Work Contracts in Action

Advancing High-Performance, Low-Latency Concurrency in C++

MICHAEL MANISCALCO

About Me:

- Michael A Maniscalco
  - C++ Software Architect and Principal Developer at Lime Trading
  - We develop low latency market data and trading software for use in HFT
  - Specialty - Ultra Low Latency Software Development (HFT)
  - At Heart - An Algorithms Guy
    - M03 - Compression by Substring Enumeration (2000)
    - M99 - BWT Compression (1999)
    - MSufSort - Suffix Array Construction Algorithm (1999)
  - Invented Work Contracts - *the subject of this talk* (2019)

- Personal
  - github.com/buildingcpp
    - Work Contracts, Networking, Messaging, Glimpse Instrumentation, etc …
  - github.com/michaelmaniscalco
    - Algorithms and Data Compression

Resources:

Source Code:

github.com/buildingcpp/work_contract

Lime Trading:

Lime.co

LIME TRADING

Contact:

wc@michael-maniscalco.com

Work Contract Talk From CppCon 2024:

www.youtube.com/watch?v=oj-_v[ZNMVw

“Everything should be made as simple as possible, but not simpler.”

- Albert Einstein

A Quick Review From CppCon 24

Work Contracts and Signal Trees

Recap from CppCon 2024: About Work Contracts

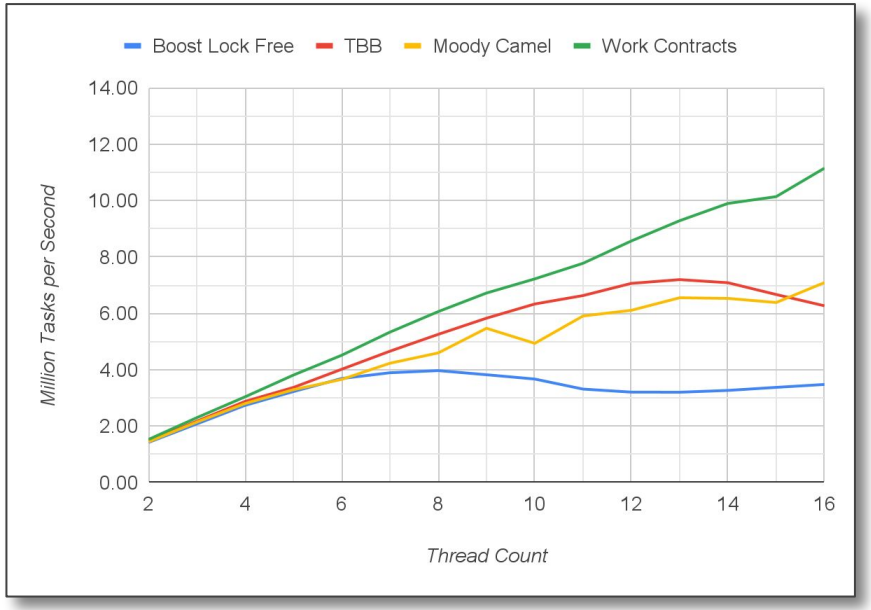
- **Signal Trees** - a novel concurrency structure for lock-free, wait-free task arbitration
  - Scheduling tasks is wait free - $O(1)$
  - Selection is lock free - expected $O(1)$
  - Worst case $O(n)$ but n is small in practice - typical around 8

- **Intrinsic Fairness -** signal trees are inherently at fair scheduling
  - No starvation: every contract gets fair execution time

- **Demonstrated Performance** - 40x-100x faster than queue-based concurrency under contention (shown in 2024)

