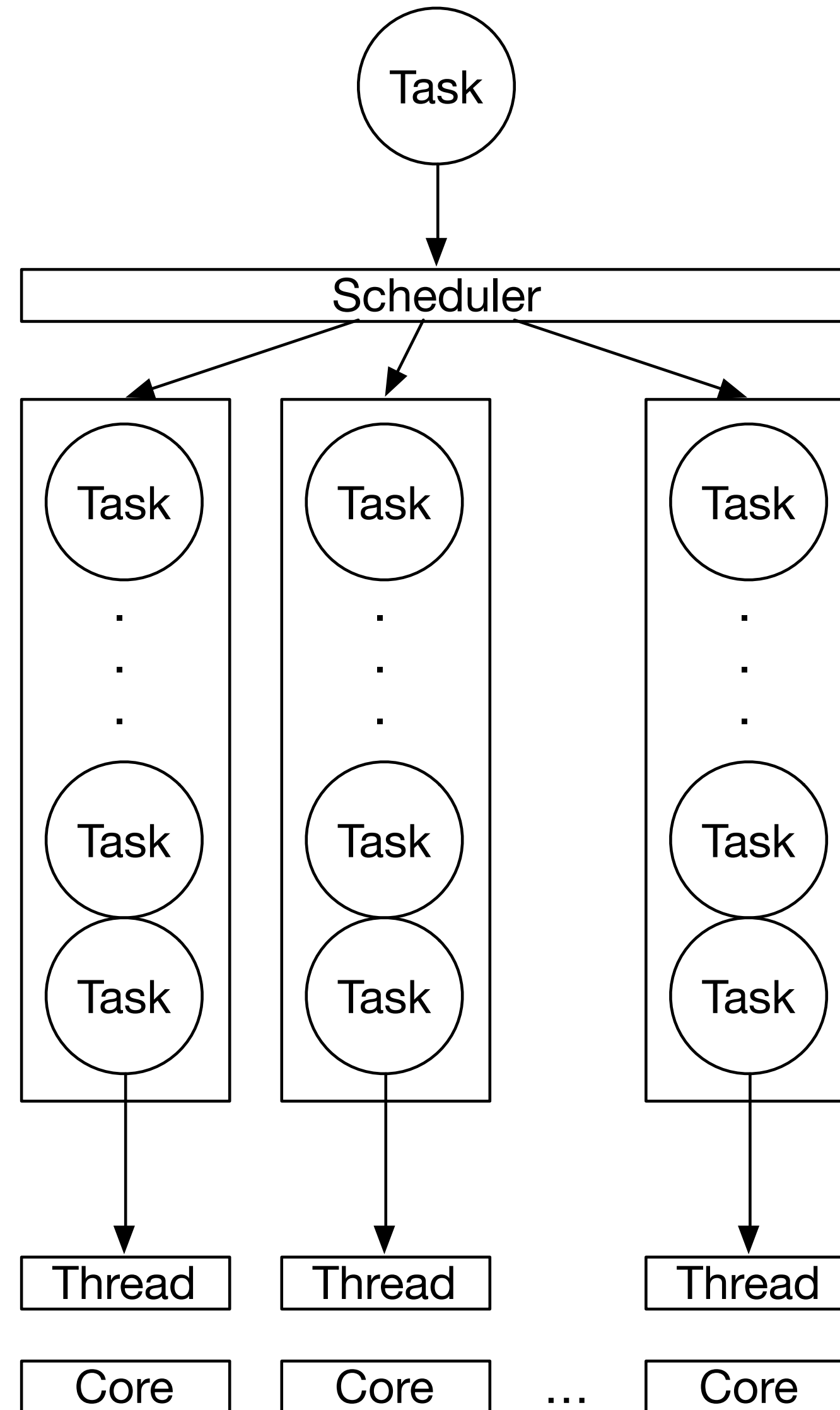
    template <typename F>
    void async_(F&& f) {
        auto i = _index++;
        _q[i % _count].push(forward<F>(f));
    }
};
```



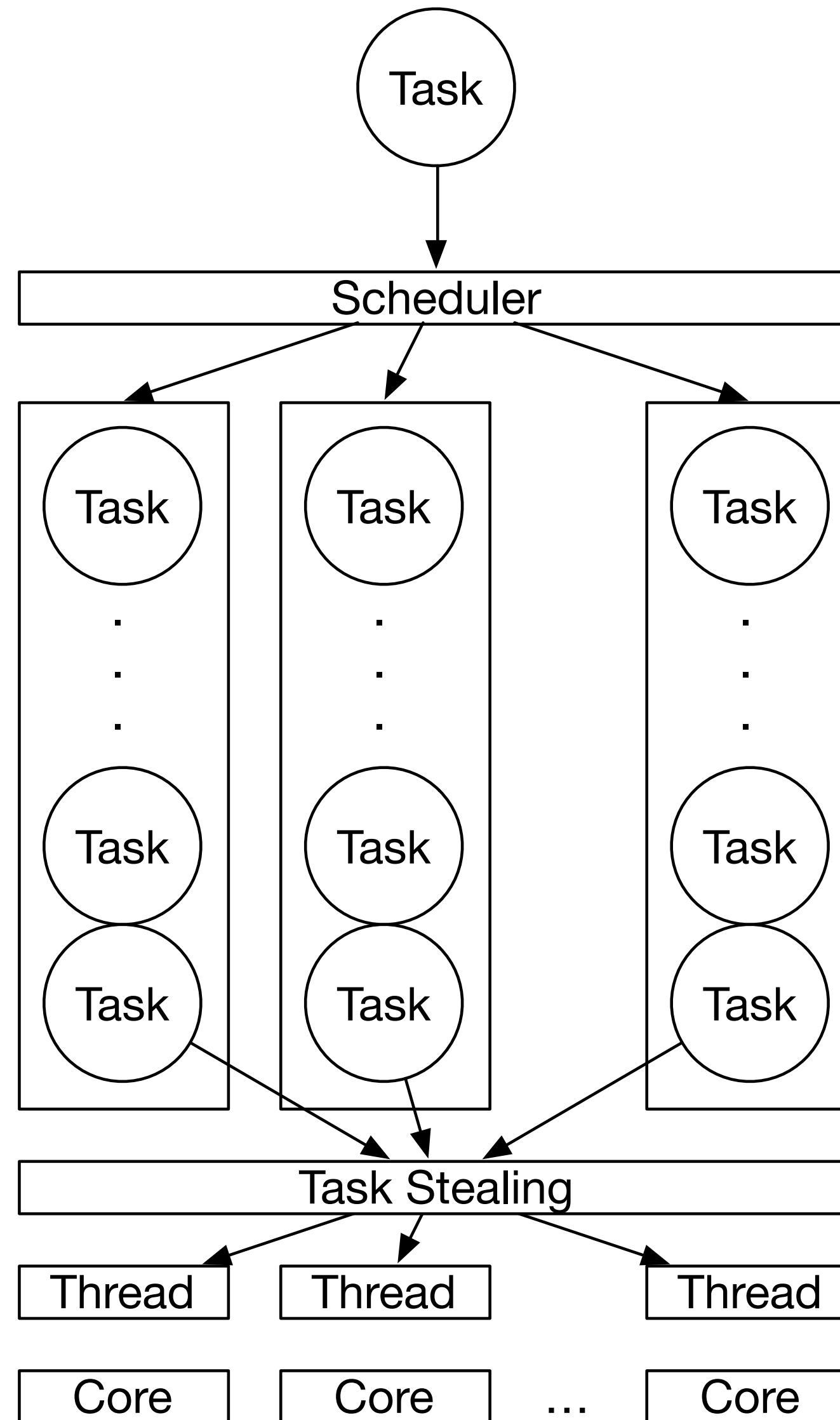
# Building a Task System



# 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};
        }
    }
};
```

# 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; ++n) {  
        if (_q[(i + n) % _count].try_push(forward<F>(f))) return;  
    }  
  
