Executors and Asynchronous Operations

http://chriskohlhoff.github.io/executors/

Using the executors library: a two minute introduction

Run a function asynchronously.

```
#include <experimental/executor>

using std::experimental::post;

int main()
{
  post([]{
      // ...
    });
}
```

Run a function asynchronously on your own thread pool.

```
#include <experimental/executor>
#include <experimental/thread_pool>

using std::experimental::post;
using std::experimental::thread_pool;

int main()
{
  thread_pool pool;

  post(pool, []{
      // ...
    });

  pool.join();
}
```

Run a function asynchronously. Wait for the result.

```
#include <experimental/executor>
#include <experimental/future>
#include <iostream>

using std::experimental::post;
using std::experimental::package;

int main()
{
  std::future<int> f =
    post(package([]{
      // ...
      return 42;
    }));

  std::cout << f.get() << std::endl;
}
```

Run a function asynchronously on your own thread pool. Wait for the result.

```
#include <experimental/executor>
#include <experimental/future>
#include <experimental/thread_pool>
#include <iostream>

using std::experimental::post;
using std::experimental::package;
using std::experimental::thread_pool;

int main()
{
  thread_pool pool;

  std::future<int> f =
    post(pool, package([]{
      // ...
      return 42;
    }));

  std::cout << f.get() << std::endl;
}
```

+

Run a function in the future.
Wait for the result.

```
#include <experimental/executor>
#include <experimental/future>
#include <experimental/timer>
#include <iostream>

using std::experimental::post_after;
using std::experimental::package;


int main()
{
  std::future<int> f =
    post_after(
      std::chrono::seconds(1),
      package([]{
        // ...
        return 42;
      }));

  std::cout << f.get() << std::endl;
}
```

The vocabulary

Executor

Executors are to function execution as allocators are to memory allocation

- An executor is a set of rules governing where, when and how to run a function object.
- Like allocators, executors are lightweight and cheap to copy.
- Examples:
  - The system executor
  - A strand

Execution context

- An execution context is a place where function objects are executed.
- Examples:
  - A fixed-size thread pool
  - A loop scheduler
  - An asio::io_service
  - The set of all threads in the process

Example: a thread pool

- A thread pool *is* an execution context.
- A thread pool *has* an executor.
- A thread pool's executor embodies this rule:

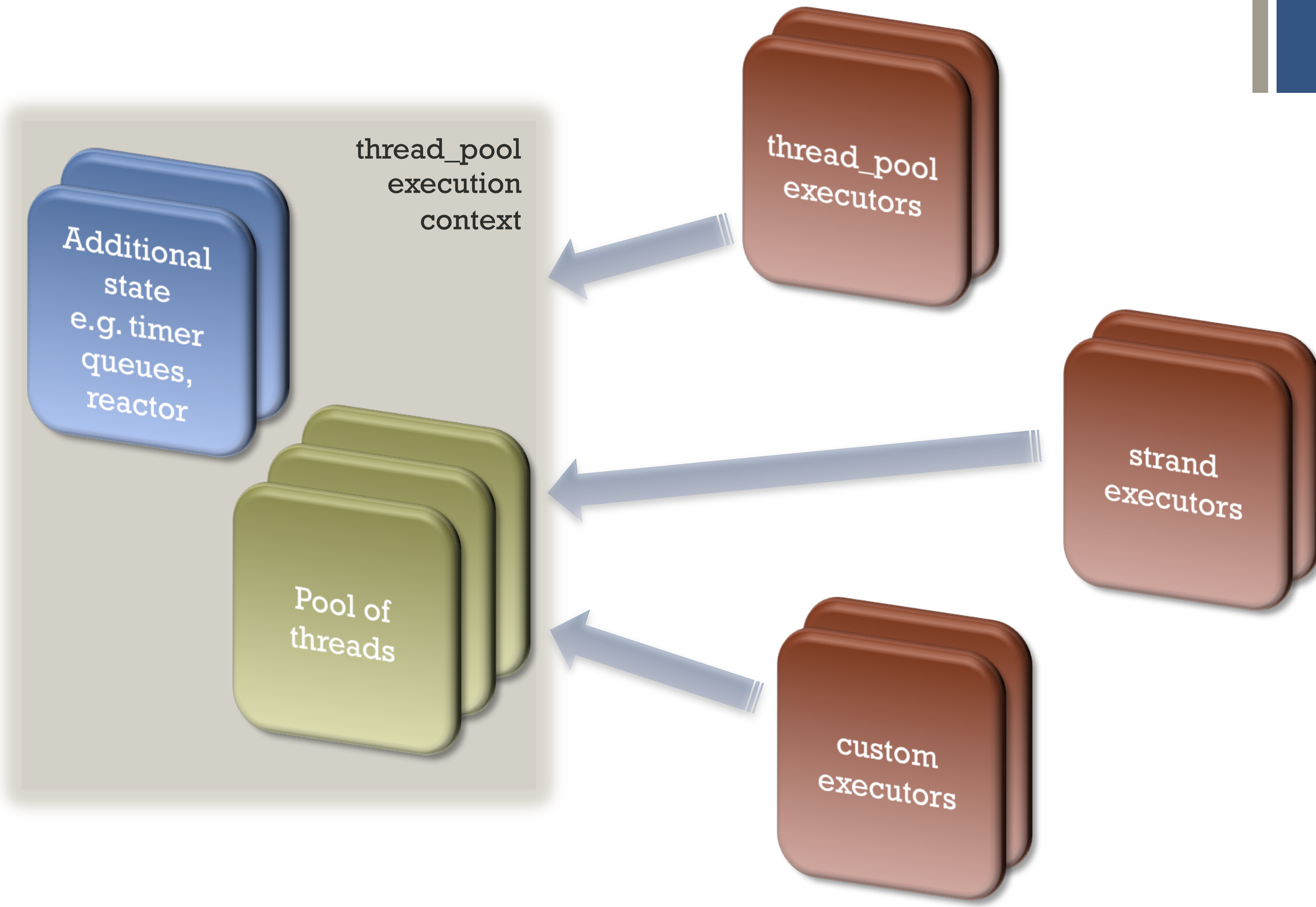
  Run function objects *in the pool and nowhere else*.

