Linden Lab

Elegant Asynchronous Code

Nat Goodspeed

CppCon 2016

Program Organization

- How do you design a nontrivial program?
- Allocation of responsibility
- Abstraction layers
- Encapsulation

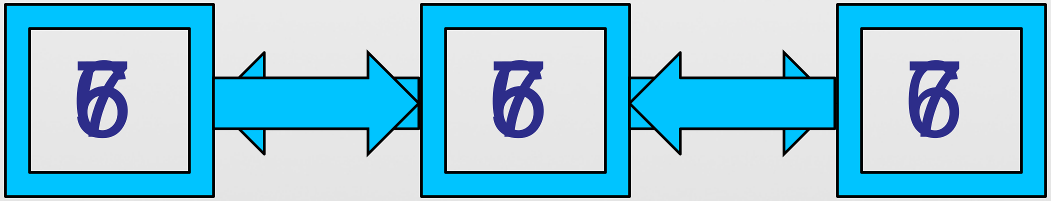
These Are Not Your Mom’s Applications

- Single task is great for classic console application
- batch processing
- unit tests
- etc.
- Event-driven GUI app cannot use classic procedural organization
- Relatively few useful applications are purely local any more

Threads

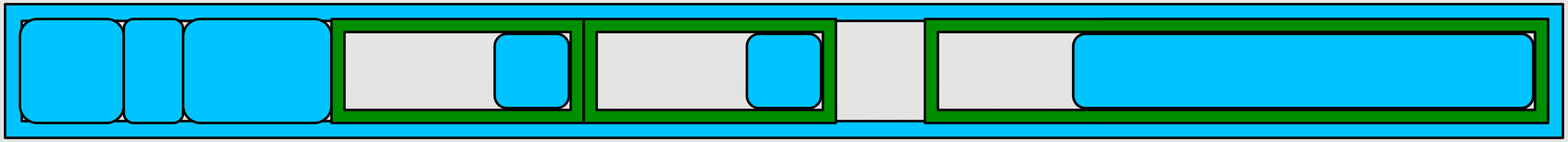
- Pro:
  - Decent language support
  - Normal blocking I/O “just works”

- Con:
  - Data races!



Threads

- **Pro:**
  - Decent language support
  - Normal blocking I/O "just works"
- **Con:**
  - Data races!
  - Synchronization overhead
  - Context switching
  - Stack size