    _q[i % _count].push(forward<F>(f));  
}  
};
```



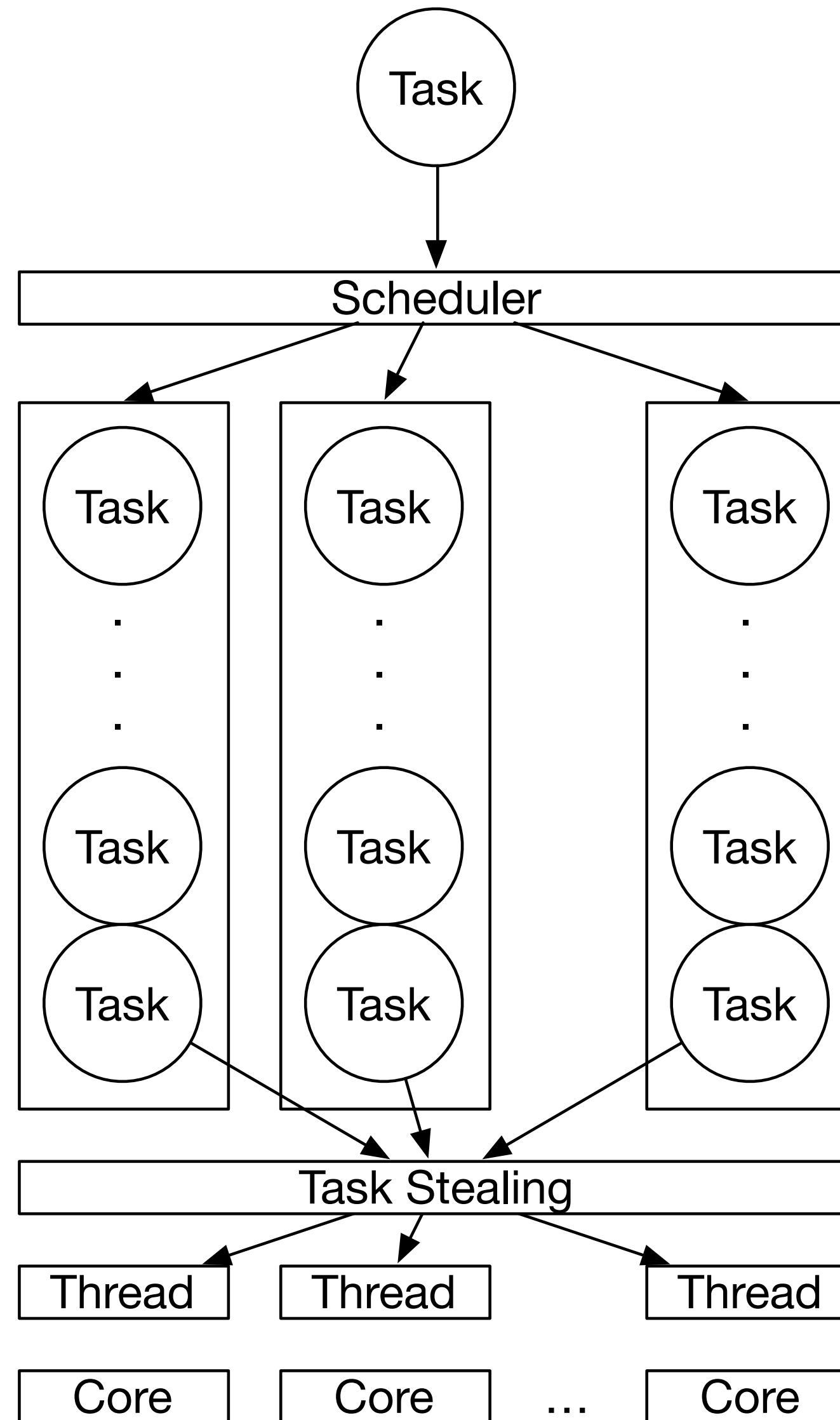
# Building a Task System

```
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; ++n) {  
        if (_q[(i + n) % _count].try_push(forward<F>(f))) return;  
    }  
  
    _q[i % _count].push(forward<F>(f));  
}  
};
```

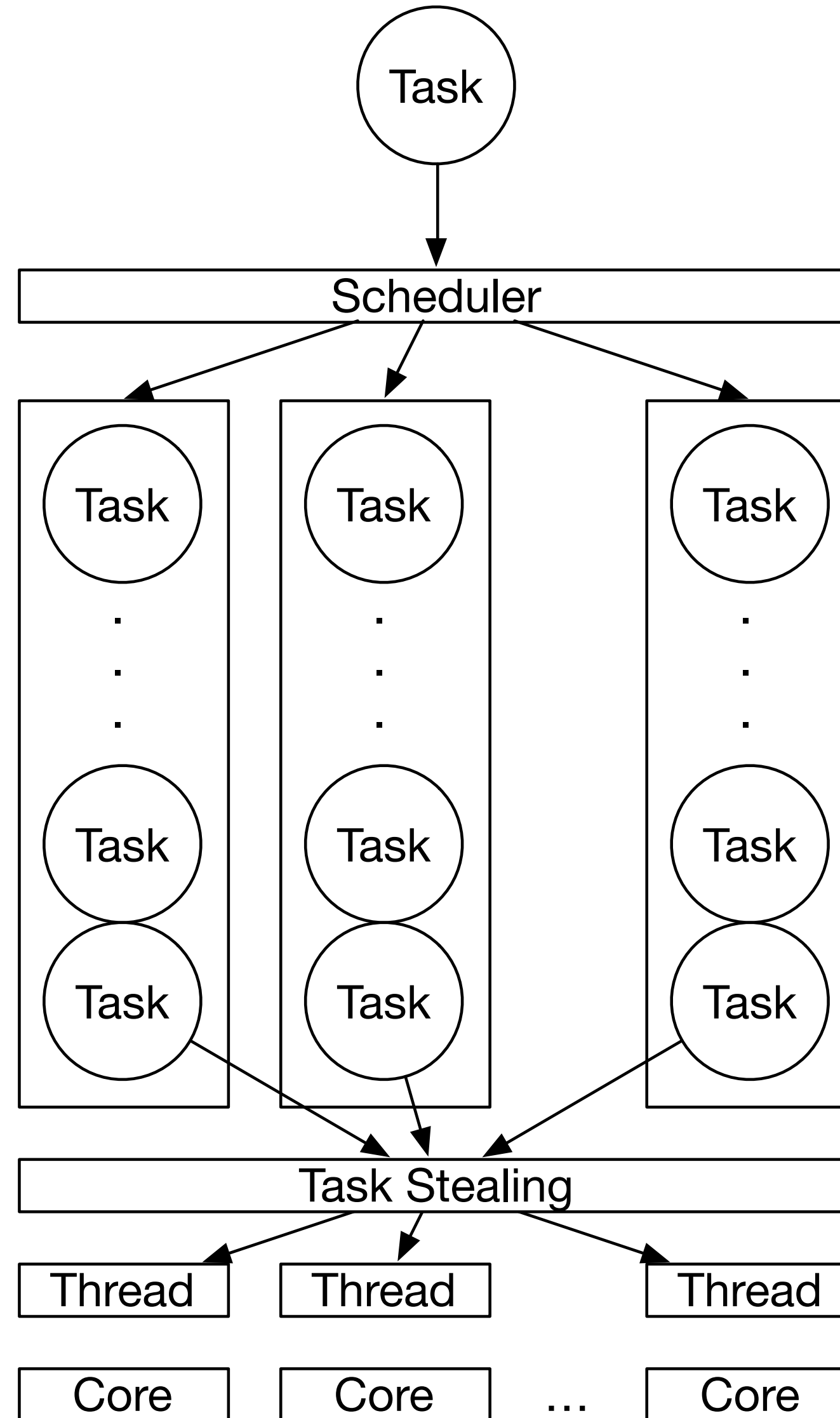
# Building a Task System

```
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; ++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



- Within a few percentage points of Apple's GCD (libdispatch) under load
  - Can be improved by spinning more on `try_pop` in run

# Futures

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

```
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];
}
```

# Futures

19532821287077577316320149475962563324435429965918733969534051945716252578870156947666419876341501461288795  
24335220236084625510912019560233744015438115196636156919962125642894303370113827800638002767411527927466669  
86557837931882283206127149758323033485489348957259923072291290192820926433162752173086146001791258204269965  
99360209593392020051848620284024473431398113674187202038684801753185386211128781082406177413832935545616876  
06454065125954718029126547942894036981659206361019359291352135410376799082940320155702716115395031975973247  
78216295763162965335669477766328506234524559346064757502593581344345781676764625878859011372729907372947851  
14480895724561915035070255895291168685500088020132334587472177947814475467920160901706425856293597475465327  
57575740077432034913428785189795354304734560307765078938767286539166799232817449361991523768149557632085371  
04785970618843873153058239562756087906310781900497516959470973671389174570455520213512335079440336071203050  
41446852210415650373210679322756258647511914611417360349681217380234224786080292021093192496490409832397066  
83247054441763512526732455275419501683845206023007394959854279298297831204382115757645787692495583351402522  
15272066244180900325938075362849179668095297118507191379833678873770459913639333955814212036990261617972113  
25091840023055327607104316478190974300434647793363287601469996128023925829471557316688943339455429292871877  
48774789204296166356536610796023919702109728472966709427334586344798048633944635211654971507261342768205479  
32093175079888010130416027982506354182344034558742236701282666356934611294613123128389060036547327660245693  
15151850018328483150645480029978935985161237074046158229354440701748339514575869547491750264542126364262224  
72060048855462589961190475892101224280542898621594646662478564373572217775549876087685912030118551635668902  
01034463998397732663888903650784161807091545252992759735213957415477729146008794314339156060445825107823511  
66271892637923313014643880597879468444879060576786297460989627426663569682474293386740207436559426057944790  
71193052258931590719386545525880429139747140181849169733838138446154843063123649290835584278078456131936457  
55911722136946338180311600307896211668652895953778436464402382516362449718197385444149563131714002850338928  
22274134603018094224837216321854717270452813824078425638747365249141118080783866506339945376239206700513391  
87333107136069698189628284763245423299306272870457991293245741167533902274499963096566680922262516468582544  
55785134982414412726124015815753818098466667145006988839178551800894370189025721992485208742915560261917752  
28124660628996787166529678487268484905041328497297712688011639978376434280202452251550102240354169885185375  
01584673881194047619720619603126534496759917893244478170702904446589571950228809157793897642423751814020998  
99958161231477902295781100168670186738619861797138139854666281969548553740707356228616165539428076418408092  
12047932816683005984504787929406356318097479755152035094682765918741610907637506902765294367561539803261388  
00104485041004520227541880045735620705421800662063441346306055080001375010053760250440113617210176881147264