Example: a strand

- A strand *is* an executor.
- A strand is an adapter for an underlying executor.
- A strand embodies this rule:

  Run function objects according to the underlying executor's rules, but also run them *in FIFO order and not concurrently.*

Execution contexts and executors



Execution contexts and executors

Execution contexts	Executors
Usually long lived.	May be long or short lived.
Non-copyable.	Lightweight and copyable.
May contain additional state. - Timer queues. - Socket reactors. - Hidden threads to emulate asynchronous functionality.	May be customized on a fine-grained basis. - Example: an executor to capture exceptions generated by an asynchronous operation into an exception_ptr.

Dispatch, post and defer

- The three fundamental operations for submitting function objects for execution.
- They differ in the level of eagerness to execute a function.
- May be used to submit function objects to an executor or an execution context.

Dispatch

- Run the function object immediately if the rules allow it.
- Otherwise, submit for later execution.
- Example: a thread pool
  - Rule: run function objects *in the pool and nowhere else*.
  - If we are on a thread in the pool, run the function object immediately.
  - If we are *not* on a thread in the pool, queue the function object for later and wake up a thread to process it.

+ Post

- Submit the function for later execution.
- Never run the function object immediately.
- Example: a thread pool
  - Whether or not we are on a thread in the pool, queue the function object for later and wake up a thread to process it.

+ Defer

- Submit the function for later execution.
- Never run the function immediately.
- Implies a continuation relationship between caller and function object.
- Example: a thread pool
  - If we are *not* on a thread in the pool, queue the function object for later and wake up a thread to process it.
  - If we are on a thread in the pool, queue the function object for later, but don't wake up a thread to process it until control returns to the pool.

Use case #1:
replacing std::async

A replacement for std::async

- With std::async we can submit a function object that runs in a different thread.

```
std::future<int> f = std::async([]{
    // ...
    return 42;
  }));

int i = f.get();
```

- The equivalent is to post a packaged task.

```
std::future<int> f = post(
  package([]{
      // ...
      return 42;
    }));
```

Collecting the function result

- We can package and post a function with any return type.

```
std::future<std::string> f = std::async([]{
    // ...
    return "hello"s;
  }));

std::string s = f.get();
```

- This includes functions that just return void.

```
std::future<void> f = post(
  package([]{
      // ...
    }));

f.get();
```

+ Using packaged_task

- The package function creates a packaged_task.
- We can also post a packaged_task directly. If so, we must explicitly specify the call signature.

```
std::future<int> f = post(
  std::packaged_task<int()>(
    []{
      // ...
      return 42;
    }));

int i = f.get();
```

Using function objects

- Any 0-argument function object can be submitted using post.

- Example: lambdas

```
post([]{
    // ...
  });
```

- Example: functions

```
void do_something();

post(&do_something);
```

Using function objects

- Example: function object binders

```
post(std::bind(&my_class::my_function, this));
```

- Example: hand-rolled function objects

```
struct my_function {
  void operator()() {
    // ...
  }
};

post(my_function());
```

The system executor

- By default, the post function submits function objects to the system executor.
- The system executor represents the set of all threads in the process.
- The system executor embodies this rule:

Function objects are allowed to run on *any thread in the system.*

- Like std::async, the system executor can automatically allocate threads to run function objects that are submitted to it.

Using a thread pool

- Unlike std::async, with post we can specify that the function object be run on a particular executor or execution context.

```
thread_pool pool;

std::future<int> f = post(pool,
  package([]{
      // ...
      return 42;
    }));
```

- If the thread pool is stopped, any queued function objects will be abandoned.

```
pool.stop();
pool.join();
```

Use case #2: active objects

Active objects

- In the Active Object design pattern, all operations associated with an object are run in its own private thread.