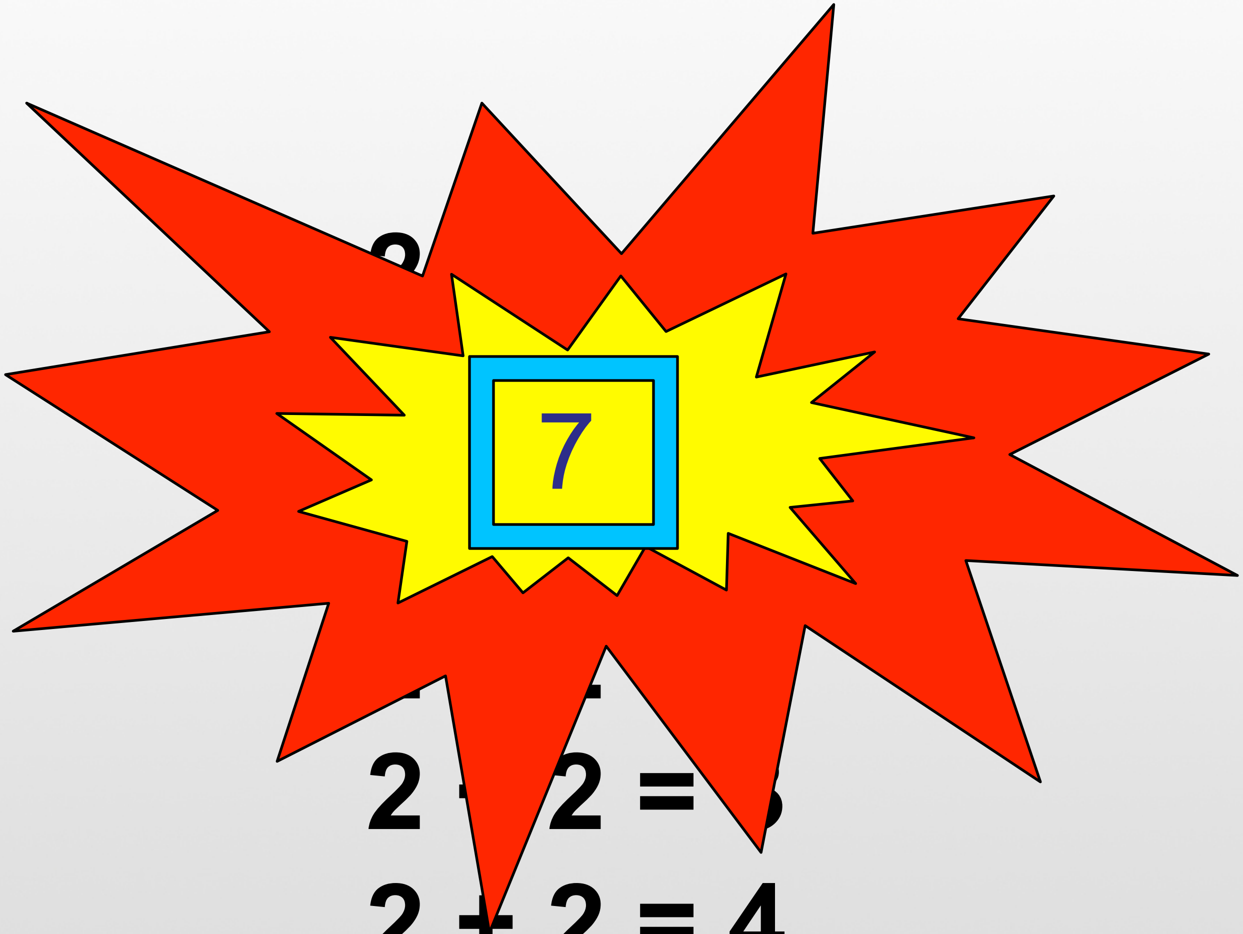
The Cost of Locking

- Runtime cost
  - Kernel entry, context switch
  - Thread makes no progress while blocked by other thread
  - Context switch