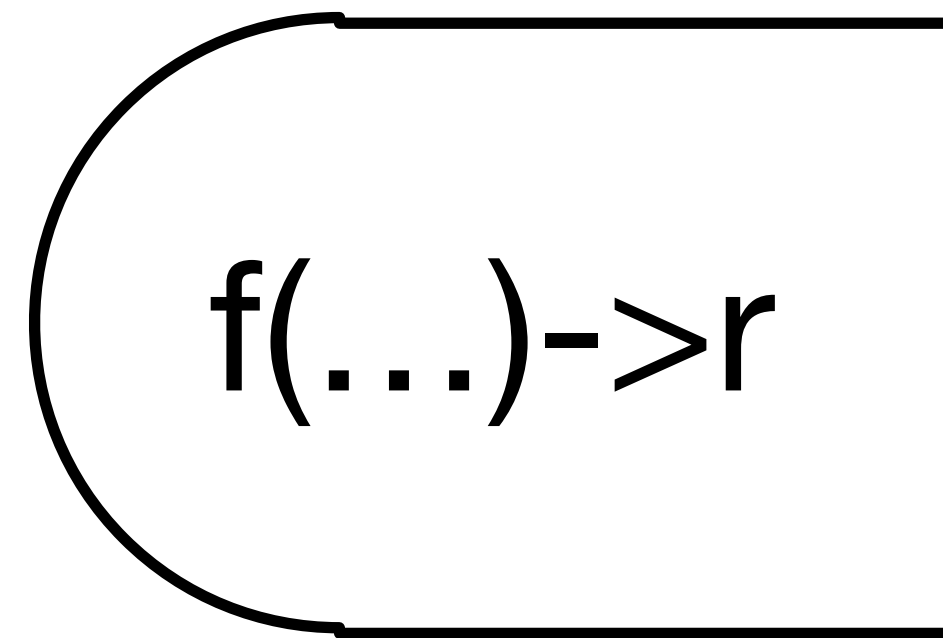


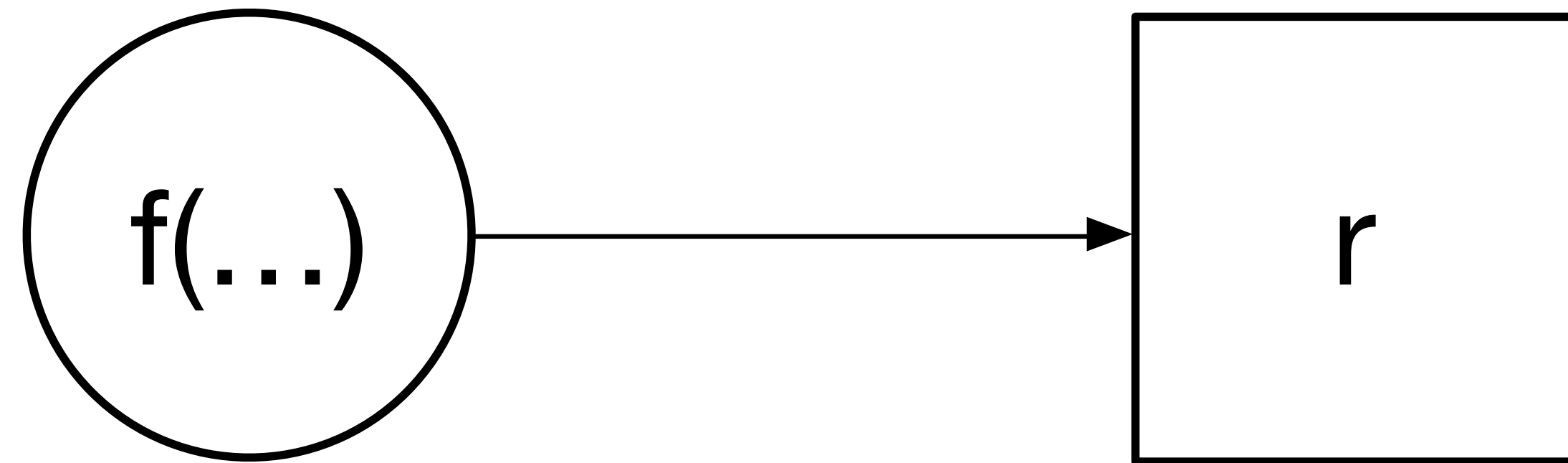
**Futuros**  
15790155892833100345673846243104676900000936756893803676769777642059716492347060997973282994459039755683869  
10568541105888505197986232161807165960864316652383369579251545877324797429523572491518310013505994095431367  
23454418539676396422570487868443336735568511535850565172490141772333018072390350689838662532338266203548476  
87722321662223383305226882245421258277211223435986491973881404168406609216954760818955479619408040043497601  
35646408461148077885537891122888139618703907906033147416881433658136276942006644505679690480702792206520855  
12245086839375655196861305232092138041808273198852928058246964575561801618520046644949262341864859342928965  
21378574554544426221453176445385228867960454072522804961741905198550911362542849130027243353553345377968558  
49780195976636516290598457219043489821358221206856924121139313137132134865741440892670003665555632446499775  
56853514681289887391700907057970839124191923062570547772748610990924519168225326823578140721238189631411471  
29610287340041050015549547086272721534936510345705849389706515725684266079756708385889612130516276472992631  
59674474594901199950849178952149715987731953191759591623424021718579696778102054496598766846143959650647332  
21985323521378108187030642875506951890343587181633604126397675020909133548480151135951824112432636080497447  
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 allow minimal code transformations to express dependencies

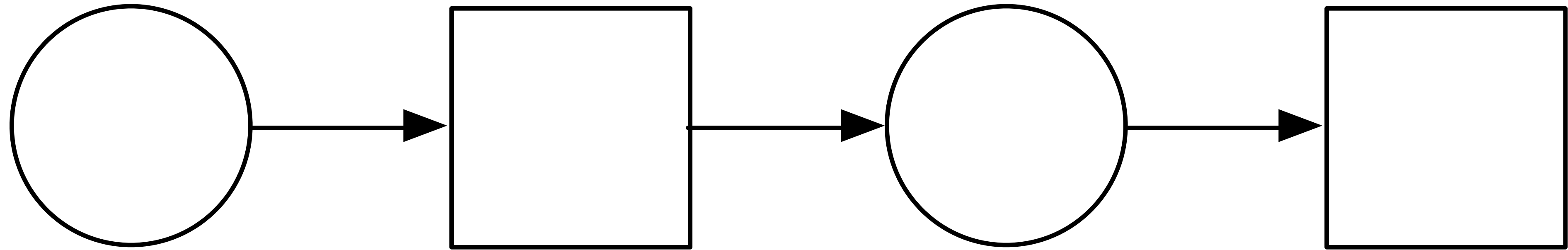


## Futures: What year is this?

- C++14 futures have:
  - Exception Marshaling
  - Sever Notification (broken promise)

## Futures: What year is this?

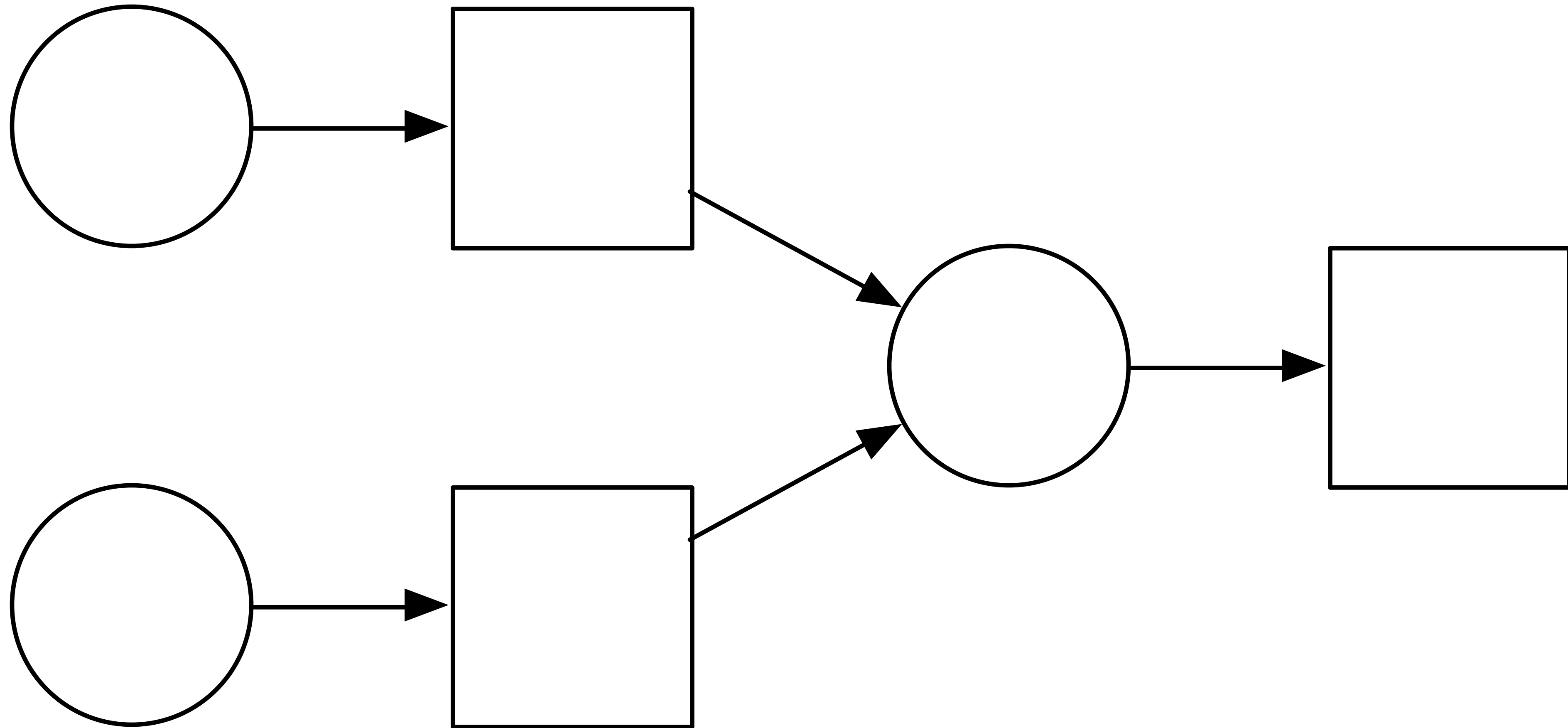
- C++14 futures lack:
  - Continuations - `.then()`
  - Joins - `when_all()`
  - Cancellation
  - Progress Monitoring (Except Ready)
  - ...
- And C++14 futures don't compose (easily) to add these features