- To implement an active object, use a class member that is a thread pool containing a single thread.

```
class bank_account {
  int balance_ = 0;
  thread_pool pool_{1};
  // ...
};
```

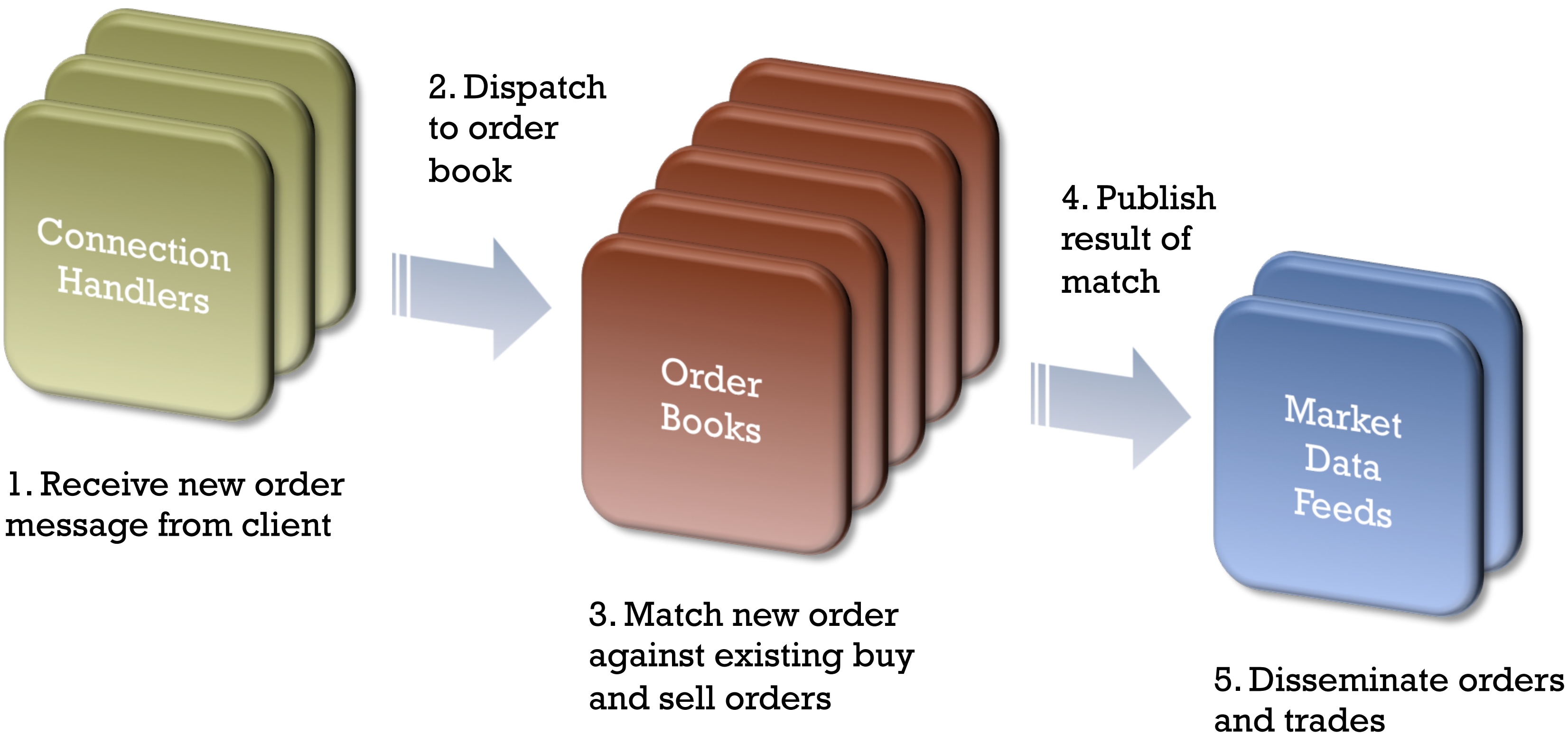
+ Active object operations

- An active object operation involves three steps.
  - Package the body of the operation .
  - Post the package to the thread pool.
  - Use a future to wait for the operation to complete.

```
class bank_account {
  // ...
  void deposit(int amount) {
    post(pool_,
      package([]{
        balance_ += amount;
      })).get();
  }
};
```

Use case #3: parallelism in application data flow

Design of a simple trading system



1. Connection handler

- A connection handler is responsible for receiving messages from a client.
- Uses a thread pool to implement the Leader/Followers design pattern.
  - A leader thread waits for the next message.
  - A new message arrives. The leader thread promotes a follower to become the new leader.
  - The former leader processes the message.
  - The former leader returns to the pool as a follower thread.