- Development cost
  - Developer chops
  - Reviewer chops
  - Detection

The Cost of *Not* Locking

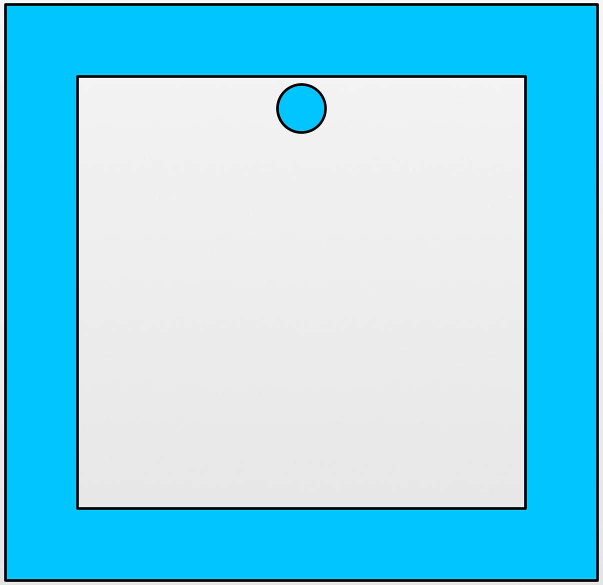
- Undefined Behavior



2 + 2 = 4

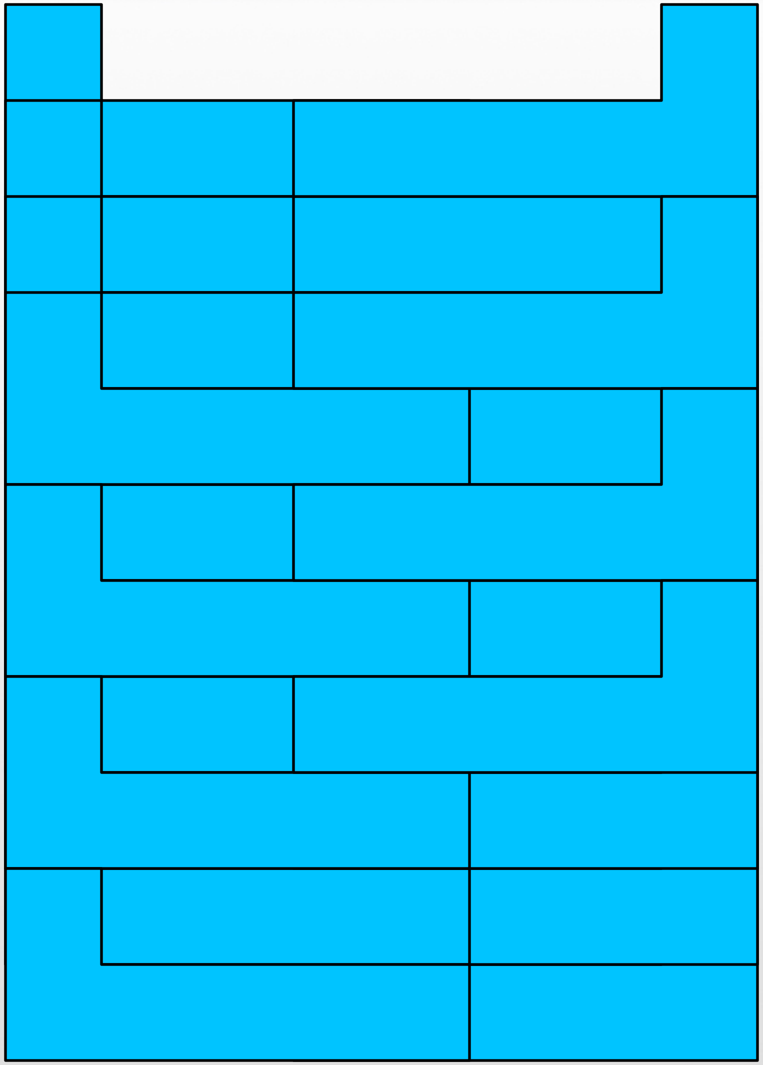
Tooling?

- Helgrind
- DRD
- ThreadSanitizer



Tooling?