- Blocking on `std::future.get()` has two problems
  - One thread resource is consumed, increasing contention
  - 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 them
    - Boost futures have them now

# Futures: Continuations

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



# 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

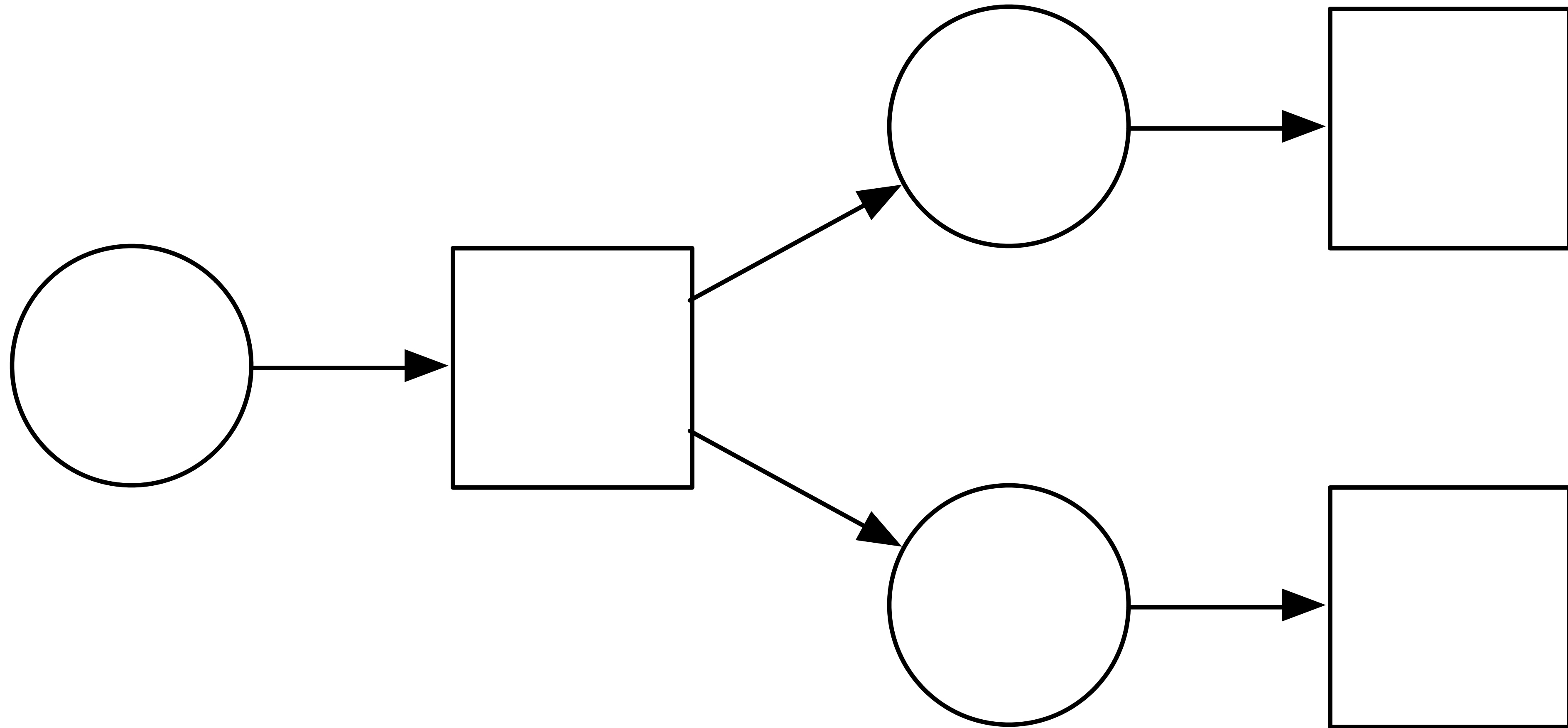
```
auto x = async([]{ return fibonacci<cpp_int>(1'000'000); });
auto y = async([]{ return fibonacci<cpp_int>(2'000'000); });

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: 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); });
```

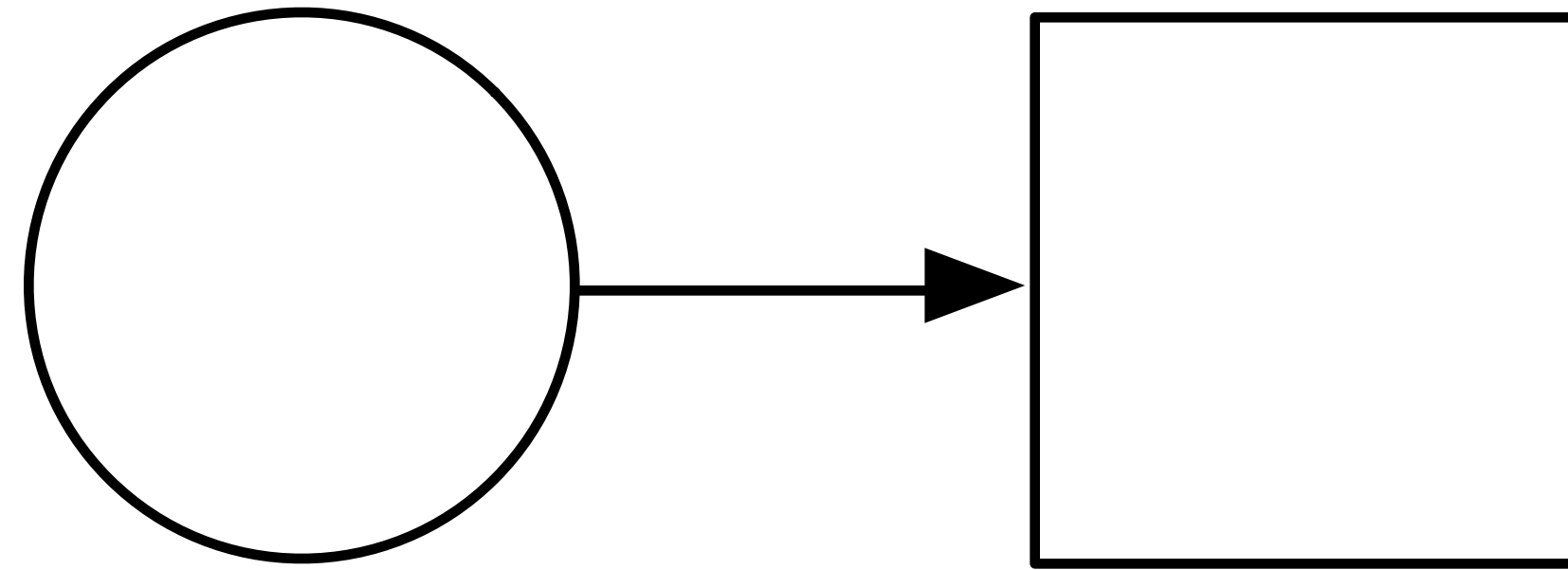
```
future<cpp_int> y = x.then([](future<cpp_int> x){ return cpp_int(x.get() * 2); });
```

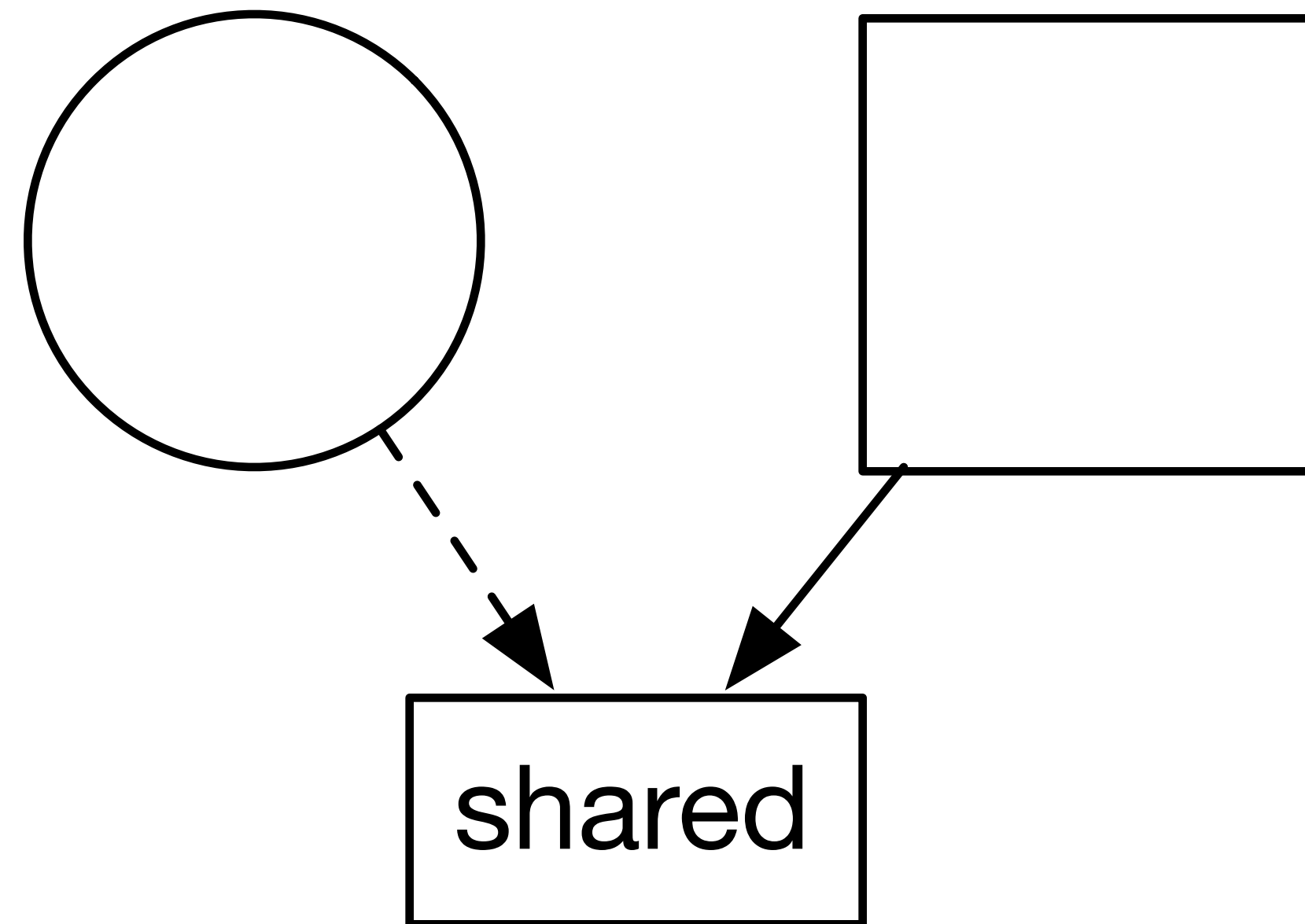
```
future<cpp_int> z = x.then([](future<cpp_int> x){ return cpp_int(x.get() / 15); });
```