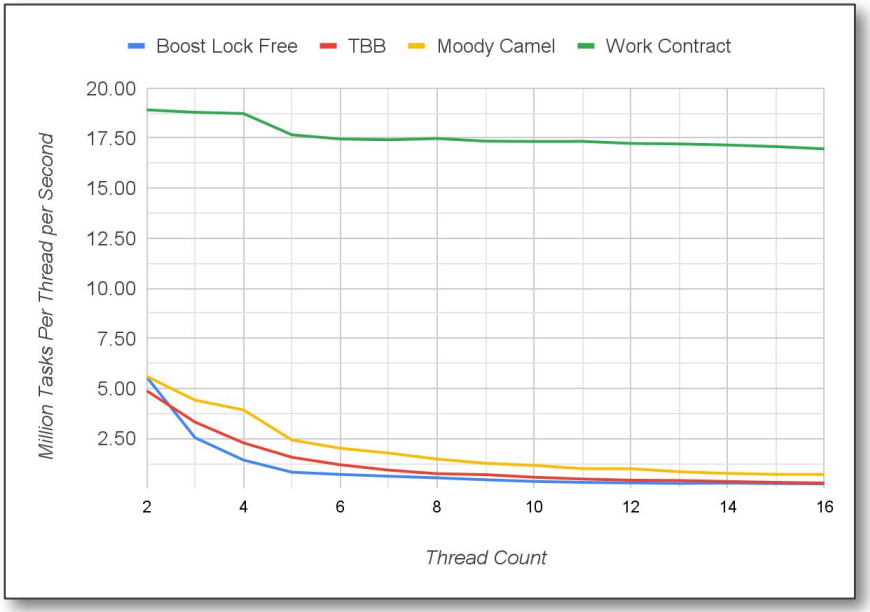
- **Work Contracts** - a low-latency concurrency abstraction utilizing signal trees

Performance Advantage: Benchmark (medium contention)



Coefficient of Variation at 16 Cores:	Boost Lock Free	TBB	Moody Camel	Work Contract
Task Fairness	< 0.01	< 0.01	1.02	0.03
Thread Fairness	0.04	0.04	0.21	0.02

Performance Advantage: Benchmark (high contention)



Coefficient of Variation at 16 Cores:	Boost Lock Free	TBB	Moody Camel	Work Contract
Task Fairness	< 0.01	< 0.01	1.05	< 0.01
Thread Fairness	0.07	0.13	0.03	< 0.01