- Helgrind
- DRD
- ThreadSanitizer



Synchronous I/O

●

●

●

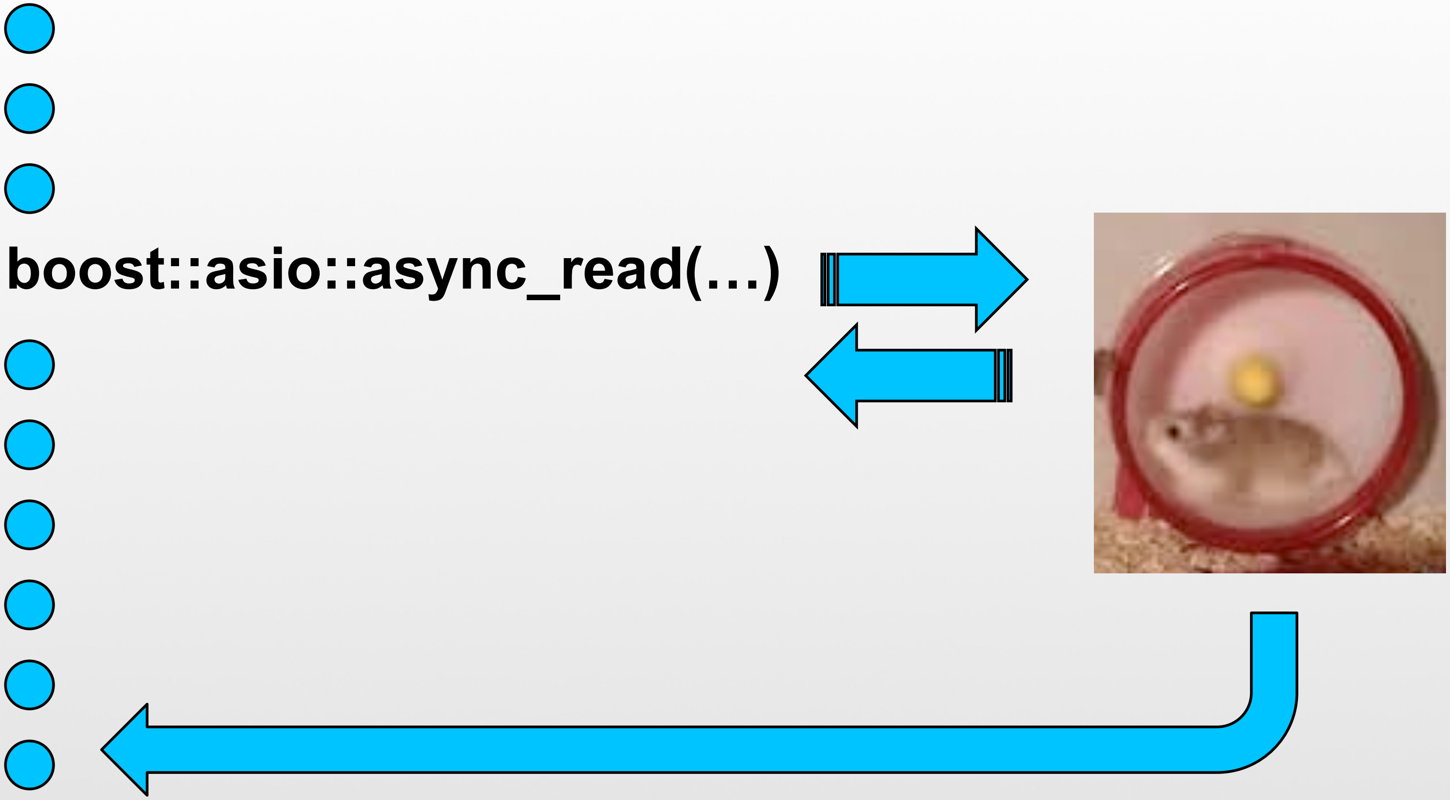
boost::asio::read(…)

●

●

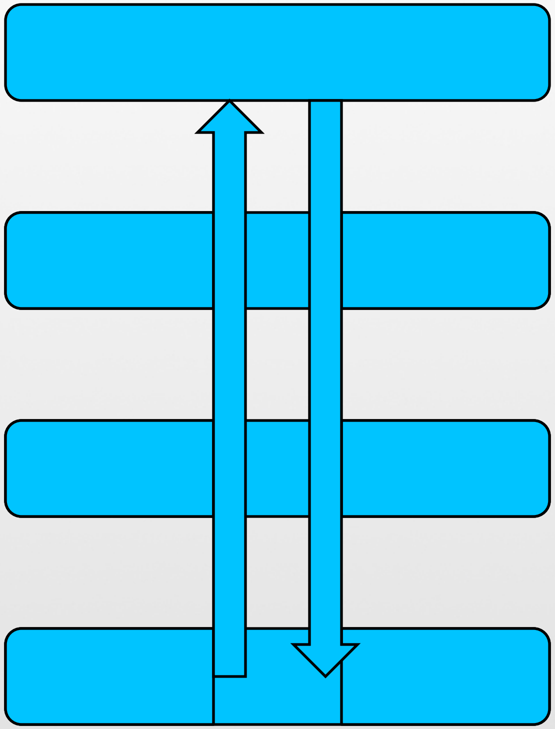
●

Asynchronous I/O



boost::asio::async_read(…)

Async hole



Async lifelines

- Chains of callbacks
- Chains of then()
- State machines
- Big switch statement
- Boost.Asio coroutines
- Visual Studio 2015 resumable functions
- Boost.Coroutine

Boost.Fiber

Fibers are the best way I know to organize code already based on async I/O.