1. Connection handler

- Leader/Followers implementation:

```
void connection_handler::receive_and_dispatch()
{
  // Wait until a new message is received.
  char buffer[1024];
  std::size_t length = socket_.receive(buffer, sizeof(buffer));

  // Wake another thread to wait for new messages.
  std::experimental::post(thread_pool_,
      [this]{ receive_and_dispatch(); });

  // Process the new message and pass it to the order management bus.
  std::istringstream is(std::string(buffer, length));
  order_management::new_order event;
  if (is >> event)
    order_management_bus_.dispatch_event(event);
}
```

2. Order management bus

- Passes new messages to the appropriate order book.
- Order books are subscribed to the bus only during program start. No synchronization is required to dispatch an event.

```
void order_management_bus::dispatch_event(
    order_management::new_order o)
{
  auto iter = books_.find(o.symbol);
  if (iter != books_.end())
    iter->second->handle_event(o);
}
```

3. Order book

- An order book maintains the open buy and sell orders for a given stock, such as GOOG or MSFT.
- An incoming order triggers a search for matching orders.
- For each matching order found, the order book creates one or more trades.
- Any left over quantity on the incoming order is added to the book.

3. Order book

- New orders must be processed atomically and in FIFO order.
- To meet these requirements, we combine three components:
  - The system executor
  - A strand
  - The dispatch function

```
class price_time_order_book : public order_book
{
  std::experimental::strand<std::experimental::system_executor> strand_;
  // ...
};

void price_time_order_book::handle_event(order_management::new_order o)
{
  std::experimental::dispatch(strand_, [=]{ process_new_order(o); });
}
```

3. Order book

- A system_executor embodies this rule:

Function objects are allowed to run on *any thread in the system.*

- A strand embodies this rule:

Run function objects according to the underlying executor's rules, but also run them *in FIFO order and not concurrently.*

- The dispatch function says:
  - Run the function object immediately *if the rules allow it*.
  - Otherwise, submit for later execution.

3. Order book

- Thus, the combination of system_executor, strand and dispatch…

```
std::experimental::dispatch(strand_, [=]{ process_new_order(o); });
```

- means:

If the strand is not busy, run process_new_order immediately.

- If there is no contention on the strand, latency is minimized.
- If there is contention, the strand in any case ensures that process_new_order is never run concurrently.
- Distinct order books can still process orders in parallel.

4. Market data bus

- Passes the result of a match to the market data feeds for dissemination.
- Feeds are subscribed to the bus only during program start. No synchronization is required to dispatch an event.

```
void market_data_bus::dispatch_event(market_data::new_order o)
{
  for (auto& f: feeds_)
    f->handle_event(o);
}

void market_data_bus::dispatch_event(market_data::trade t)
{
  for (auto& f: feeds_)
    f->handle_event(t);
}
```

5. Market data feed

- Sends messages to subscribers, e.g. using UDP multicast.
- Messages must be processed atomically and in FIFO order.
- Uses system_executor, strand and dispatch.

```
void market_by_order::handle_event(market_data::new_order o)
{
  std::experimental::dispatch(strand_,
      [=]() mutable
      {
        o.sequence_number = next_sequence_number_++;
        std::ostringstream os;
        os << o;
        std::string msg = os.str();
        socket_.send(msg.data(), msg.length());
      });
}
```

