C++11 Concurrency

Agenda

- High-level components
- Low-level lock-based components

It’s a standard!

- The technology may be old, but having it as an international standard is brand new
  - More portable code
  - Common facilities

High-Level Components

std::thread

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

Hello, World
(en Français)

French to English Dictionary

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```


boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

Look up the greeting

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

next lookup

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

wait for I/O to complete

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

wait for I/O to complete

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

It's a wrap!

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t(
        [](std::string const& x){std::cout << x << ", ";},
        std::ref(greet) );
    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t(
        [](std::string const& x){std::cout << x << ", ";},
        std::ref(greet) );
    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

Uses std::bind() protocol

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t(
        [](std::string const& x){std::cout << x << ", ";},
        std::ref(greet) );
    std::string audience = french["world"];
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

std::thread

```
struct thread {
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();

  void swap(thread&) noexcept;
  thread(thread&&) noexcept;
  thread& operator=(thread&&) noexcept;
  thread(const thread&) = delete;
  thread& operator=(const thread&) = delete;

  bool joinable() const noexcept;
  void join();
  void detach();
```

boostpro computing

std::thread

```
struct thread {
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();

  void swap(thread&) noexcept;
  thread(thread&&) noexcept;
  thread& operator=(thread&&) noexcept;
  thread(const thread&) = delete;
  thread& operator=(const thread&) = delete;

  bool joinable() const noexcept;
  void join();
  void detach();
```

Move-only

std::thread

```
struct thread {
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();

  void swap(thread&) noexcept;
  thread(thread&&) noexcept;
  thread& operator=(thread&&) noexcept;
  thread(const thread&) = delete;
  thread& operator=(const thread&) = delete;

  bool joinable() const noexcept;
  void join();
  void detach();
```

std::thread

```
struct thread {
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();
  bool joinable() const noexcept;
  void join();
  void detach();
  static unsigned hardware_concurrency() noexcept;

  class id;
  id get_id() const noexcept;
  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

Move-only

std::thread

```
struct thread {  move-only
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();
  bool joinable() const noexcept;
  void join();
  void detach();
  static unsigned hardware_concurrency() noexcept;

  class id;
  id get_id() const noexcept;
  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

We'll abbreviate it like this

boostpro computing

std::thread

```
struct thread {  move-only
  thread() noexcept;
  template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args);
  ~thread();
  bool joinable() const noexcept;
  void join();
  void detach();
  static unsigned hardware_concurrency() noexcept;

  class id;
  id get_id() const noexcept;
  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

boostpro computing

What's Not Here

- Thread priorities
- Scheduling control
- Other OS-specific details
- Use the `native_handle()` for these things if you need them

joinability

- a `joinable()` thread has a non-default id
- Must be `joinable()` when:
  - `join()`ed
  - `detach()`ed
- Must *not* be `joinable()` when:
  - destroyed
  - move-assigned

joinability

- a `joinable()` thread has a non-default id
- Must be `joinable()` when:
  - `join()`ed
  - `detach()`ed
- Must *not* be `joinable()` when:
  - destroyed
  - move-assigned

The wages of joinability is…

joinability

- a `joinable()` thread has a non-default id
- Must be `joinable()` when:
  - `join()`ed
  - `detach()`ed
- Must *not* be `joinable()` when:
  - destroyed
  - move-assigned

The wages of joinability is… terminate()!

joinability

- a `joinable()` thread has a non-default id
- Must be `joinable()` when:
    - `join()`ed
    - `detach()`ed
- Must *not* be `joinable()` when:
    - destroyed
    - move-assigned

boostpro computing

Exceptions

- Threading operations throw `system_error`
- Thread launch may throw if system resources are used up
- `detach()` and `join()` will throw if
  - the thread is not `joinable()`
  - deadlock is detected (join only)
- Thread functions must *not* leak exceptions

boostpro computing

Exceptions

- Threading operations throw `system_error`
- Thread launch may throw if system resources are used up
- `detach()` and `join()` will throw if
  - the thread is not `joinable()`
  - deadlock is detected (join only)
- Thread functions must *not* leak exceptions

The wages of exceptional thread exit is…

boostpro computing

Exceptions

- Threading operations throw `system_error`
- Thread launch may throw if system resources are used up
- `detach()` and `join()` will throw if
  - the thread is not `joinable()`
  - deadlock is detected (join only)
- Thread functions must *not* leak exceptions

The wages of exceptional thread exit is… terminate()!

boostpro computing

Exceptions

- Threading operations throw `system_error`
- Thread launch may throw if system resources are used up
- `detach()` and `join()` will throw if
  - the thread is not `joinable()`
  - deadlock is detected (join only)
- Thread functions must *not* leak exceptions

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

preventing termination

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

Uhh…

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;

    template <class Clock, class Duration>
    void sleep_until(
      const chrono::time_point<Clock,Duration>& abs_time);

    template <class Rep, class Period>
    void sleep_for(
      const chrono::duration<Rep,Period>& rel_time);
}
```

boostpro computing

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;

    template <class Clock, class Duration>
    void sleep_until(
      const chrono::time_point<Clock,Duration>& abs_time);

    template <class Rep, class Period>
    void sleep_for(
      const chrono::duration<Rep,Period>& rel_time);
}
```

This pattern repeats

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;

    template <class Clock, class Duration>
    void sleep_until(
      const chrono::time_point<Clock,Duration>& abs_time);

    template <class Rep, class Period>
    void sleep_for(
      const chrono::duration<Rep,Period>& rel_time);
}
```

boostpro computing

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;


    void sleep_until( time_point );



    void sleep_for( duration );

}
```

boostpro computing

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;

    void sleep_until( time_point );



    void sleep_for( duration );

}
```

We'll abbreviate it like this

boostpro computing

this_thread

```
namespace this_thread
{
    thread::id get_id() noexcept;
    void yield() noexcept;

    void sleep_until( time_point );



    void sleep_for( duration );

}
```

std::async
and
std::future

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

preventing termination

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

Uhh…

boostpro computing

Hello, World
(en Français)

```
#include <thread>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::thread t( [&]{std::cout << greet << ", ";} );
    try { std::string audience = french["world"]; }
    catch(...) { t.join(); throw; }
    t.join();
    std::cout << audience << std::endl;
}
```

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

Look up the greeting

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

next lookup

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

time-consuming I/O

wait for I/O to complete

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

wait for I/O to complete

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    auto f = std::async([&]{std::cout << greet << ", ";} );

    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::future<void> f = std::async(
        [&]{std::cout << greet << ", ";} );
    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::future<void> f = std::async(
        [&]{std::cout << greet << ", ";} );
    std::string audience = french["world"];
    f.get();
    std::cout <
}
```

“retrieve” the (void) value…

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::future<void> f = std::async(
        [&]{std::cout << greet << ", ";} );
    std::string audience = french["world"];
    f.get();
    std::cout <
}
```