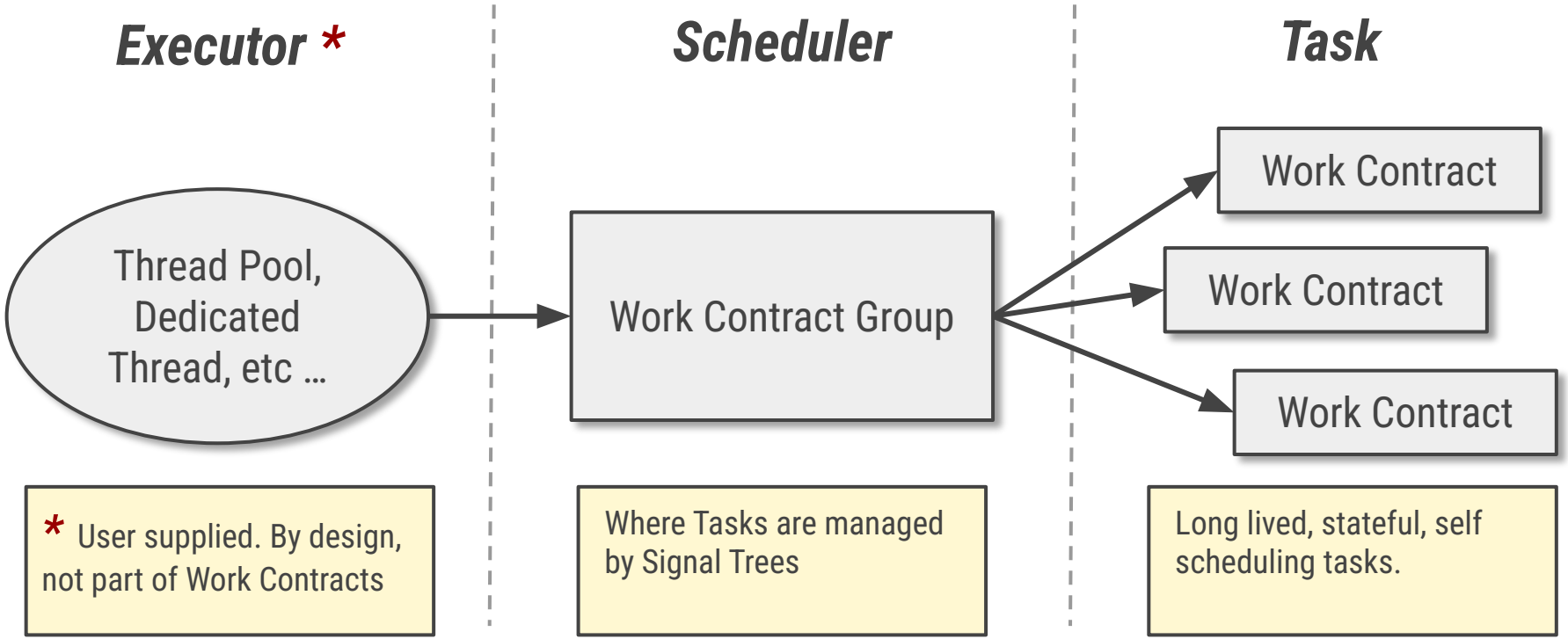
Where and Why Signal Trees Shine

- **Deterministic Latency** - workloads requiring deterministic performance under stress
- **Linear Scalability** - throughput grows with number of cores applied
- **Efficiency Under Pressure** - scheduling remains fair and balanced at scale
- **Enables leaner systems -** minimizes contention at the source by eliminating queues entirely

Mapping Things Out

Concurrency and Where Work Contracts Fits In

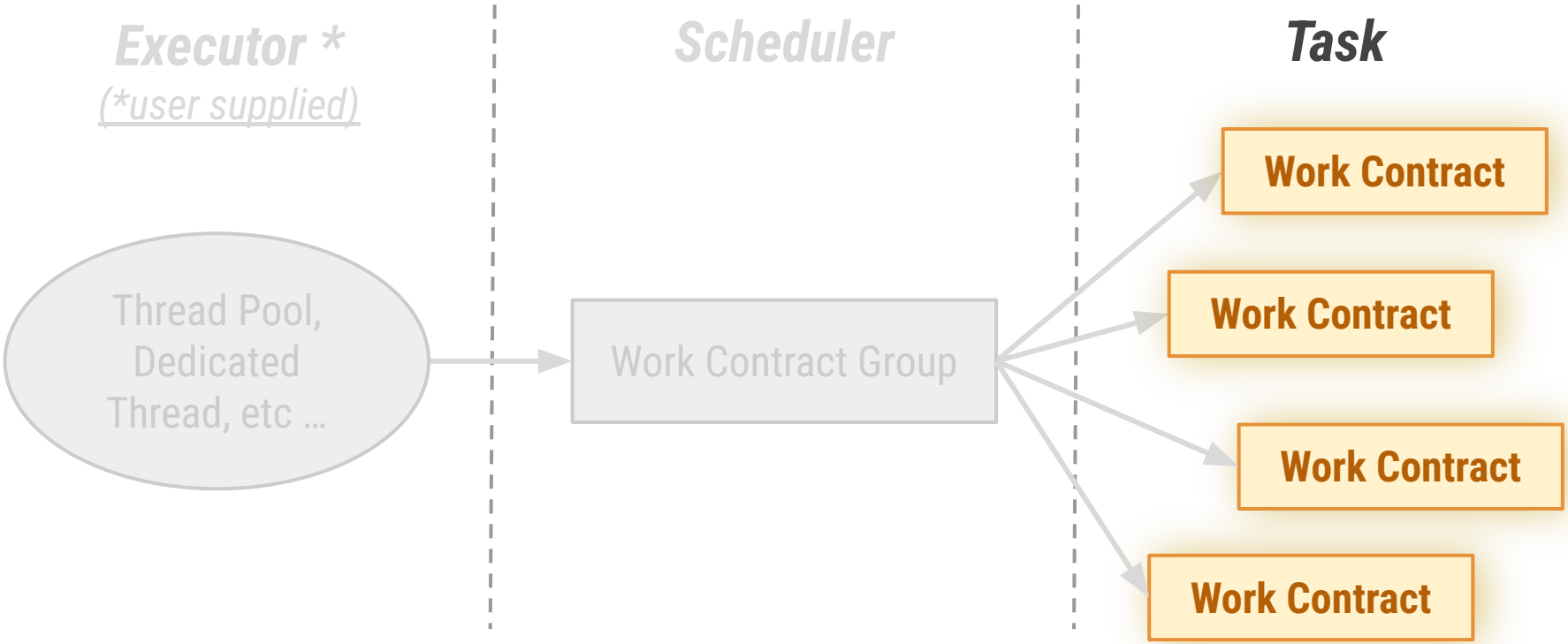
Overview: The Components of Work Contracts