GET WITH
THE PROGRAM
DUDE

Boost.Fiber

Fibers are the best way I know to organize code already based on async I/O.

What are Fibers?

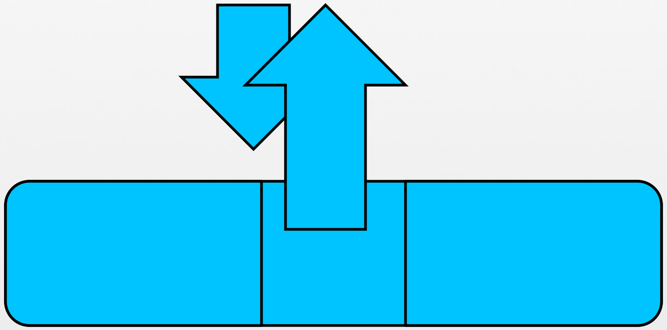
- “userland threads”
- Cooperative suspension and resumption
- On a given thread, at most one fiber is running at a time
- Suspend-by-call: suspension can be transparent to caller
- Independent stacks
- Semi-independent

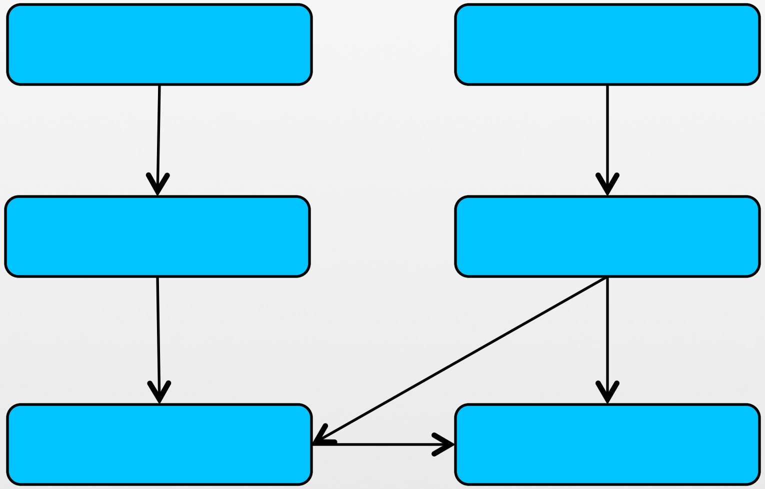
What about stacks?

- Custom stack size
- Custom stack allocator
- Alternative stack implementations

What about stackless?

- Go for it!
- No solution visible yet





What about stackless?

```
std::future<int> theAnswer() {
    // … some time-consuming computation …
    return 42;
}

void askTheQuestion() {
    int tada = co_await theAnswer();
}

…
askTheQuestion();
```

What about stackless?

```
std::future<int> theAnswer() {
    // … some computation involving suspension …
    return 42;
}

void askTheQuestion() {
    int tada = co_await theAnswer();
}

…
askTheQuestion();
```

What about stackless?

```
std::future<int> theAnswer() {
    // … some time-consuming computation …
    return 42;
}

std::future<void> askTheQuestion() {
    int tada = co_await theAnswer();
}

…
askTheQuestion();
```

What about stackless?

```
std::future<int> theAnswer() {
    // … some time-consuming computation …
    return 42;
}

std::future<void> askTheQuestion() {
    int tada = co_await theAnswer();
}

…
co_await askTheQuestion();
```

Stacks for the win

- Transparent suspension: suspend-by-call, suspend down
- Local variables Just Work

A passing glance at the Fiber API

- fiber-local storage
- join()
- detach()
- yield()
- sleep_for(), sleep_until()
- mutex
- condition_variable
- barrier
- promise, future, packaged_task
- async()
- unbounded_channel, bounded_channel