“retrieve” the (void) value…
…or rethrow the exception

boostpro computing

A Better Approach

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::string greet = french["hello"];
    std::future<void> f = std::async(
        [&]{std::cout << greet << ", ";} );
    std::string audience = french["world"];
    f.get();
    std::cout << audience << std::endl;
}
```

boostpro computing

std::future Values

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    auto greet = std::async([]{ return french["hello"]; });

    std::string audience = french["world"];
    std::cout << greet.get()
              << ", " << audience << std::endl;
}
```

boostpro
computing

std::future Values

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    auto greet = std::async([]{ return french["hello"]; });

    std::string audience = french["world"];
    std::cout << greet.get()
              << ", " << audience << std::endl;
}
```

std::future<std::string>

boostpro computing

std::future Values

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    auto greet = std::async([]{ return french["hello"]; });

    std::string audience = french["world"];
    std::cout << greet.get()
              << ", " << audience << std::endl;
}
```

retrieve the string value

boostpro computing

Tuesday, May 15, 12

std::future Values

```
#include <future>

std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    auto greet = std::async([]{ return french["hello"]; });

    std::string audience = french["world"];
    std::cout << greet.get()
              << ", " << audience << std::endl;
}
```

async Launch Policy

```
enum class launch : unspecified // a bitmask type
{
    async = unspecified,          // asynchronous, in a thread
    deferred = unspecified,       // synchronous, on demand
    implementation-defined        // your vendor's stuff here
};
```

async Launch Policy

```
enum class launch : unspecified // a bitmask type
{
    async = unspecified,          // asynchronous, in a thread
    deferred = unspecified,       // synchronous, on demand
    implementation-defined        // your vendor's stuff here
};
```

"`async | deferred`" is the default. It means, "asynchronous if there's parallelism to be had"… …but GCC's libstdc++ just treats it like `deferred`

boostpro computing

async Launch Policy

```
enum class launch : unspecified // a bitmask type
{
    async = unspecified,          // asynchronous, in a thread
    deferred = unspecified,       // synchronous, on demand
    implementation-defined        // your vendor's stuff here
};
```

"`async | deferred`" is the default. It means, "asynchronous if there's parallelism to be had"…
…but GCC's libstdc++ just treats it like `deferred`

(see http://gcc.gnu.org/bugzilla/show_bug.cgi?id=51617)

boostpro computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

boostpro computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

divide

boostpro
computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

conquer

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

unify

boostpro computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

handle short sequences synchronously

boostpro computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

use available threads for longer sequences

boostpro computing

Using Launch Policies

```
template <class Iter>
void parallel_merge_sort(Iter start, Iter finish) {
    std::size_t d = std::distance(start, finish);
    if (d <= 1) return;
    Iter mid = start; std::advance(mid, d/2);

    auto f = std::async(
        d < 768 ? std::launch::deferred
        : std::launch::deferred | std::launch::async
        [=]{ parallel_merge_sort(start,mid); });

    parallel_merge_sort(mid,finish);
    f.get();
    std::inplace_merge(start,mid,finish);
}
```

boostpro computing

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

valid() ⇔ there is, or will be, a result we can get()

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();    one-shot! afterwards, valid() == false

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;        wait for "ready" state
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

boostpro computing

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

timed wait, for the impatient

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

timed wait, for the impatient

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

upgrade *this to multi-shot, copyable

boostpro computing

std::future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct future {  move-only
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future<R> share();
};
```

boostpro computing

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future { copyable
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

boostpro computing

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future { copyable
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

boostpro computing

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future {
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

copyable

basic thread safety: distinct copies can be used concurrently

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future { copyable
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

multi-shot: afterwards, valid() == true

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future { copyable
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

upgrade f to multi-shot, copyable

boostpro computing

std::shared_future

```
enum class future_status { ready, timeout, deferred };

template <class R> struct shared_future { copyable
  future() noexcept;
  bool valid() const noexcept;

  R get();

  void wait() const;
  future_status wait_for( duration ) const;
  future_status wait_until( time_point ) const;

  shared_future(future<R>&& f) noexcept;
};
```

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```


boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

"push" interface

boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

"pull" interface

boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

“pull” interface

pull it

boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

"pull" interface

boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

"pull" interface

passing the "pull" interface

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

push it

boostpro computing

Pushy, Pushy!

```
int main()
{
    std::promise<std::string> audience_send;

    auto greet = std::async(
        [](std::future<std::string> audience_rcv)
        {
            std::cout << french["hello"] << ",";
            std::cout << audience_rcv.get() << std::endl;
        },
        audience_send.get_future()
    );
    audience_send.set_value(french["world"]);
    greet.wait();
}
```

std::promise

```
template <class R>
struct promise {  move-only
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

boostpro computing

std::promise

```
template <class R>
struct promise {
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

move-only

allocate “shared state” for result storage

boostpro computing

std::promise

```
template <class R>
struct promise {  move-only
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();   one-shot: get the "pull" interface

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

boostpro computing

std::promise

```
template <class R>
struct promise {  move-only
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

push result (or exception) and make the future "ready"

boostpro computing

std::promise

```
template <class R>
struct promise {  move-only
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

push result but defer readiness

boostpro computing

std::promise

```
template <class R>
struct promise {  move-only
  promise();
  template <class Allocator>
    promise(allocator_arg_t, const Allocator& a);

  future<R> get_future();

  void set_value(R);
  void set_exception(exception_ptr p);

  void set_value_at_thread_exit(R);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

boostpro computing

std::exception_ptr

- Is-a NullablePointer (a value type constructible from nullptr)
- Get (a copy of) currently-handled exception

```
exception_ptr current_exception() noexcept;
```

- Throw the exception stored in p

```
[[noreturn]] void rethrow_exception(exception_ptr p);
```

- Create an exception_ptr to a copy of e

```
template <class E>
exception_ptr make_exception_ptr(E e) noexcept;
```

boostpro computing

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{


    auto greet = std::async([]{ return french["hello"]; });


    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
                              launches the lookup immediately

    auto greet = std::async([]{ return french["hello"]; });


    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

boostpro computing

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{


    auto greet = std::async([]{ return french["hello"]; });


    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    do_lookup();

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    do_lookup();

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

launch deferred

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    do_lookup();

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

get future result

boostpro computing

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    do_lookup();

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

synchronous launch

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    task_queue.push(std::move(do_lookup));

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

asynchronous launch

Deferring Launch

```
std::map<std::string, std::string> french
  {{"hello","bonjour"}, {"world", "tout le monde"}};

int main()
{
    std::packaged_task<std::string()> do_lookup(
       []{ return french["hello"]; });
    auto greet = do_lookup.get_future();
    task_queue.push(std::move(do_lookup));

    std::string audience = french["world"];
    std::cout << greet.get() << ", " << audience << std::endl;
}
```

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> { move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

boostpro computing

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

move-only

store f and allocate "shared state" for result

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {  move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();    get the "pull" interface
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {  move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;   has a shared state?

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {  move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

invoke f, make future “ready” w/result

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {  move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

invoke f but defer readiness

boostpro computing

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

move-only

allocate new shared state so we can get_future()/call again

std::packaged_task