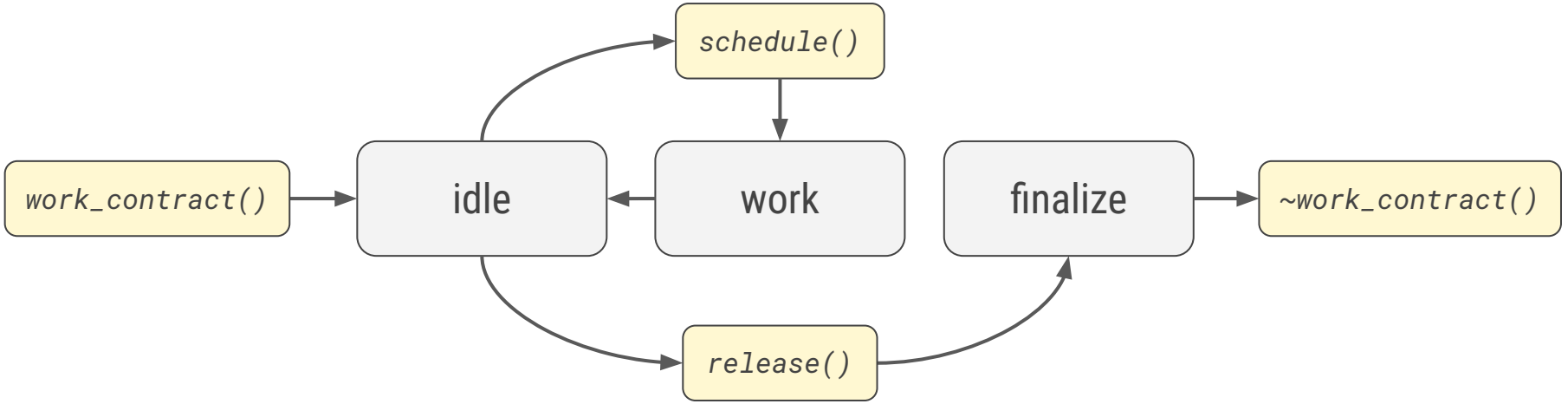
Component 1 - The Work Contract

“The Task”