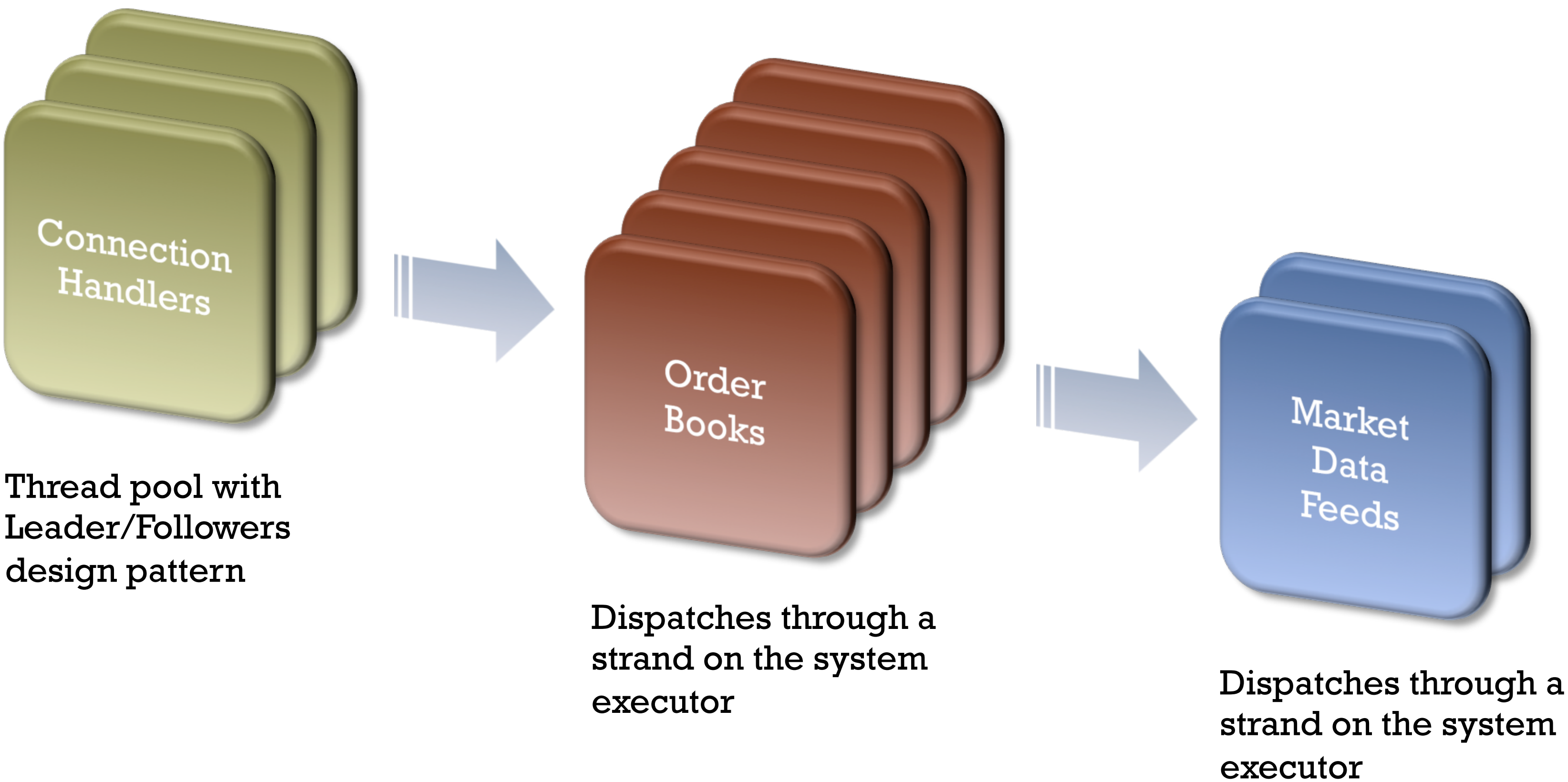
5. Market data feed

- Sends a heartbeat once a second.

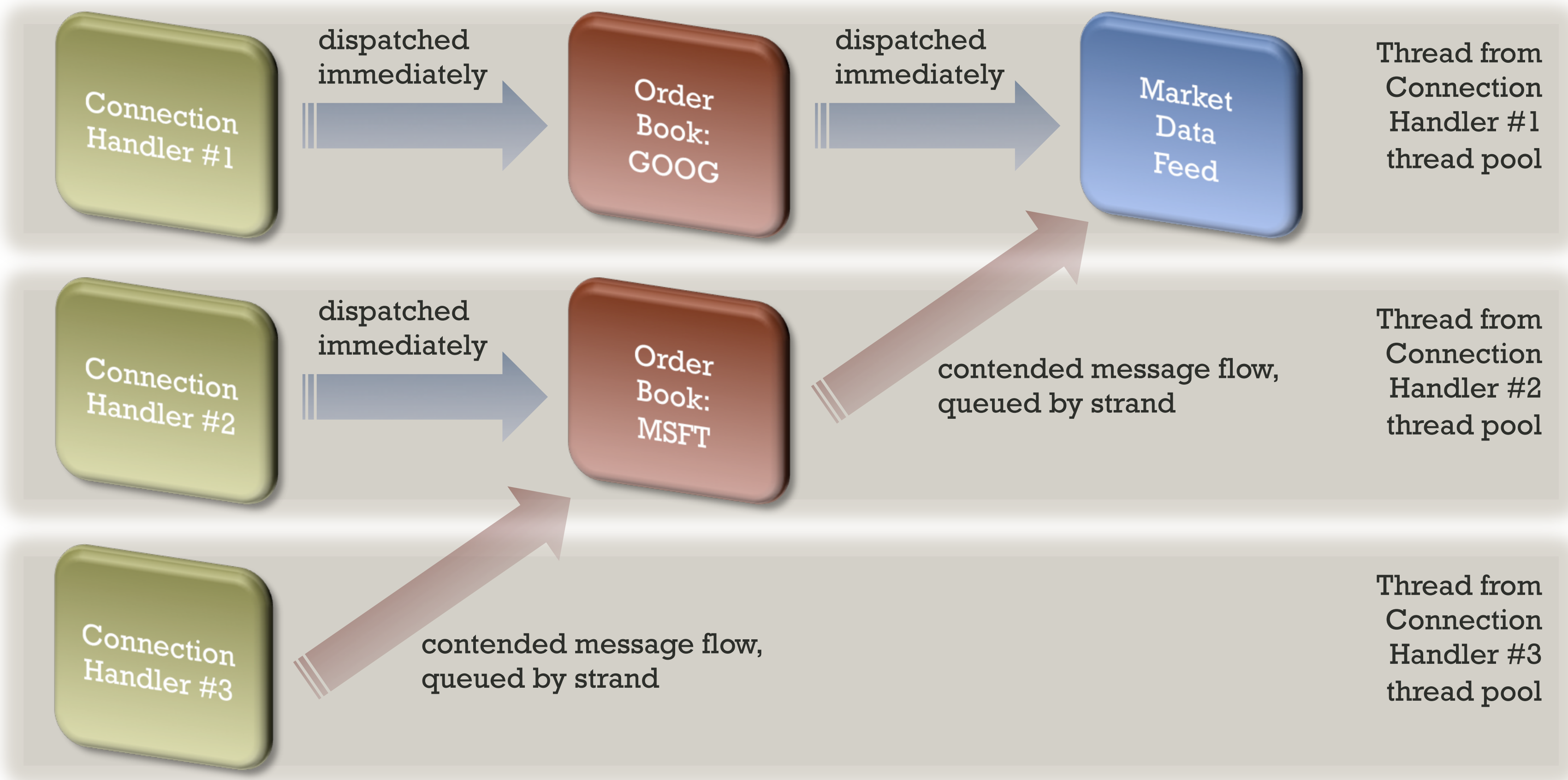
```
void market_by_order::send_heartbeat()
{
  market_data::heartbeat h;
  h.sequence_number = next_sequence_number_;
  h.time = std::time(nullptr);

  std::ostringstream os;
  os << h;
  std::string msg = os.str();

  socket_.send(msg.data(), msg.length());

  std::experimental::defer_after(std::chrono::seconds(1),
      strand_, [this]{ send_heartbeat(); });
}
```