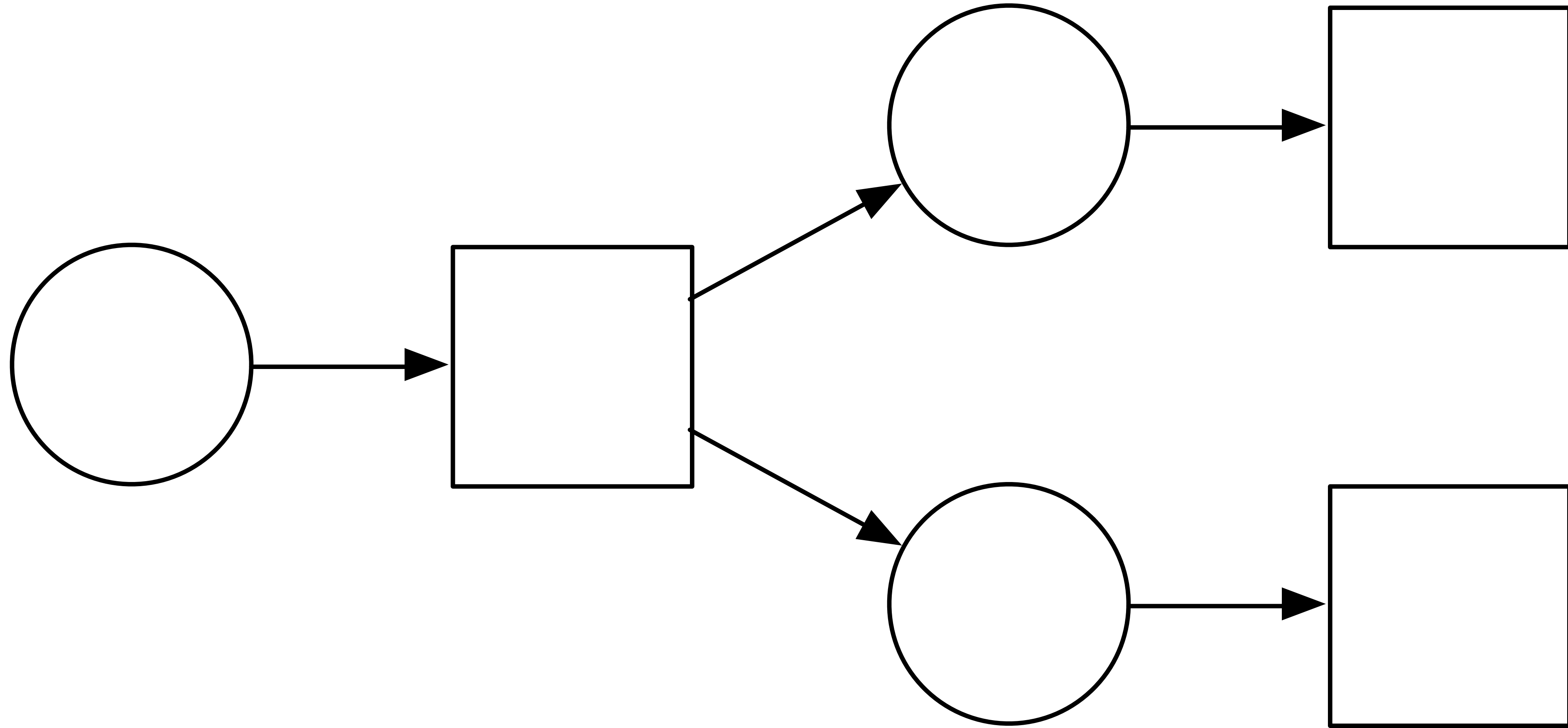
Thread 1: signal SIGABRT

**Assertion failed: (px != 0), function operator->, file shared\_ptr.hpp, line 648.**

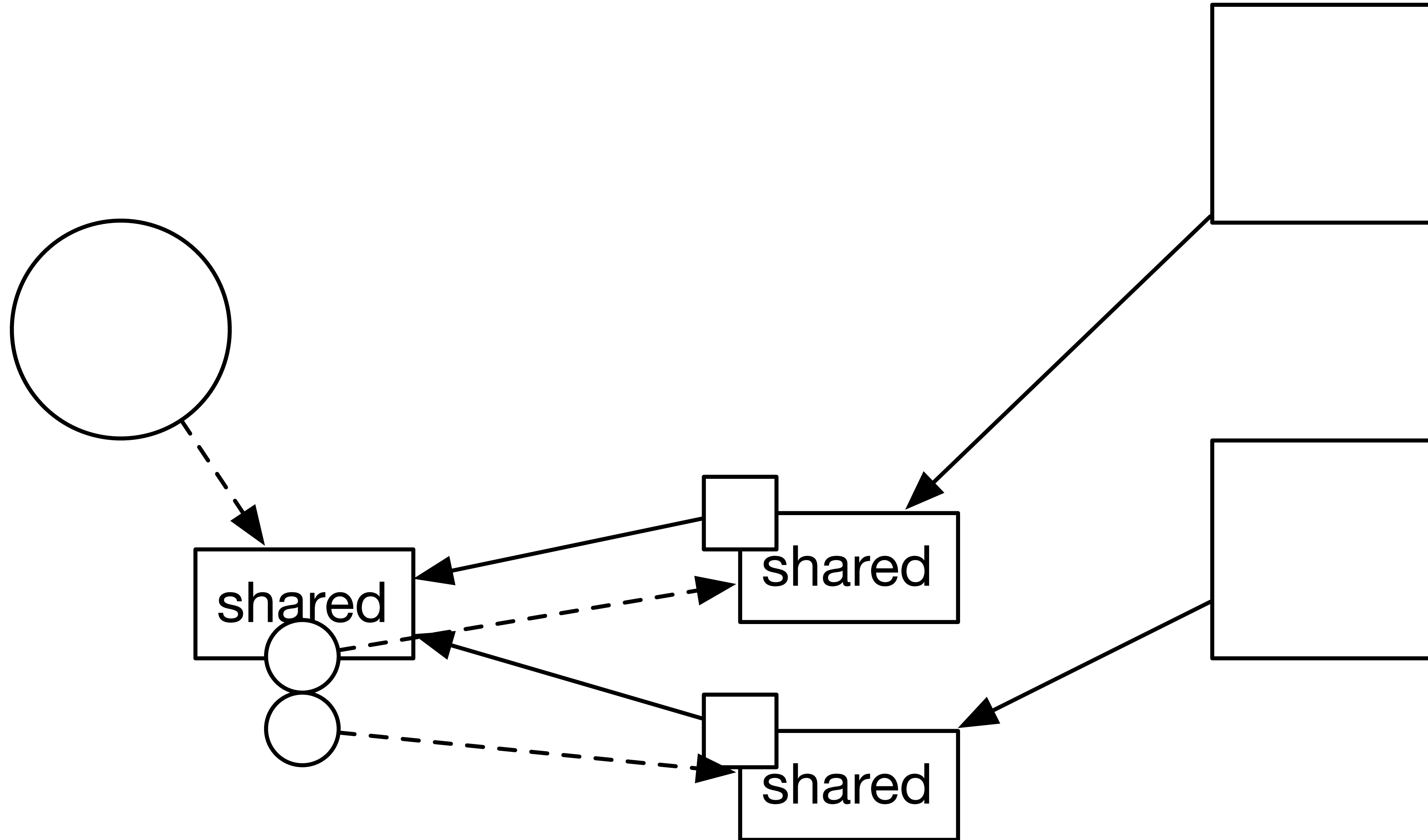
- Desired behavior
  - A future should behave as a *regular* type - a token for the actual value
    - `shared_futures` let me pass them around and do multiple `get()` operations, but don't fix continuations
      - [at least with boost]
  - Releasing the last instance of a future should cancel (no-op) any unexecuted, contributing, operations











# Futures: Building, The

```
template <typename>
struct result_of_; //not defined

template <typename R, typename... Args>
struct result_of_<R(Args...)> { using type = R; };

template <typename F>
using result_of_t_ = typename result_of_<F>::type;
```

# Futures: Building, The

```
template <typename>
struct result_of_; //not defined

template <typename R, typename... Args>
struct result_of_<R(Args...)> { using type = R; };

template <typename F>
using result_of_t_ = typename result_of_<F>::type;
```

result\_of\_t\_<int(double)> -> int

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
```

```
template <typename R>  
class future {  
    shared_ptr</* ... */> _p;
```

```
public:  
    future() = default;
```

```
    template <typename F>  
    auto then(F&& f) { }
```

```
    const R& get() const { }  
};
```

```
template<typename R, typename ...Args >  
class packaged_task<R (Args...)> {  
    weak_ptr</* ... */> _p;
```