Fibers and Asynchronous Callbacks

```
class AsyncAPI {
public:
    // constructor acquires some resource that can be read
    AsyncAPI();

    // callbacks accept an int error code; 0 == success
    typedef int errorcode;

    // read callback needs to accept both errorcode and data:
    // void callback(errorcode ec, std::string data);
    template< typename Fn >
    void init_read( Fn && callback);
};
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                        ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                          if ( ! ec) {
                              promise.set_value( data);
                          } else {
                              promise.set_exception(
                                      std::make_exception_ptr(
                                          make_exception("read", ec) ) );
                          }
                  });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                    ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                     if ( ! ec) {
                         promise.set_value( data);
                     } else {
                         promise.set_exception(
                                 std::make_exception_ptr(
                                     make_exception("read", ec) ) );
                     }
              });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                        ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                          if ( ! ec) {
                              promise.set_value( data);
                          } else {
                              promise.set_exception(
                                      std::make_exception_ptr(
                                          make_exception("read", ec) ) );
                          }
                 });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                     ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                       if ( ! ec) {
                           promise.set_value( data);
                       } else {
                           promise.set_exception(
                                   std::make_exception_ptr(
                                       make_exception("read", ec) ) );
                       }
                  });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
  boost::fibers::promise< std::string > promise;
  boost::fibers::future< std::string > future( promise.get_future() );
  api.init_read([promise=std::move( promise)]
                  ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                    if ( ! ec) {
                      promise.set_value( data);
                    } else {
                      promise.set_exception(
                          std::make_exception_ptr(
                            make_exception("read", ec) ) );
                    }
              });
  return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                    ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                     if ( ! ec) {
                         promise.set_value( data);
                     } else {
                         promise.set_exception(
                                 std::make_exception_ptr(
                                     make_exception("read", ec) ) );
                     }
                 });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                         ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                           if ( ! ec) {
                               promise.set_value( data);
                           } else {
                               promise.set_exception(
                                       std::make_exception_ptr(
                                           make_exception("read", ec) ) );
                           }
                  });
    return future.get();
}
```

Linden Lab®

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                        ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                          if ( ! ec) {
                              promise.set_value( data);
                          } else {
                              promise.set_exception(
                                      std::make_exception_ptr(
                                          make_exception("read", ec) ) );
                          }
                  });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                        ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                          if ( ! ec) {
                              promise.set_value( data);
                          } else {
                              promise.set_exception(
                                      std::make_exception_ptr(
                                          make_exception("read", ec) ) );
                          }
                  });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
std::string read( AsyncAPI & api) {
    boost::fibers::promise< std::string > promise;
    boost::fibers::future< std::string > future( promise.get_future() );
    api.init_read([promise=std::move( promise)]
                        ( AsyncAPI::errorcode ec, std::string const& data) mutable {
                          if ( ! ec) {
                              promise.set_value( data);
                          } else {
                              promise.set_exception(
                                      std::make_exception_ptr(
                                          make_exception("read", ec) ) );
                          }
                  });
    return future.get();
}
```

Fibers and Asynchronous Callbacks

```
AsyncAPI myAsyncInstance(…);
// …
std::string result = read(myAsyncInstance);
```

Linden Lab®

Fibers and Nonblocking I/O

```
class NonblockingAPI {
public:
    NonblockingAPI();

    // nonblocking operation: may return EWOULDBLOCK
    // may return size < desired
    int read( std::string & data, std::size_t desired);
};
```

Fibers and Nonblocking I/O