- Uses a defer operation since the submitted function object represents a continuation of the caller.

Trading system design summary



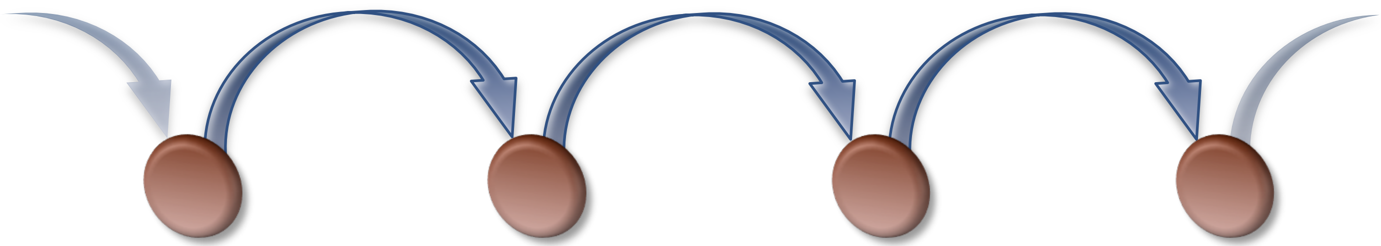
Example: flow of three simultaneously arriving orders



Use case #4: asynchronous operations

Chains of asynchronous operations

- Asynchronous operations are often chained.



```
void connection::do_read()
{
  socket_.async_read_some(in_buffer_,
    [this](error_code ec, size_t n)
    {
      // ... process input data ...
      if (!ec) do_read();
    });
}
```

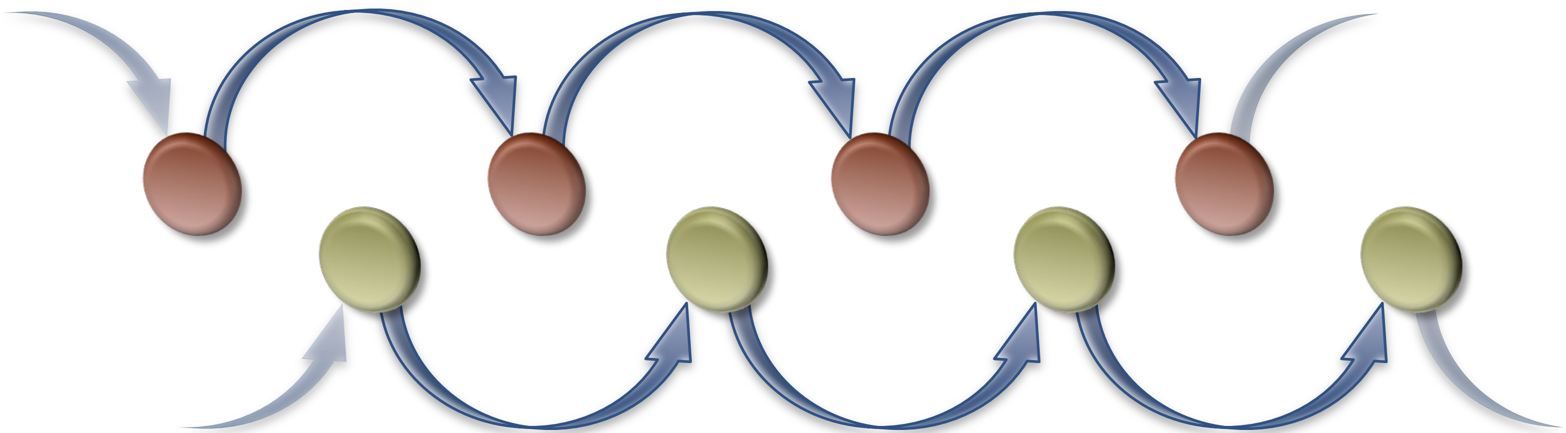
Chains of asynchronous operations

- And in many cases an object will have more than one chain.

```
void connection::do_write()
{
  // ... generate output data ...
  async_write(socket_, out_buffer_,
    [this](error_code ec, size_t n)
    {
      if (!ec) do_write();
    });
}
```