```
public:  
    packaged_task() = default;
```

```
    template <typename... A>  
    void operator()(A&&... args) const { }  
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
```

```
template <typename R>  
class future {  
    shared_ptr</* ... */> _p;
```

```
public:  
    future() = default;
```

```
    template <typename F>  
    auto then(F&& f) { }
```

```
    const R& get() const { }  
};
```

```
template<typename R, typename ...Args >  
class packaged_task<R (Args...)> {  
    weak_ptr</* ... */> _p;
```

```
public:  
    packaged_task() = default;
```

```
    template <typename... A>  
    void operator()(A&&... args) const { }  
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
```

```
template <typename R>
class future {
    shared_ptr</* ... */> _p;

public:
    future() = default;

    template <typename F>
    auto then(F&& f) { }

    const R& get() const { }
};
```

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const { }
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
```

```
template <typename R>
class future {
    shared_ptr< /* ... */> _p;

public:
    future() = default;

    template <typename F>
    auto then(F&& f) { }

    const R& get() const { }
};
```

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr< /* ... */> _p;

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const { }
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
```

```
template <typename R>
class future {
    shared_ptr</* ... */> _p;

public:
    future() = default;

    template <typename F>
    auto then(F&& f) { }

    const R& get() const { }
};
```

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const { }
};
```



# Futures: Building, The

```
template <typename> class packaged_task; //not defined
template <typename> class future;

template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

template <typename R>
class future {
    shared_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};

template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
template <typename> class future;

template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

template <typename R>
class future {
    shared_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};

template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
template <typename> class future;

template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

template <typename R>
class future {
    shared_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};

template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};
```

# Futures: Building, The

```
template <typename> class packaged_task; //not defined
template <typename> class future;

template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

template <typename R>
class future {
    shared_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};

template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr</* ... */> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr</* ... */> p) : _p(move(p)) { }
    /* ... */
};
```

# Futures: Building, The

```
template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>> {
    auto p = make_shared<shared<S>>(forward<F>(f));
    return make_pair(packaged_task<S>(p), future<result_of_t_<S>>(p));
}
```

# Futures: Building, The

```
template <typename S, typename F>
auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>> {
    auto p = make_shared<shared<S>>(forward<F>(f));
    return make_pair(packaged_task<S>(p), future<result_of_t_<S>>(p));
}
```

```
package<int(double)>(f) -> { void(double), future<int> }
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    /* ... */
};

template <typename> struct shared; // not defined

template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    /* ... */
};
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    /* ... */
};

template <typename> struct shared; // not defined

template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    /* ... */
};
```



# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    /* ... */
};

template <typename> struct shared; // not defined

template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    /* ... */
};
```

# Futures: Building, The

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr<shared<R(Args...)>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr<shared<R(Args...)>> p) : _p(move(p)) { }

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const {
        auto p = _p.lock();
        if (p) (*p)(forward<A>(args)...);
    }
};
```

# Futures: Building, The

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr<shared<R(Args...)>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr<shared<R(Args...)>> p) : _p(move(p)) { }

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const {
        auto p = _p.lock();
        if (p) (*p)(forward<A>(args)...);
    }
};
```

# Futures: Building, The

```
template<typename R, typename ...Args >
class packaged_task<R (Args...)> {
    weak_ptr<shared<R(Args...)>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit packaged_task(weak_ptr<shared<R(Args...)>> p) : _p(move(p)) { }

public:
    packaged_task() = default;

    template <typename... A>
    void operator()(A&&... args) const {
        auto p = _p.lock();
        if (p) (*p)(forward<A>(args)...);
    }
};
```

# Futures: Building, The

```
template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    template <typename... A>
    void operator()(A&&... args) {
        this->set(_f(forward<A>(args)...));
        _f = nullptr;
    }
};
```

# Futures: Building, The

```
template <typename R, typename... Args>
struct shared<R(Args...)> : shared_base<R> {
    function<R(Args...)> _f;

    template<typename F>
    shared(F&& f) : _f(forward<F>(f)) { }

    template <typename... A>
    void operator()(A&&... args) {
        this->set(_f(forward<A>(args)...));
        _f = nullptr;
    }
};
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    void set(R&& r) {
        vector<function<void()>> then;
        {
            lock_t lock{_mutex};
            _r.push_back(move(r));
            swap(_then, then);
        }
        _ready.notify_all();
        for (const auto& f : then) _system.async_(move(f));
    }
};
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    void set(R&& r) {
        vector<function<void()>> then;
        {
            lock_t lock{_mutex};
            _r.push_back(move(r));
            swap(_then, then);
        }
        _ready.notify_all();
        for (const auto& f : then) _system.async_(move(f));
    }
};
```



# Futures: Building, The

```
template <typename R>
class future {
    shared_ptr<shared_base<R>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr<shared_base<R>> p) : _p(move(p)) { }
public:
    future() = default;

    template <typename F>
    auto then(F&& f) {
        auto pack = package<result_of_t<F(R)>>()>([p = _p, f = forward<F>(f)]()){
            return f(p->_r.back());
        });
        _p->then(move(pack.first));
        return pack.second;
    }

    const R& get() const { return _p->get(); }
};
```

# Futures: Building, The

```
template <typename R>
class future {
    shared_ptr<shared_base<R>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr<shared_base<R>> p) : _p(move(p)) { }
public:
    future() = default;

    template <typename F>
    auto then(F&& f) {
        auto pack = package<result_of_t<F(R)>()>([p = _p, f = forward<F>(f)]()){
            return f(p->_r.back());
        });
        _p->then(move(pack.first));
        return pack.second;
    }

    const R& get() const { return _p->get(); }
};
```

# Futures: Building, The

```
template <typename R>
class future {
    shared_ptr<shared_base<R>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr<shared_base<R>> p) : _p(move(p)) { }
public:
    future() = default;

    template <typename F>
    auto then(F&& f) {
        auto pack = package<result_of_t<F(R)>>()>([p = _p, f = forward<F>(f)]()){
            return f(p->_r.back());
        });
        _p->then(move(pack.first));
        return pack.second;
    }

    const R& get() const { return _p->get(); }
};
```

# Futures: Building, The

```
template <typename R>
class future {
    shared_ptr<shared_base<R>> _p;

    template <typename S, typename F>
    friend auto package(F&& f) -> pair<packaged_task<S>, future<result_of_t_<S>>>;

    explicit future(shared_ptr<shared_base<R>> p) : _p(move(p)) { }
public:
    future() = default;

    template <typename F>
    auto then(F&& f) {
        auto pack = package<result_of_t<F(R)>>()>([p = _p, f = forward<F>(f)]()){
            return f(p->_r.back());
        });
        _p->then(move(pack.first));
        return pack.second;
    }

    const R& get() const { return _p->get(); }
};
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    void set(R&& r) { ... }

    template <typename F>
    void then(F&& f) {
        bool resolved{false};
        {
            lock_t lock{_mutex};
            if (_r.empty()) _then.push_back(forward<F>(f));
            else resolved = true;
        }
        if (resolved) _system.async_(move(f));
    }

    const R& get() {
        lock_t lock{_mutex};
        while (_r.empty()) _ready.wait(lock);
        return _r.back();
    }
};
```

# Futures: Building, The

```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

    void set(R&& r) { ... }

    template <typename F>
    void then(F&& f) {
        bool resolved{false};
        {
            lock_t lock{_mutex};
            if (_r.empty()) _then.push_back(forward<F>(f));
            else resolved = true;
        }
        if (resolved) _system.async_(move(f));
    }

    const R& get() {
        lock_t lock{_mutex};
        while (_r.empty()) _ready.wait(lock);
        return _r.back();
    }
};
```

# Futures: Building, The

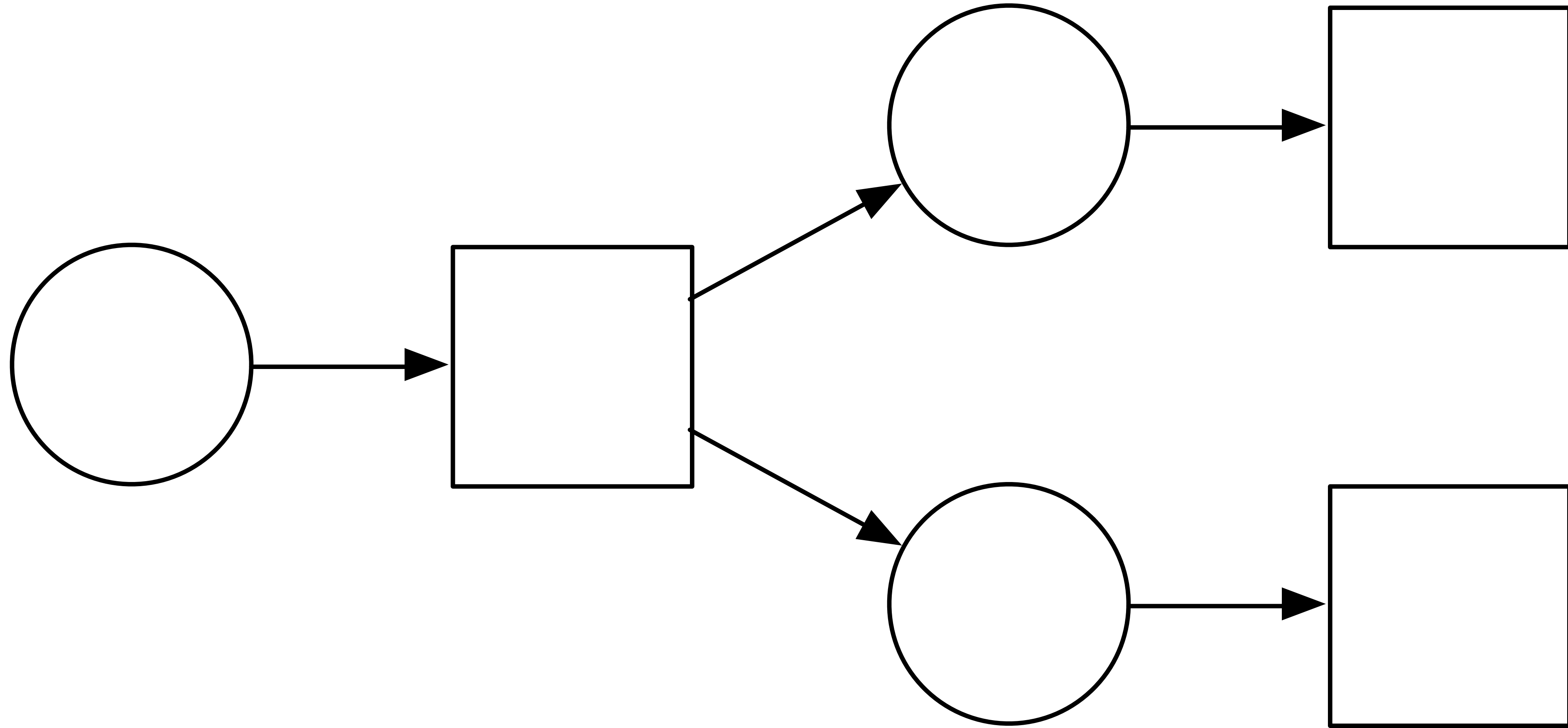
```
template <typename R>
struct shared_base {
    vector<R> _r; // optional
    mutex _mutex;
    condition_variable _ready;
    vector<function<void()>> _then;