```
// guaranteed not to return EWOULDBLOCK
int read_chunk( NonblockingAPI & api, std::string & data,
                std::size_t desired) {
    int error;
    while ( EWOULDBLOCK ==
                ( error = api.read( data, desired) ) ) {
        // not ready yet – run other fibers and then try again
        boost::this_fiber::yield();
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
// guaranteed not to return EWOULDBLOCK
int read_chunk( NonblockingAPI & api, std::string & data,
                std::size_t desired) {
    int error;
    while ( EWOULDBLOCK ==
            ( error = api.read( data, desired) ) ) {
        // not ready yet – run other fibers and then try again
        boost::this_fiber::yield();
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
// guaranteed not to return EWOULDBLOCK
int read_chunk( NonblockingAPI & api, std::string & data,
                                std::size_t desired) {
    int error;
    while ( EWOULDBLOCK ==
                ( error = api.read( data, desired) ) ) {
        // not ready yet – run other fibers and then try again
        boost::this_fiber::yield();
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
// guaranteed not to return EWOULDBLOCK
int read_chunk( NonblockingAPI & api, std::string & data,
                                std::size_t desired) {
    int error;
    while ( EWOULDBLOCK ==
                ( error = api.read( data, desired) ) ) {
        // not ready yet – run other fibers and then try again
        boost::this_fiber::yield();
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
// guaranteed not to return EWOULDBLOCK
int read_chunk( NonblockingAPI & api, std::string & data,
                          std::size_t desired) {
    int error;
    while ( EWOULDBLOCK ==
                ( error = api.read( data, desired) ) ) {
        // not ready yet – run other fibers and then try again
        boost::this_fiber::yield();
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                  std::size_t desired) {
    // we're going to accumulate results into 'data'
    data.clear();
    std::string chunk;
    int error = 0;
    while ( data.length() < desired &&
            ! (error = read_chunk(api, chunk, desired - data.length())))
    {
        data.append( chunk);
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                             std::size_t desired) {
    // we're going to accumulate results into 'data'
    data.clear();
    std::string chunk;
    int error = 0;
    while ( data.length() < desired &&
            ! (error = read_chunk(api, chunk, desired - data.length())))
    {
        data.append( chunk);
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                  std::size_t desired) {
   // we're going to accumulate results into 'data'
   data.clear();
   std::string chunk;
   int error = 0;
   while ( data.length() < desired &&
           ! (error = read_chunk(api, chunk, desired-data.length())))
   {
      data.append( chunk);
   }
   return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                  std::size_t desired) {
    // we're going to accumulate results into 'data'
    data.clear();
    std::string chunk;
    int error = 0;
    while ( data.length() < desired &&
            ! (error = read_chunk(api, chunk, desired-data.length())))
    {
        data.append( chunk);
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                  std::size_t desired) {
  // we're going to accumulate results into 'data'
  data.clear();
  std::string chunk;
  int error = 0;
  while ( data.length() < desired &&
      ! (error = read_chunk(api, chunk, desired - data.length())))
  {
    data.append( chunk);
  }
  return error;
}
```

Fibers and Nonblocking I/O

```
int read_desired( NonblockingAPI & api, std::string & data,
                              std::size_t desired) {
    // we're going to accumulate results into 'data'
    data.clear();
    std::string chunk;
    int error = 0;
    while ( data.length() < desired &&
           ! (error = read_chunk(api, chunk, desired - data.length())))
    {
        data.append( chunk);
    }
    return error;
}
```

Fibers and Nonblocking I/O

```
std::string read( NonblockingAPI & api, std::size_t desired) {
    std::string data;
    int ec( read_desired( api, data, desired) );
    // for present purposes, EOF isn't a failure
    if ( 0 == ec || EOF == ec) {
        return data;
    }
    // oh oh, partial read
    std::ostringstream msg;
    msg << "NonblockingAPI::read() error " << ec << " after "
        << data.length() << " of " << desired << " characters";
    throw std::runtime_error( msg.str());
}
```

when_any()

wait_any()

```
std::string query_one();
std::string query_two();
std::string query_three();
auto queries = { query_one, query_two, query_three };
…
std::string result = wait_any<std::string>(queries);
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

Linden Lab®

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    typedef boost::fibers::future< return_t > future_t;
    typedef boost::fibers::unbounded_channel< future_t > channel_t;
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    for ( auto function : container ) {
        boost::fibers::fiber([function, channelp]() {
            boost::fibers::packaged_task< return_t() > task( function);
            task();
            channelp->push( task.get_future() );
        }).detach();
        ++count;
    }
    // …
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

wait_any()

```
template < typename return_t, typename Container >
return_t wait_any(Container const& container) {
    auto channelp( std::make_shared< channel_t >() );
    std::size_t count = 0;
    // …
    for ( std::size_t i = 0; i < count; ++i) {
        future_t future( channelp->value_pop() );
        std::exception_ptr error( future.get_exception_ptr() );
        if ( ! error) {
            channelp->close();
            return future.get();
        }
    }
    throw std::runtime_error("ashes, ashes, we all fell down");
}
```