```
template<class> class packaged_task; // undefined

template<class R, class... ArgTypes>
struct packaged_task<R(ArgTypes...)> {  move-only
  packaged_task() noexcept;
  template <class F> explicit packaged_task(F&& f);
  template <class F, class Alloc>
    explicit packaged_task(allocator_arg_t, const Alloc& a, F&& f);

  future<R> get_future();
  bool valid() const noexcept;

  void operator()(ArgTypes... );
  void make_ready_at_thread_exit(ArgTypes...);
  void reset();
};
```

boostpro computing

Lock-Based Data Sharing

Shared Mutable State

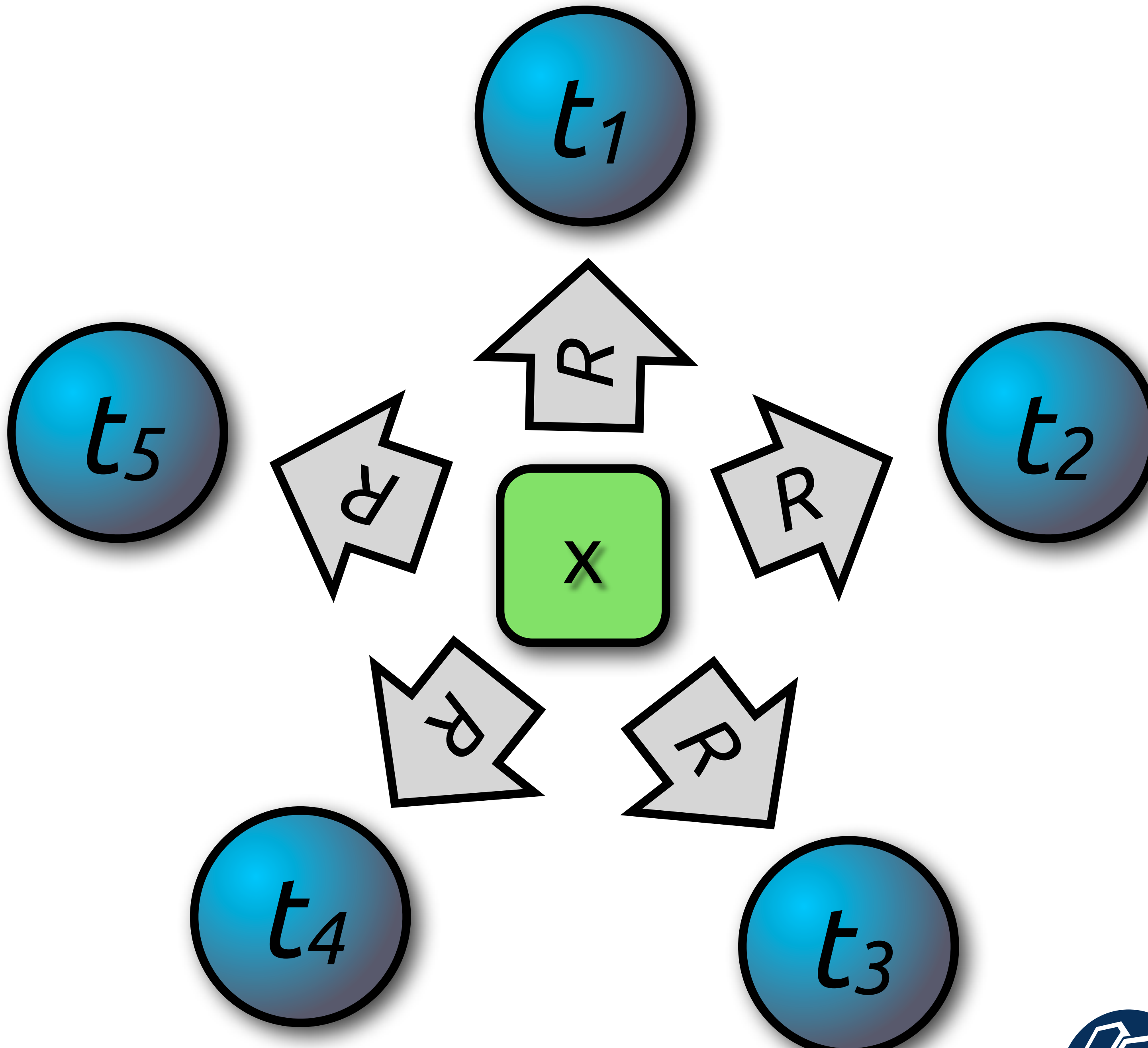
- So far all our examples have been
  - functional ("compute this and get back to me with the result") or