    virtual ~shared_base() { }

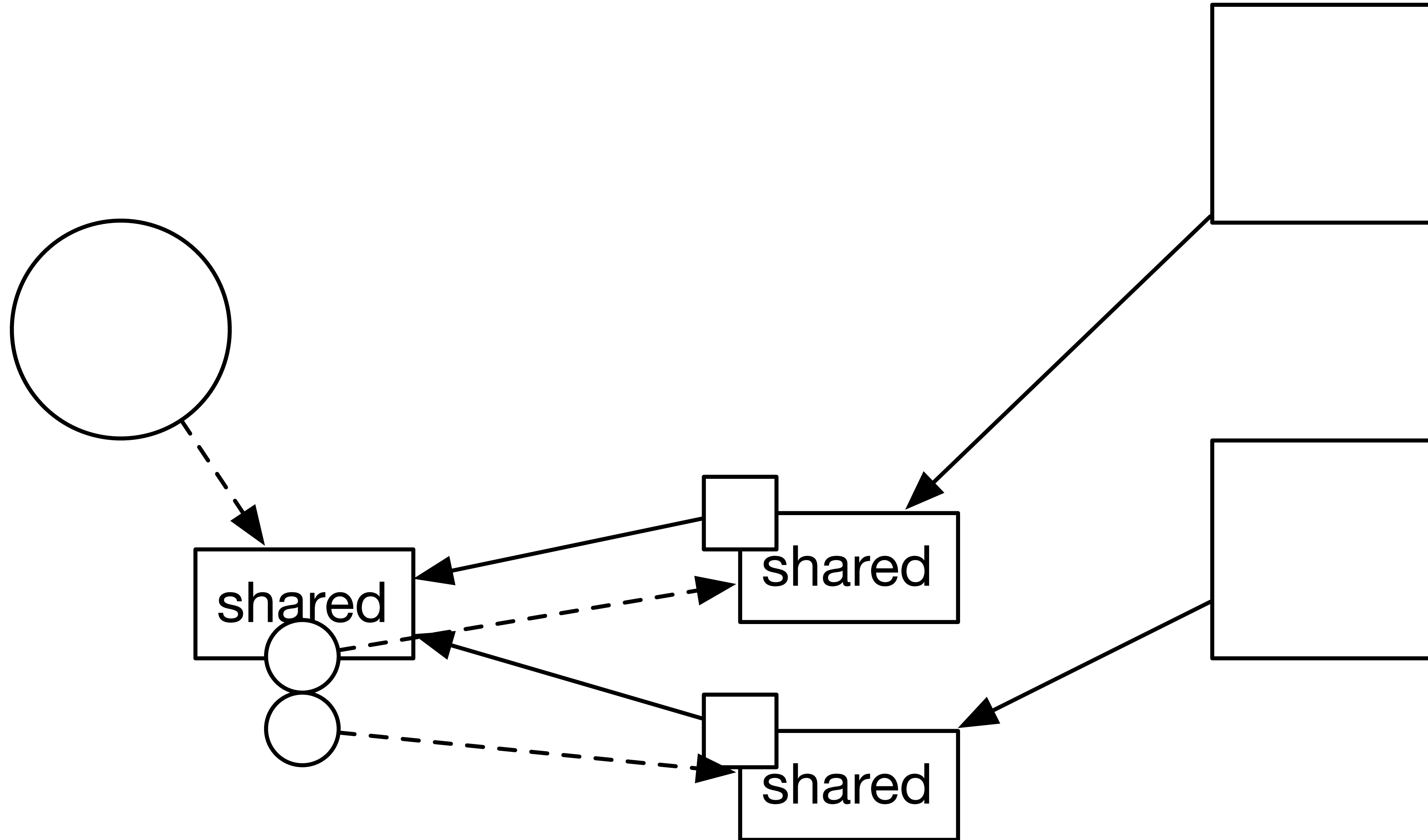
    void set(R&& r) { ... }

    template <typename F>
    void then(F&& f) {
        bool resolved{false};
        {
            lock_t lock{_mutex};
            if (_r.empty()) _then.push_back(forward<F>(f));
            else resolved = true;
        }
        if (resolved) _system.async_(move(f));
    }

    const R& get() {
        lock_t lock{_mutex};
        while (_r.empty()) _ready.wait(lock);
        return _r.back();
    }
};
```







# Futures: Building, The

```
template <typename F, typename ...Args>
auto async(F&& f, Args&&... args)
{
    using result_type = result_of_t<F (Args...)>;
    using packaged_type = packaged_task<result_type>;

    auto pack = package<result_type>(bind(forward<F>(f), forward<Args>(args)...));

    _system.async_(move(get<0>(pack)));
    return get<1>(pack);
}
```

# Futures: Continuations

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

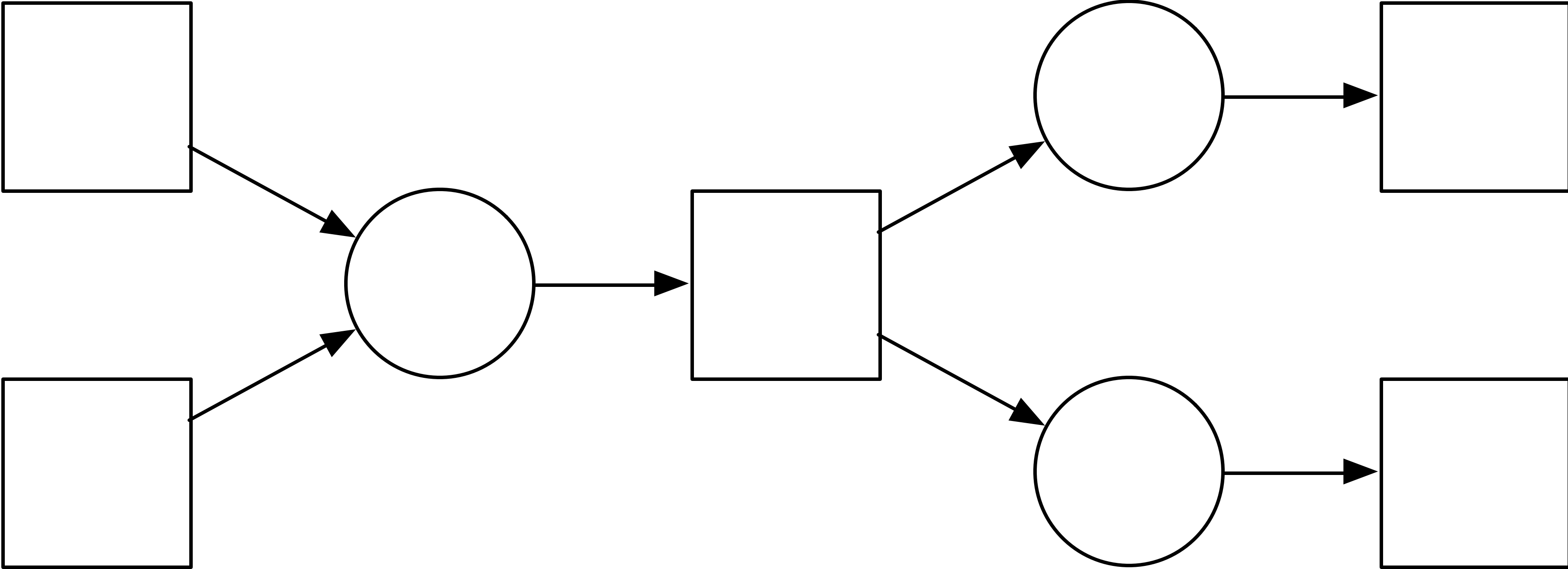
# Futures: Continuations

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