~~when~~ wait_all()

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all( Fns && ... functions);
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all( Fns && ... functions);

struct Data {
    std::string str;
    double inexact;
    int exact;
};

auto data = wait_all<Data>(strfunc, doublefunc, intfunc);
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all_members( Fns && ... functions) {
    return wait_all_get< Result >(
            boost::fibers::async( std::forward< Fns >( functions) ) ... );
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all_members( Fns && ... functions) {
    return wait_all_get< Result >(
            boost::fibers::async( std::forward< Fns >( functions) ) ... );
}

template< typename Result, typename ... Futures >
Result wait_all_get( Futures && ... futures) {
    return Result{ futures.get() ... };
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all( Fns && ... functions) {
    return Result{ boost::fibers::async(functions).get()... };
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all( Fns && ... functions) {
    return Result{ functions()... };
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all_members( Fns && ... functions) {
    return wait_all_get< Result >(
            boost::fibers::async( std::forward< Fns >( functions) ) ... );
}

template< typename Result, typename ... Futures >
Result wait_all_get( Futures && ... futures) {
    return Result{ futures.get() ... };
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all_members( Fns && ... functions) {
    return wait_all_get< Result >(
        boost::fibers::async(std::forward<Fns>(functions) ) ... );
}

template< typename Result, typename ... Futures >
Result wait_all_get( Futures && ... futures) {
    return Result{ futures.get() ... };
}
```

wait_all()

```
template< typename Result, typename ... Fns >
Result wait_all_members( Fns && ... functions) {
    return wait_all_get< Result >(
            boost::fibers::async( std::forward< Fns >( functions) ) ... );
}

template< typename Result, typename ... Futures >
Result wait_all_get( Futures && ... futures) {
    return Result{ futures.get() ... };
}
```

Integrating with an Event Loop

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);

}
```

Integrating with an Event Loop

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    boost::this_fiber::yield();
}
```

Integrating with Another Framework

```
int main(int argc, char* argv[]) {
    boost::asio::io_service io_svc;
    // … setup …
    io_svc.run();
    // control does not return from the run() call until done
}
```

Integrating with Another Framework

```
void yielder(boost::asio::io_service& io_svc) {
    boost::this_fiber::yield();
    io_svc.post([&io_svc](){ yielder(io_svc); });
}

    // … setup …
    io_svc.post([&io_svc](){ yielder(io_svc); });
```

Integrating with Another Framework

```
void yielder(boost::asio::steady_timer& timer) {
    boost::this_fiber::yield();
    timer.expires_from_now(std::chrono::milliseconds(10));
    timer.async_wait([&timer]
                     (boost::system::error_code)
                     { yielder(timer); });
}

    // … setup …
    boost::asio::steady_timer timer(io_svc);
    io_svc.post([&timer](){ yielder(timer); });
```

Customizing the Fiber Scheduler

```
struct sched_algorithm {
    virtual void awakened( context *) noexcept = 0;
    virtual context * pick_next() noexcept = 0;
    virtual bool has_ready_fibers() const noexcept = 0;

    virtual void suspend_until(
        std::chrono::steady_clock::time_point const&) noexcept=0;
    virtual void notify() noexcept = 0;
};

boost::fibers::use_scheduling_algorithm< your_subclass >();
```

References

- **Fiber will be part of Boost 1.62**

Boost 1.62 beta 2 download:
https://sourceforge.net/projects/boost/files/boost/1.62.0.beta.2/

- **Questions?**

Performance

Haskell | stack-1.0.4	fiber (single threaded/ raw) | gcc-5.2.1	fiber (single threaded/ atomics) | gcc-5.2.1	Erlang | erts-7.0	Go | go1.4.2
58ms - 108ms	205ms - 263ms	221ms - 278ms	237ms-470ms	614ms - 883ms

skynet with N=100000 actors/goroutines/fibers