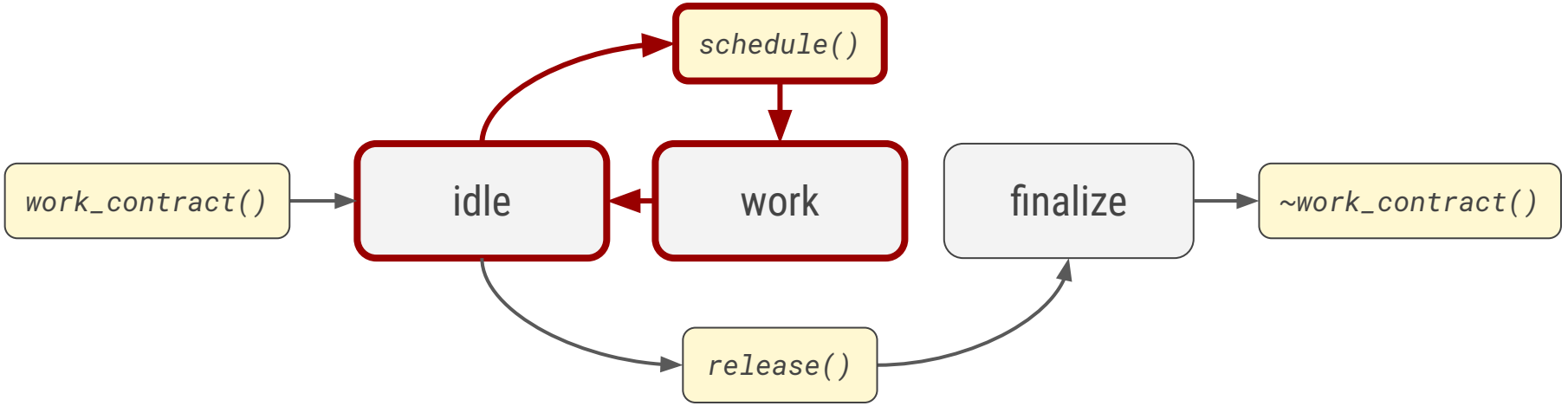
Work Contracts: The Tasks



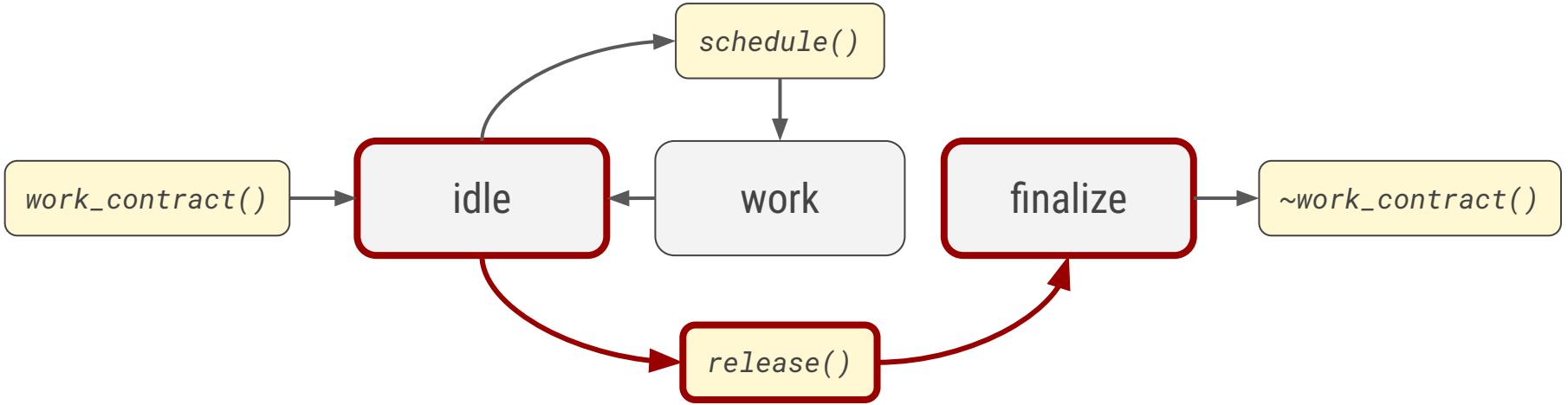
The Lifecycle of a Work Contract



Repeatable Execution States



One-Shot Finalization States



Work Contracts - In A Nutshell:

- Callable that is guaranteed to be executed once per scheduling
- Can be rescheduled as often as is required
- Scheduling is wait-free/lock-free and with $O(1)$ time
- Maintains State across execution calls
- Guaranteed single threaded execution without need for explicit synchronization

Work Contracts - In A Nutshell:

- Scheduling Coalescence - duplicate calls to schedule a contract coalesce into a single scheduling
- Deterministic Finalization - once a contract is ended, a contract will not run again; its cleanup then runs exactly once (asynchronously)
- Can self schedule during execution - **VERY USEFUL**

Component 1 - The Work Contract

What it is ***Not***

What it is *not*:

- **Not a future/promise** — no return value delivery, no blocking.
  - This is the opposite of the Work Contract design philosophy

- **Not one-shot -** contracts are intended for long duration and multiple executions of some specific logical task

Component 2 - Work Contract Group

“The Scheduler”

Work Contract Group: The Scheduler

LIME TRADING

Work Contract Group - At A Glance:

- **A Scheduler** - owns and schedules one or more work contracts
- **Selection Fairness** - ensures fair selection and execution of contracts
  - No task starvation
- **Signal Tree Based** - uses signal trees for low contention and high-efficiency
  - Covered in CppCon 2024 talk

Work Contract Group - Key Properties:

- **Lock Free Selection -** expected $O(1)$, worst-case $O(n)$ using signal-tree arbitration
- **Single Threaded Contract Execution** - Ensures that contracts are run by only one thread at a time
- **Parallel Execution -** Different contracts can run in parallel
- **Parallelism within a single contract** - planned for the future

Component 2 - Work Contract Group

What it is ***Not***

What a Work Contract Group is *not*:

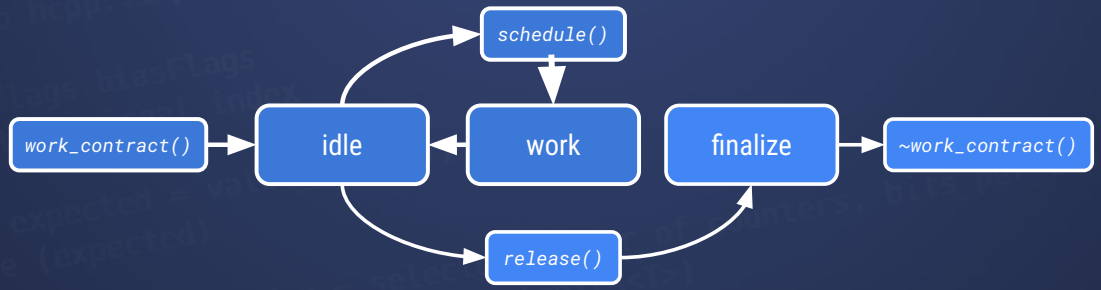
- **Not a queue** - a work contract group doesn't order work
  - It uses a signal tree to track each contract's scheduled status

- **Does not spawn threads** - a work contract group doesn't create threads
  - It relies on an external executor to provide threads for running contracts
  - It is only the scheduler and only selects which contract runs next

Summary

Work Contract (task)

- Primary work callable - the “Contract” (repeatable)
- Optional release callable (one-shot finalization)
- Single threaded execution guarantee
- Deterministic execution flow
  - Idle -> schedule -> run -> idle
  - Idle -> release -> finalize -> destroy



Summary

Work Contract Group (scheduler)

- Fair selection across contracts (no starvation)
- Lock-free selection based on signal trees
- Ensures single threaded execution of contracts
- No queues - threading contention virtually eliminated
- Allows for near linearly scaling with number of cores