```
708449696358523830150  
23614989878617461005
```

# Property Models

# What if we persist the graph?

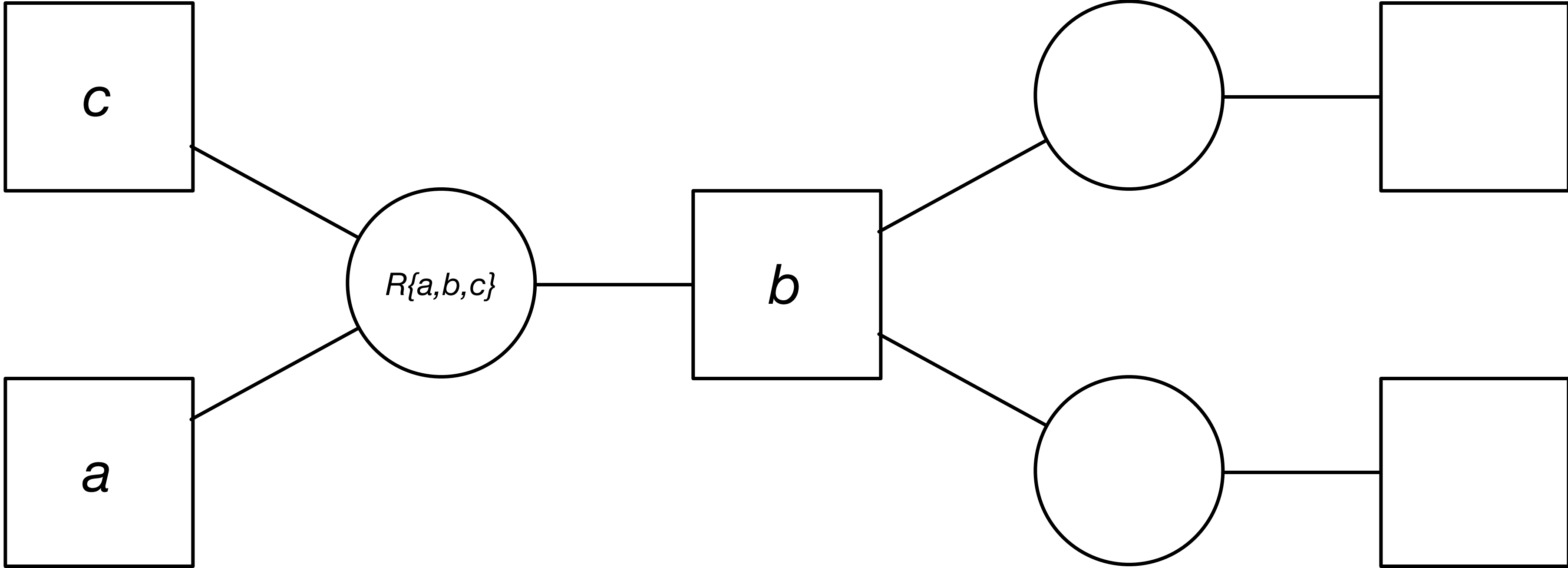


## What if we persist the graph?

- 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*

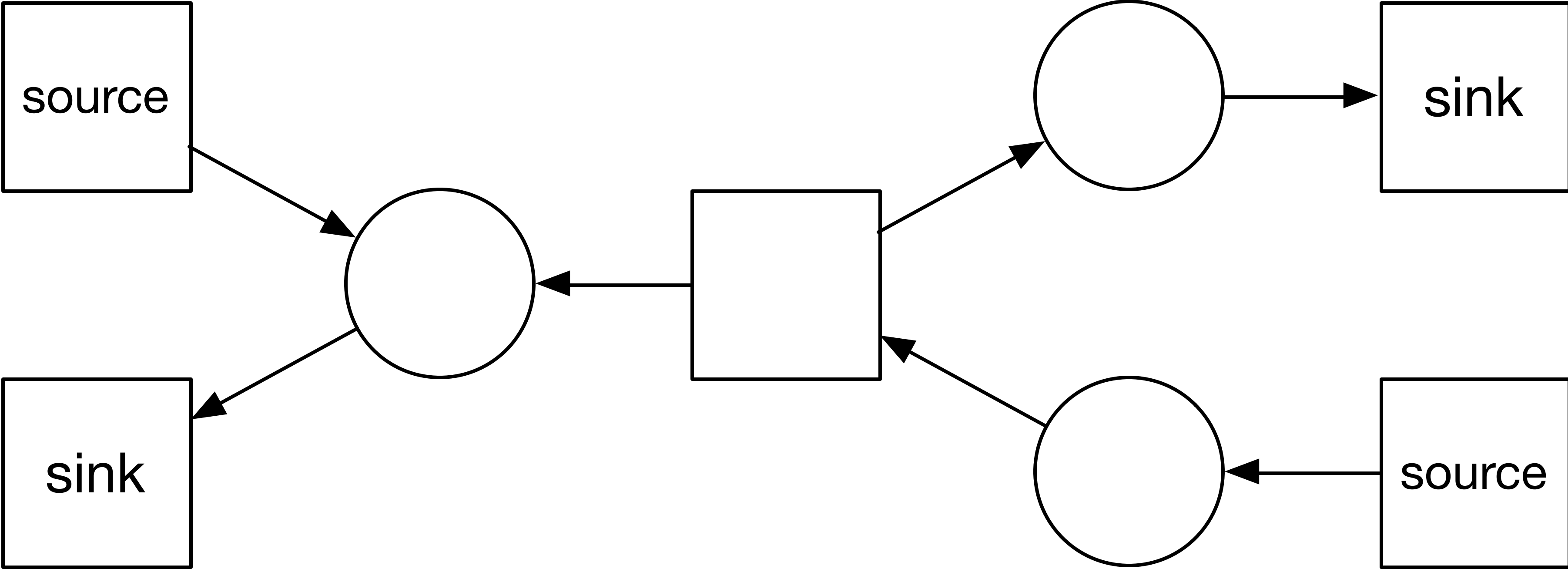
# How do the graphs change during execution?





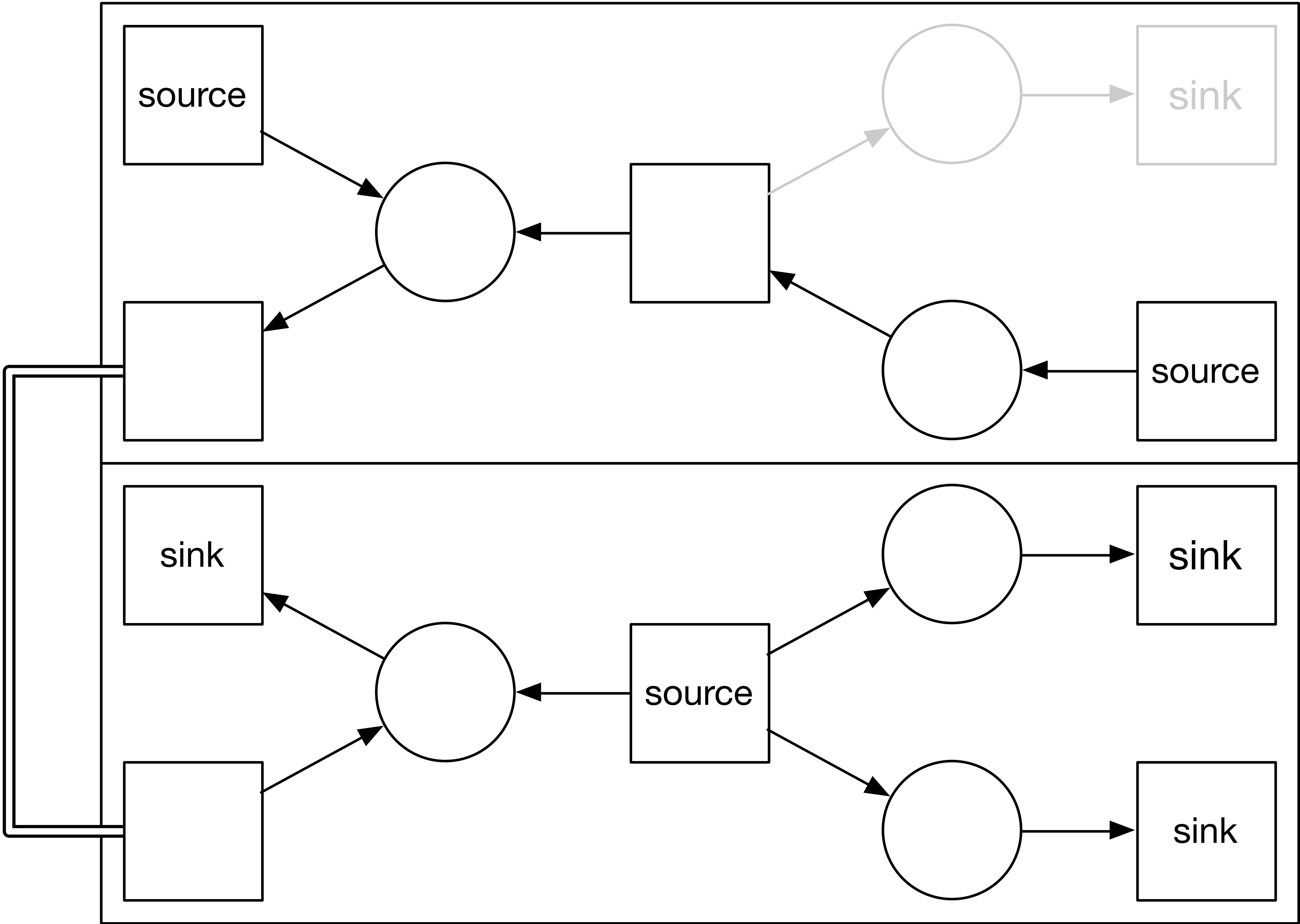
## A function is a directed relationship

- We can remove the arrows by providing a package of functions to represent the relationship
  - $a = b * c$   
 $b = a / c$   
 $c = a / b$
  - This forms a type of constraint system called a *property model*
  - Flow is determined by value, or *cell*, priority
- Cells can only have one in-edge for a given flow or the system is over constrained



- Reflowing a property model doesn't require all relationships to be resolved
- The task representing them can still be executing concurrently
- This creates a single dependency graph that is appended to for each new flow and is pruned and *unravels* as tasks are complete

# Property Model



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



**Adobe**