  - operating on iostreams, which are already threadsafe
- Avoiding shared, mutable data makes concurrency much simpler!
- However, only "embarrassingly parallel" problems can really be solved that way

boostpro computing

The Challenge of Data Races

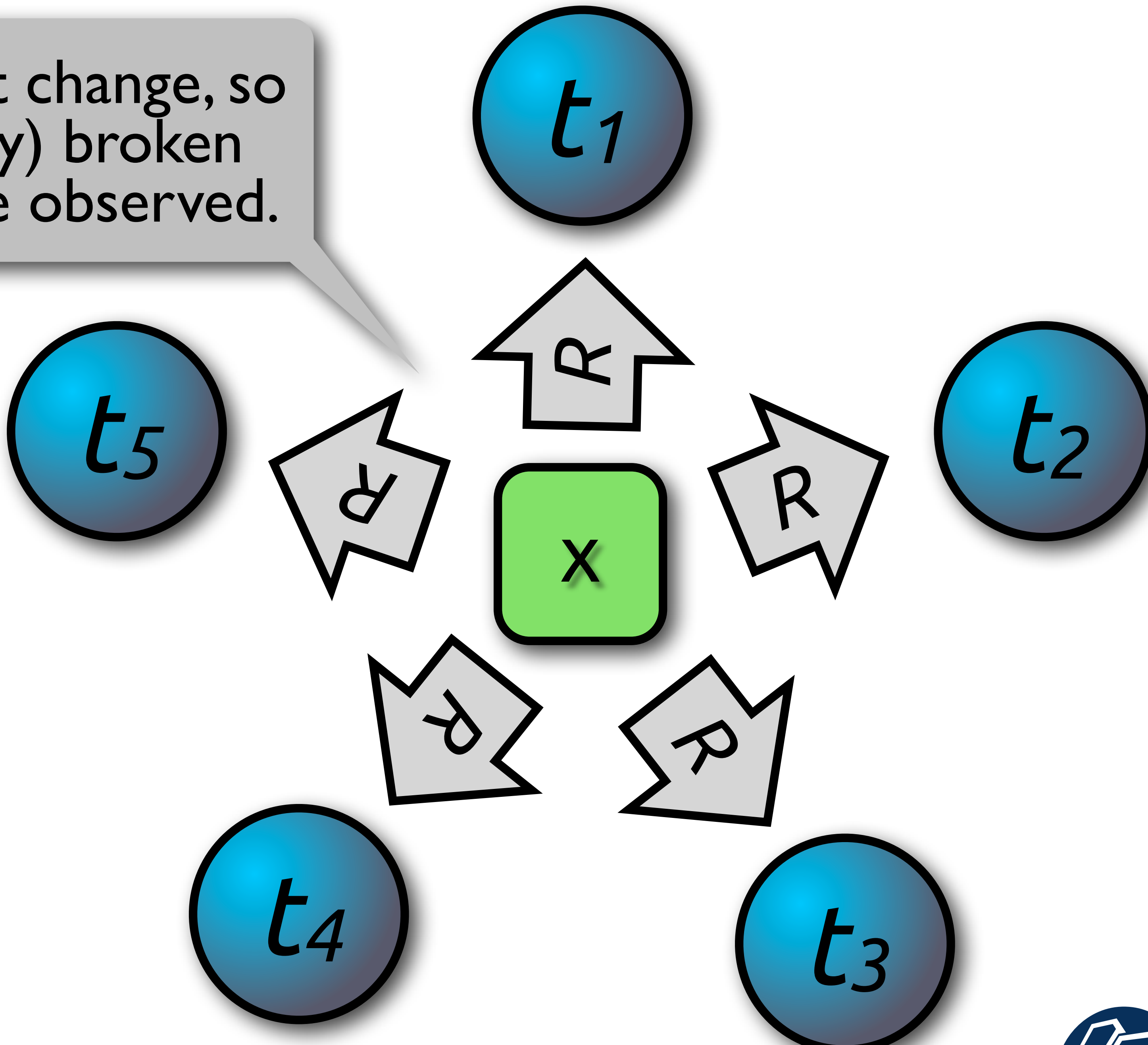
- Every piece of data has an invariant that is broken while it is being written
- Threads must not observe these broken invariants

Concurrent Reads



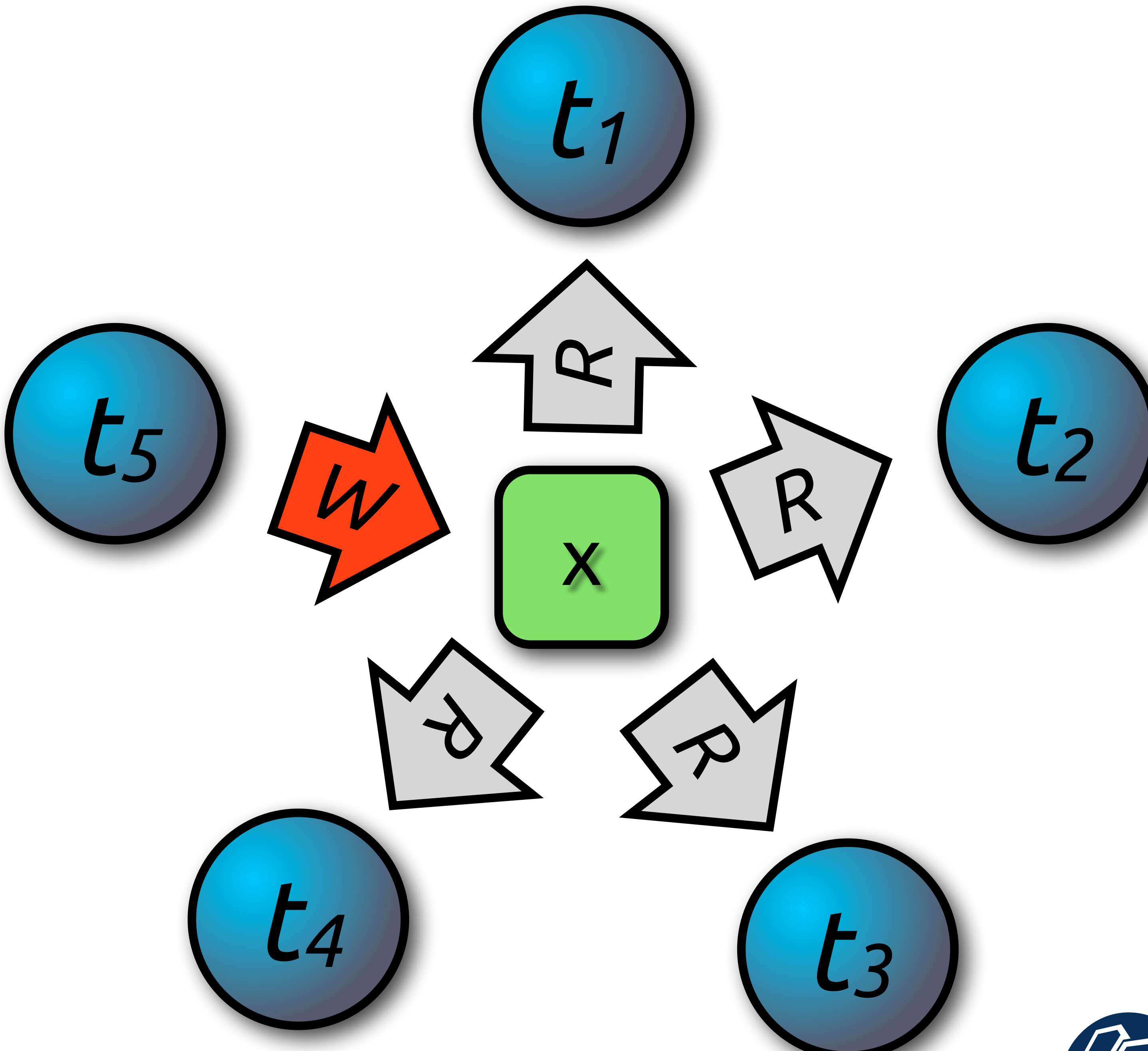
Concurrent Reads

OK: data doesn't change, so no (temporarily) broken invariants can be observed.



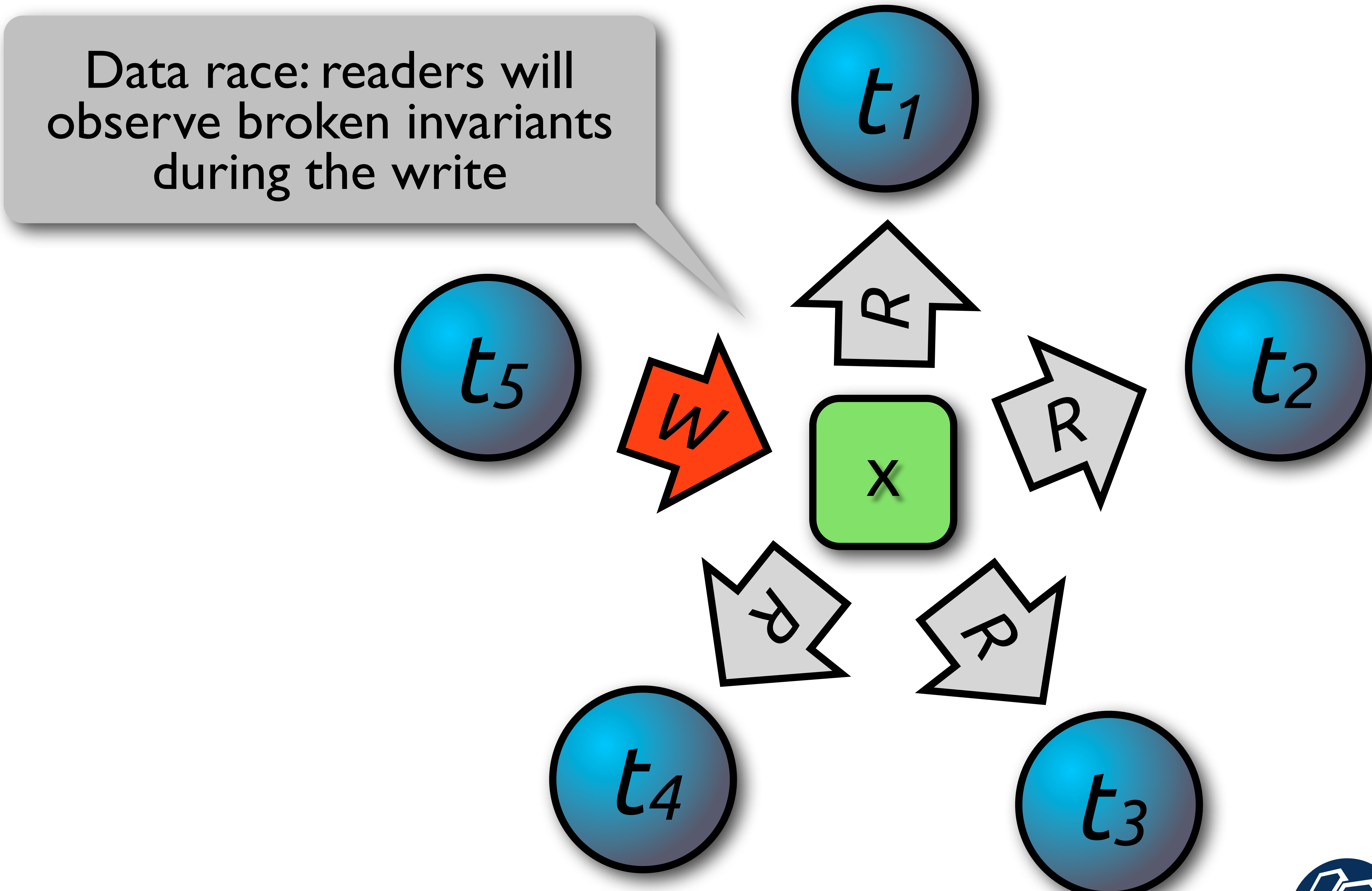
Tuesday, May 15, 12

Concurrent Read/Write