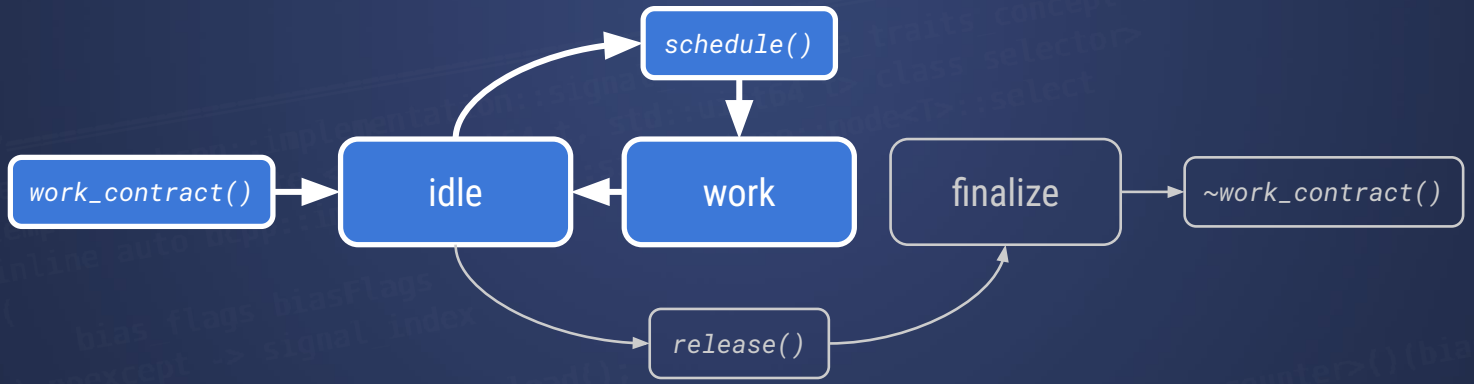
Work Contract Fundamentals

Basic Contract Execution

- Work contract groups
- Creating work contracts
- Scheduling work contracts
- Executing work contracts

Work Contract Fundamentals

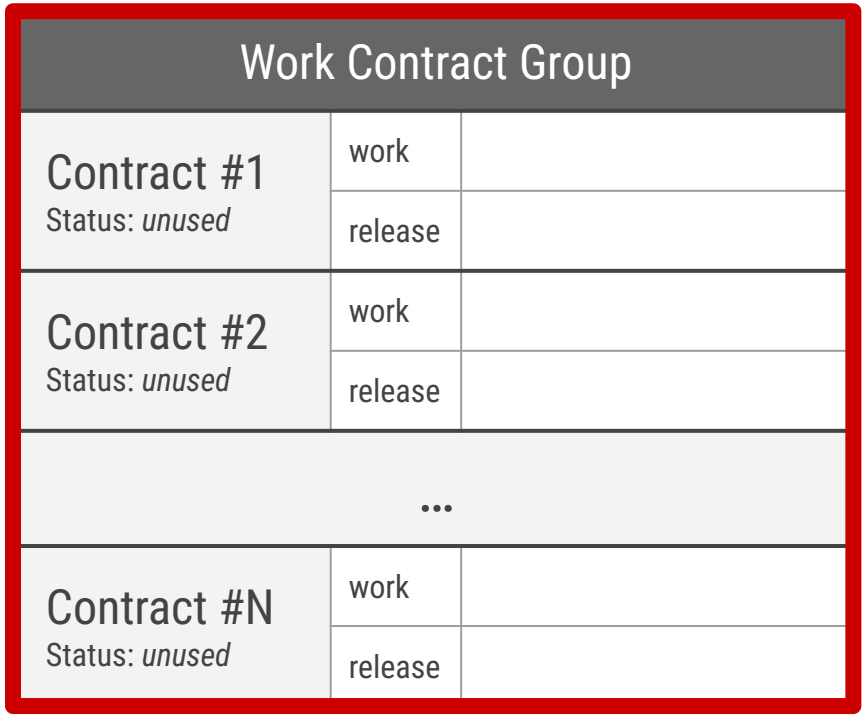
Basic Contract Execution



Fundamentals: Basic Contract Execution

```
// Create a work contract group
bcpp::work_contract_group group;
```

Create the work contract group
This is where all the contracts are contained.