Coordinating multiple chains

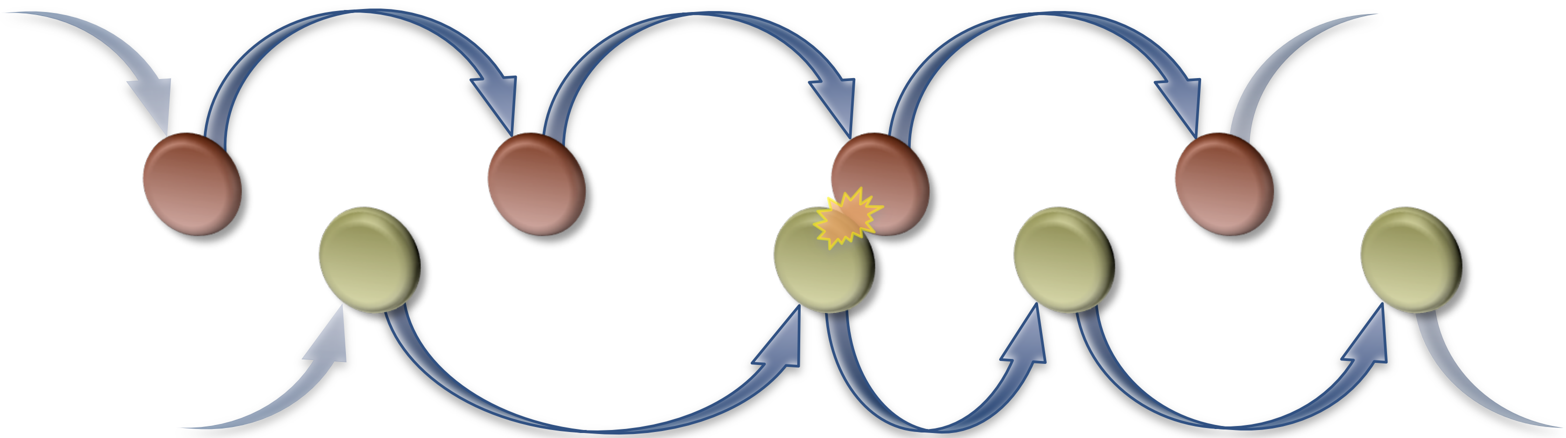
- With a single-threaded event loop, only one handler can execute at a time.



- No synchronization is required to protect shared data.

+ Coordinating multiple chains

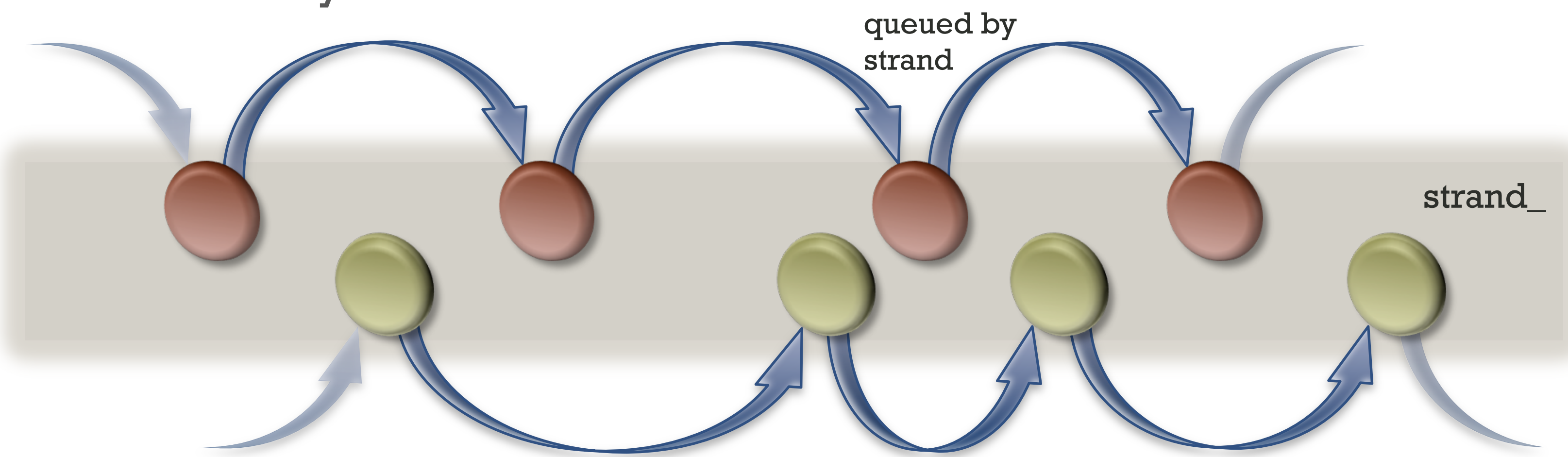
- However, if we choose to execute the completion handlers on a thread pool ...



- ... we may introduce data races.

+ Coordinating chains using a strand

- A strand ensures that completion handlers never execute concurrently.