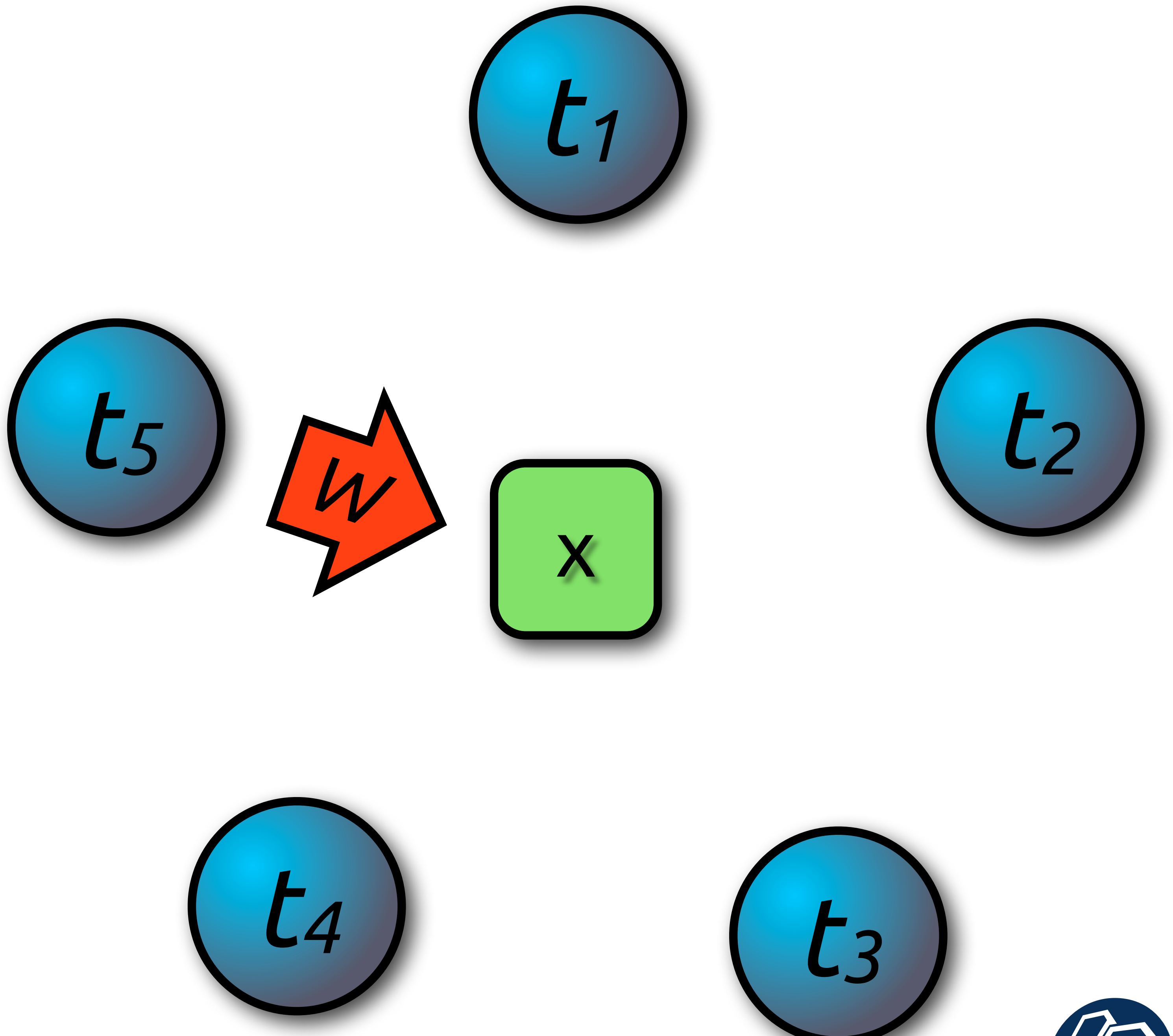
boostpro computing

Concurrent Read/Write



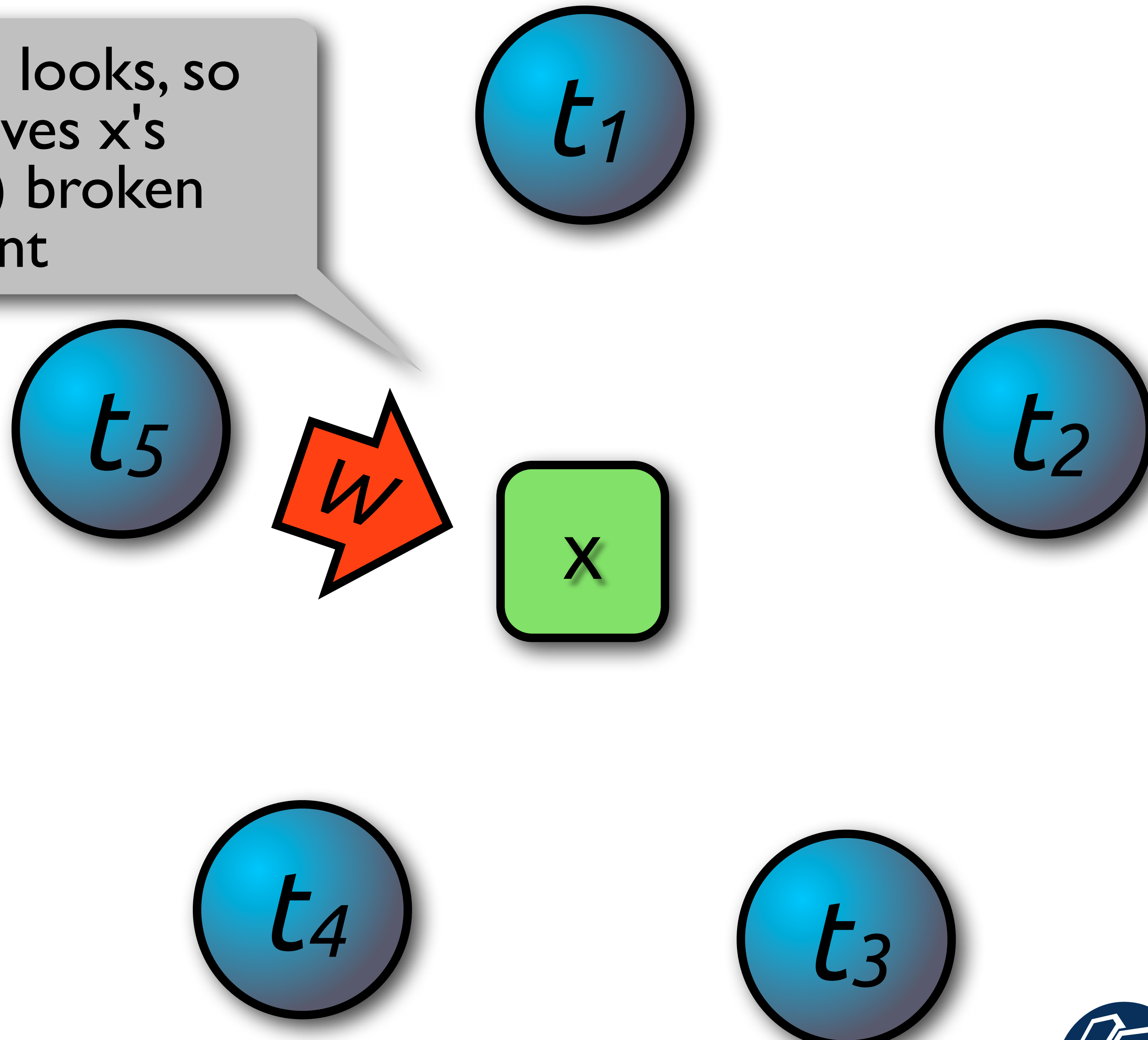
Tuesday, May 15, 12

Single Write



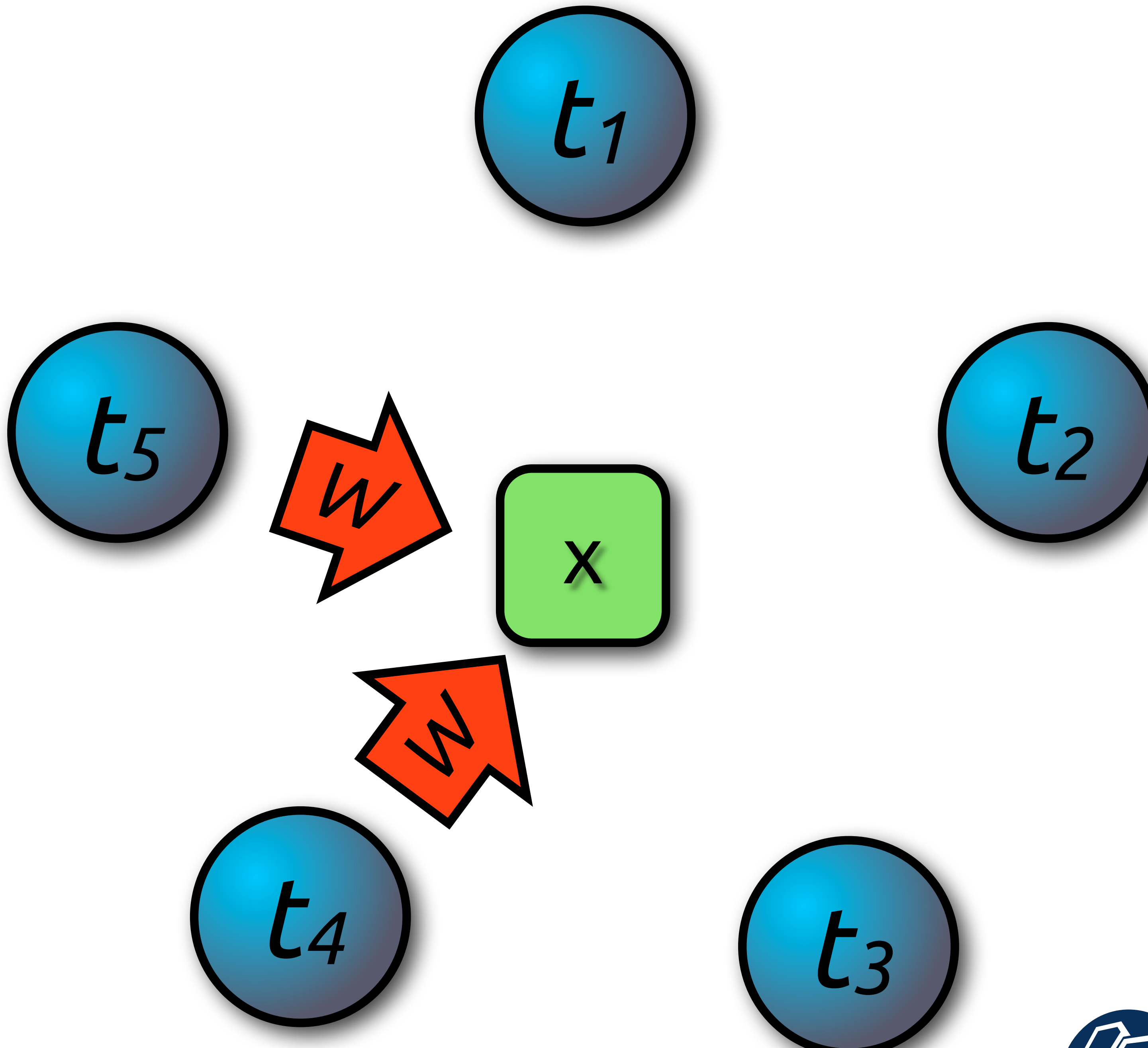
Single Write

OK: no thread looks, so none observes x's (temporarily) broken invariant

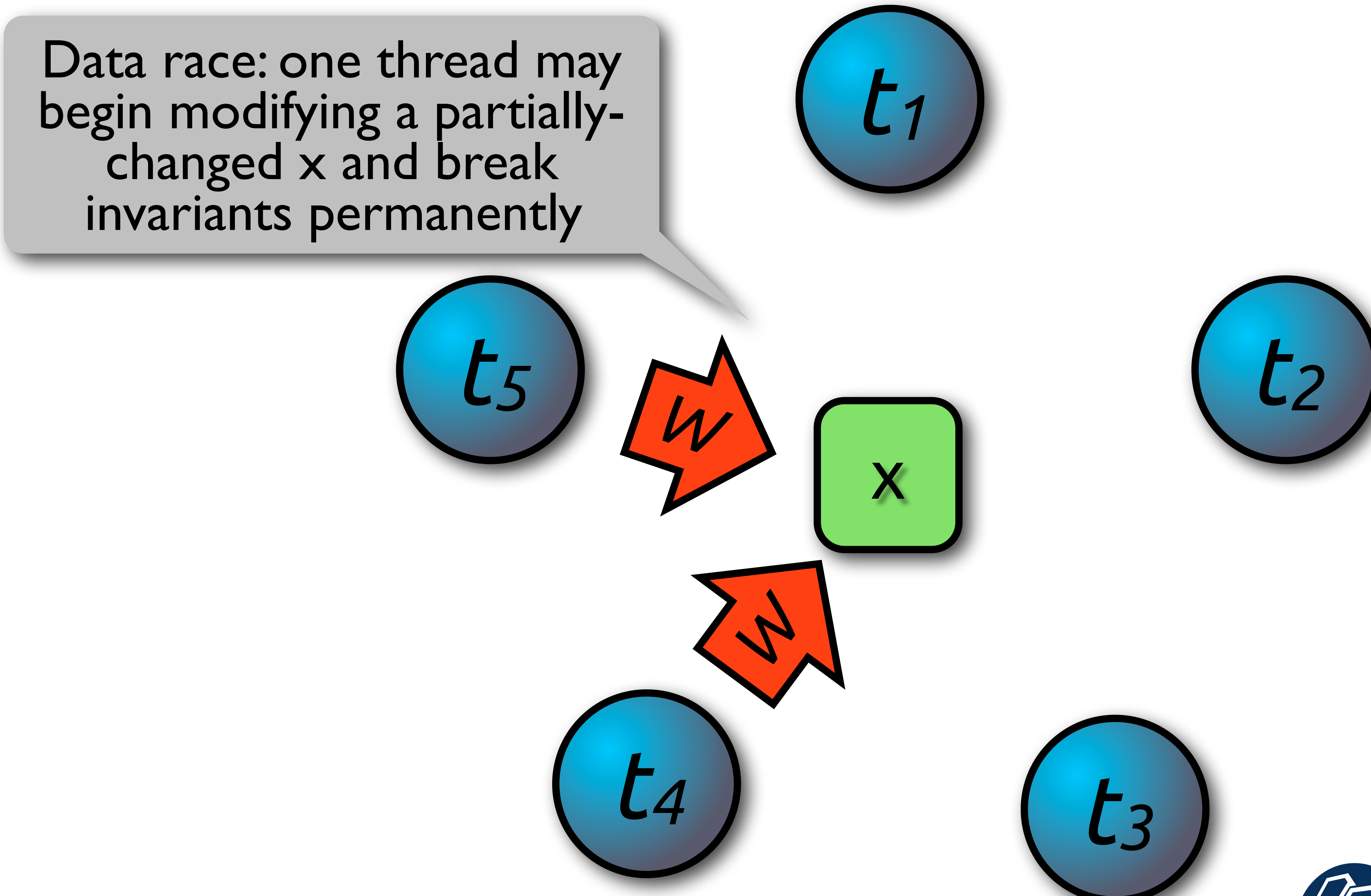


boostpro computing

Concurrent Writes



Concurrent Writes



Tuesday, May 15, 12

Thread Safety

- **Basic** thread-safety (“as threadsafe as int”)
  - an object can be read concurrently by any number of threads
  - distinct *copies* of an object can be read and written at will in distinct threads (no sharing)
  - This is the minimum required for sanity
- **Strong** thread-safety
  - A single object can be read and written at will by any number of threads
  - Shared mutable objects must be strongly threadsafe

boostpro computing

Basic Locking

- Associate a lock with some shared state *x*
- A lock can be **owned** by zero or one threads at any time
- By agreement, every thread
    - **waits** (if necessary) to **acquire** sole ownership of the lock before reading or writing *x*, and
    - **releases** ownership of the lock afterward
- *x* is no longer accessed concurrently; accesses to *x* are said to be serialized

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()                    // where m is of type L
m.unlock()
```

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

boostpro computing

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()
m.unlock()
```

this_thread acquires m, waiting if necessary

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

boostpro computing

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()                    // where m is of type L
m.unlock()
```

this_thread releases m

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

boostpro computing

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()                    // where m is of type L
m.unlock()
```

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

If m is unowned, acquire it and return true, else return false

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

boostpro computing

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()                    // where m is of type L
m.unlock()
```

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

If m can be acquired in time, do so and return true, else return false

Locking Concepts

- BasicLockable<L> *requires*

```
m.lock()                    // where m is of type L
m.unlock()
```

- Lockable<L> *requires* BasicLockable<L> and

```
bool x = m.try_lock();
```

- TimedLockable<L> *requires* Lockable<L> and

```
bool y = m.try_lock_for( duration );
bool z = m.try_lock_until( time_point );
```

boostpro computing

std::mutex

```
struct mutex {   non-copyable, non-movable
  constexpr mutex() noexcept;

  void lock();
  bool try_lock();
  void unlock();

  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

boostpro computing

std::mutex

A minimal model of Lockable<L>

```
struct mutex {   non-copyable, non-movable
  constexpr mutex() noexcept;

  void lock();
  bool try_lock();
  void unlock();

  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

std::mutex

```
struct mutex {   non-copyable, non-movable
  constexpr mutex() noexcept;

  void lock();
  bool try_lock();
  void unlock();

  typedef unspecified native_handle_type;
  native_handle_type native_handle();
};
```

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {

    bool empty() const;

    T top() const;

    void pop();

    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {

    bool empty() const;

    T top() const;

    void pop();

    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

Associate a mutex with the shared mutable state

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {

    bool empty() const;

    T top() const;

    void pop();

    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```


boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const {
        m.lock(); bool r = v.empty(); m.unlock();
        return r;
    }

    T top() const;
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

return *by value*

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

Q: what if this throws?