- Explicit synchronization is still not required to protect shared data.

+ Coordinating chains using a strand

- To implement this, we use a single strand for all asynchronous operations associated with an object.

```
void connection::do_read()
{
  socket_.async_read_some(in_buffer_,
    wrap(strand_, [this](error_code ec, size_t n)
      {
        // ... process input data ...
        if (!ec) do_read();
      }));
}
```

- The wrap function is used to associate an executor with an object.
  - In this example, we associate the strand with the lambda.

Coordinating chains using other executor types

- The wrap function works with any executor or execution context.

```
void connection::do_read()
{
  socket_.async_read_some(in_buffer_,
    wrap(pool_, [this](error_code ec, size_t n)
      {
        // ... process input data ...
        if (!ec) do_read();
      }));
}
```

- Here we are associating a thread pool with the lambda.

The associated executor

- Rather than using the wrap function, the associated executor can be manually specified.
  - Provide a nested executor_type typedef and a get_executor member function.

- Example: hand-rolled function object

```
struct my_function {
  typedef system_executor executor_type;

  executor_type get_executor() const noexcept {
    return system_executor();
  }

  void operator()() { ... }
};
```

Executor-aware asynchronous operations

- For this to work correctly, an asynchronous operation must participate in an executor-aware model.
- An executor-aware asynchronous operation must:
  - Ask the completion handler for its associated executor.
  - While pending, maintain an executor_work object for the associated executor.
    - Tells the executor to expect a function object in the future.
    - Example: tells a thread pool to keep running.
  - Dispatch, post or defer any intermediate handlers, and the final completion handler, through the associated executor.
    - Ensures handlers are executed according to the rules.
    - Example: execute all handlers within the same strand.

Example: an executor-aware asynchronous file read

- Asynchronously read a line from a file and pass the string to the handler.

```
template <class Handler>
void async_getline(std::istream& is, Handler handler)
{
  // Create executor_work for the handler’s associated executor.
  auto work = make_work(handler);

  post([&is, work, handler=std::move(handler)]() mutable {
      std::string line;
      std::getline(is, line);

      // Pass the result to the handler, via the associated executor.
      dispatch(work.get_executor(),
        [line=std::move(line), handler=std::move(handler)]() mutable {
          handler(std::move(line));
        });
    });
}
```

+ Composing executor-aware asynchronous operations

- When composing asynchronous operations, intermediate operations can simply reuse the associated executor of the final handler.

```
template <class Handler>
void async_getlines(std::istream& is, std::string init, Handler handler)
{
  // Get the final handler's associated executor.
  auto ex = get_associated_executor(handler);

  // Use the associated executor for each operation in the composition.
  async_getline(is,
      wrap(ex, [&is, lines=std::move(init), handler=std::move(handler)]
        (std::string line) mutable
        {
          if (line.empty())
            handler(lines);
          else
            async_getlines(is, lines + line + "\n", std::move(handler));
        }));
}
```

The executors library and asynchronous operations

- Executors and execution contexts are key parts of an asynchronous model.
- The functions provided by the executors library …
  - dispatch, post, defer
  - dispatch_at, post_at, defer_at
  - dispatch_after, post_after, defer_after
- … are really just executor-aware asynchronous operations.

Summary of executors library key features

Type traits

- Class template handler_type
  - Transforms a completion token into a completion handler.
- Class template async_result
  - Determines the result of an asynchronous operation's initiating function.
- Class template async_completion
  - Helper to simplify implementation of an asynchronous operation.

Memory

- Class template associated_allocator
  - Used to determine a handler’s associated allocator.
- Function get_associated_allocator.
  - Obtain a handler’s associated allocator.

Executors

- Class template execution_context
  - Base class for execution context types.
- Class template associated_executor
  - Used to determine a handler’s associated executor.
- Function get_associated_executor
  - Obtain a handler’s associated executor.
- Class template executor_wrapper
  - Associates an executor with an object.
- Function wrap
  - Associate an executor with an object.

Executors

- Class template executor_work
  - Tracks outstanding work against an executor.
- Function make_work
  - Create work to track an outstanding operation.
- Class system_executor
  - Executor representing all threads in system.
- Class executor
  - Polymorphic wrapper for executors.

Executors

- Functions dispatch, post and defer
  - Execute a function object.
- Class template strand
  - Executor adapter than runs function objects non-concurrently and in FIFO order.

Timers

- Functions dispatch_at, post_at and defer_at
  - Execute a function at an absolute time.
- Functions dispatch_after, post_after and defer_after
  - Execute a function after a relative time.

Futures

- Class template specialization async_result for packaged_task
  - Supports use of packaged_task with dispatch, post, defer, etc.
- Class template packaged_handler
  - Implements lazy creation of a packaged_task.
- Class template packaged_token
  - Implements lazy creation of a packaged_task.
- Function package
  - Return a packaged_token for use with dispatch, post, defer, etc.

Execution contexts

- Class thread_pool
  - A fixed size thread pool.
- Class loop_scheduler
  - A thread pool where threads are explicitly donated by the caller.