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

A: m remains locked(!)

boostpro computing

Strongly-Threadsafe Stack

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        m.lock(); T r = v.back(); m.unlock();
        return r;
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        std::lock_guard<std::mutex> lk(m);
        return v.back();
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

boostpro computing

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        std::lock_guard<std::mutex> lk(m);
        return v.back();
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

constructor locks m

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        std::lock_guard<std::mutex> lk(m);
        return v.back();
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

destructor unlocks it

boostpro computing

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;

    T top() const {
        std::lock_guard<std::mutex> lk(m);
        return v.back();
    }
    void pop();
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

Yes, you *can* lock a mutex directly, but that doesn't mean you *should*

boostpro computing

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;
    T top() const;

    void pop() {
        std::lock_guard<std::mutex> lk(m);
        v.pop_back();
    }
    void push(T x);
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

std::lock_guard

```
template <class T>
struct shared_stack {
    bool empty() const;
    T top() const;
    void pop();

    void push(T x) {
        std::lock_guard<mutex> lk(m);
        v.push( std::move(x) );
    }
 private:
    mutable std::mutex m;
    std::vector<T> v;
};
```

std::lock_guard

```
template <class Mutex>
struct lock_guard {   non-copyable, non-movable
  typedef Mutex mutex_type;

  explicit lock_guard(mutex_type& m);
  lock_guard(mutex_type& m, adopt_lock_t);
  ~lock_guard();

private:
  mutex_type& pm;
};
```

boostpro computing

std::lock_guard

```
template <class Mutex>
struct lock_guard {   non-copyable, non-movable
  typedef Mutex mutex_type;

  explicit lock_guard(mutex_type& m);
  lock_guard(mutex_type& m, adopt_lock_t);
  ~lock_guard();

private:
  mutex_type& pm;
};
```

take ownership of a pre-locked mutex.

Note that there's no ownership *release*

std::kitchen_sink_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

std::unique_lock

familiar from lock_guard

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();
```

post-hoc lock/unlock

```
  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool () const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```


Tuesday, May 15, 12

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

construction without ownership

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

try-locking

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();
```

timed locking

```
  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

std::unique_lock

```
template <class Mutex>
struct unique_lock {   move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

misc.

boostpro computing

std::unique_lock

```
template <class Mutex>
struct unique_lock {  move-only
  typedef Mutex mutex_type;

  explicit unique_lock(mutex_type& m);
  unique_lock(mutex_type& m, adopt_lock_t);
  ~unique_lock();

  void lock();
  void unlock();

  unique_lock() noexcept;
  unique_lock(
    mutex_type& m, defer_lock_t) noexcept;

  unique_lock(
    mutex_type& m, try_to_lock_t);
  bool try_lock();

  unique_lock(mutex_type& m, time_point );
  bool try_lock_until( time_point );

  unique_lock(mutex_type& m, duration );
  bool try_lock_for( duration );

  mutex_type* mutex() const noexcept;
  mutex_type *release() noexcept;

  bool owns_lock() const noexcept;
  explicit operator bool() const noexcept;

private:
  mutex_type *pm;
  bool owns;
};
```

boostpro computing

Message Passing

- **Goal**: use a bounded FIFO queue to pass messages between threads
- How do I avoid burning CPU:
  - looking to see if a message arrived?
  - looking to see if there's room to queue a new message?
- And how do I avoid sleeping when there's work to do?

Threadsafe Queue

```
template <unsigned size, class T>
struct bounded_msg_queue
{
    bounded_msg_queue()
      : begin(0), end(0), buffered(0) {}

    void send(T x);
    T receive();
 private:
    std::mutex broker;
    unsigned int begin, end, buffered;
    T buf[size];
    std::condition_variable not_full, not_empty;
};
```

boostpro computing

Threadsafe Queue

```
template <unsigned size, class T>
struct bounded_msg_queue
{
    bounded_msg_queue()
      : begin(0), end(0), buffered(0) {}

    void send(T x);
    T receive();
 private:
    std::mutex broker;
    unsigned int begin, end, buffered;
    T buf[size];
    std::condition_variable not_full, not_empty;
};
```

boostpro computing

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
     }
     not_empty.notify_all();
}
```

boostpro computing

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
    }
    not_empty.notify_all();
}
```

always start locked

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
     }
     not_empty.notify_all();
}
```

while condition not satisfied

boostpro computing

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
    }
    not_empty.notify_all();
}
```

wait to be notified that something changed (library unlocks for us)

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
    }
    not_empty.notify_all();
}
```

we've changed conditions for readers!

Threadsafe Queue

```
template <unsigned size, class T>
void bounded_msg_queue<size,T>::send( T x )
{
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == size)
            not_full.wait(lk);

        buf[end] = x;
        end = (end + 1) % size;
        ++buffered;
     }
     not_empty.notify_all();
}
```

boostpro computing

Threadsafe Queue

```
template <unsigned size, class T>
T bounded_msg_queue<size,T>::receive()
{
    T r;
    {
        std::unique_lock<std::mutex> lk(broker);
        while (buffered == 0)
            not_empty.wait(lk);

        r = buf[begin];
        begin = (begin + 1) % size;
        --buffered;
    }
    not_full.notify_all();
    return r;
}
```


boostpro computing

std::condition_variable

```
struct condition_variable {
  noncopyable, non-movable
  condition_variable();

  typedef unspecified native_handle_type;
  native_handle_type native_handle();

  void notify_one() noexcept;
  void notify_all() noexcept;

  void wait(
    unique_lock<mutex>& lock);

  template <class Predicate>
  void wait(
    unique_lock<mutex>& lock,
    Predicate p);

  cv_status wait_until(
    unique_lock<mutex>& lock,
    time_point);

  template <class Predicate>
  bool wait_until(
      unique_lock<mutex>& lock,
      time_point, Predicate pred);

  cv_status wait_for(
    unique_lock<mutex>& lock,
    duration );

  template <class Predicate>
  bool wait_for(
    unique_lock<mutex>& lock,
    duration, Predicate);
};
```

boostpro computing

`boost::shared_mutex`

- Not in the standard library.

  See http://boost.org/libs/thread
- Can be acquired by multiple readers *or* one writer
- Great for often-read, seldom-written data

boostpro computing

Go forth and mutate!
(safely)