Fundamentals: Basic Contract Execution

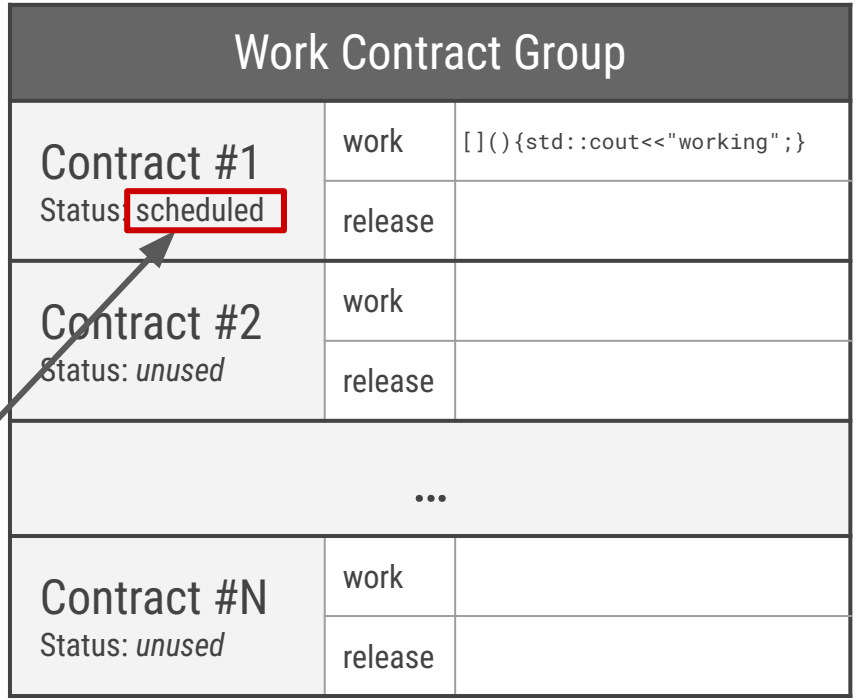
```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";});
```

Create a contract within the group and assign the primary work callable at the time of creation.

Fundamentals: Basic Contract Execution

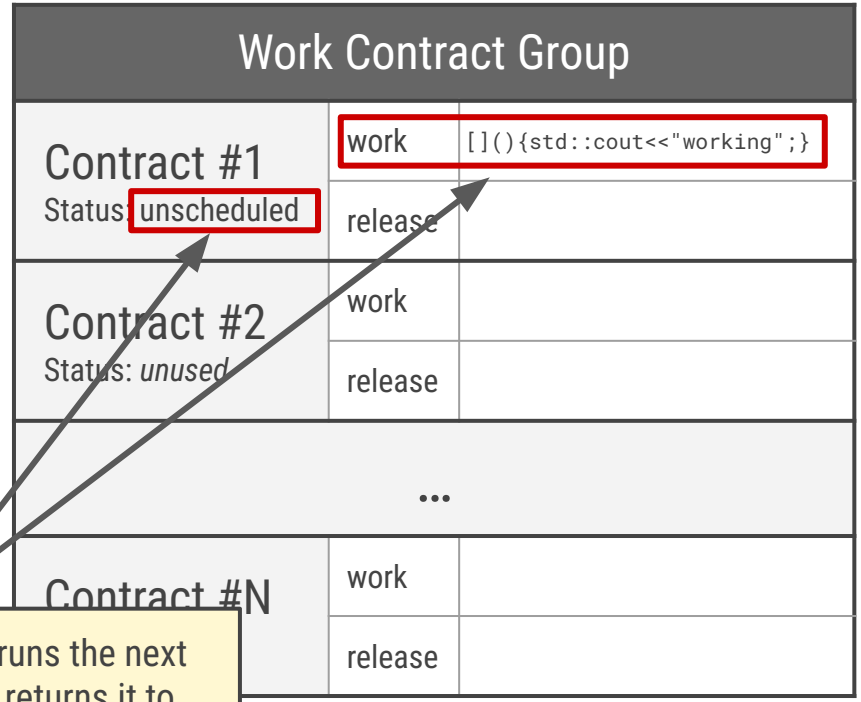
```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";});

// Don't forget to schedule the contract
contract.schedule();
```

By default, contracts are unscheduled. They must be scheduled before they can be run.



Fundamentals: Basic Contract Execution

```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";});

// Don't forget to schedule the contract
contract.schedule();

// Select and invoke the next scheduled contract
group.execute_next_contract();
```



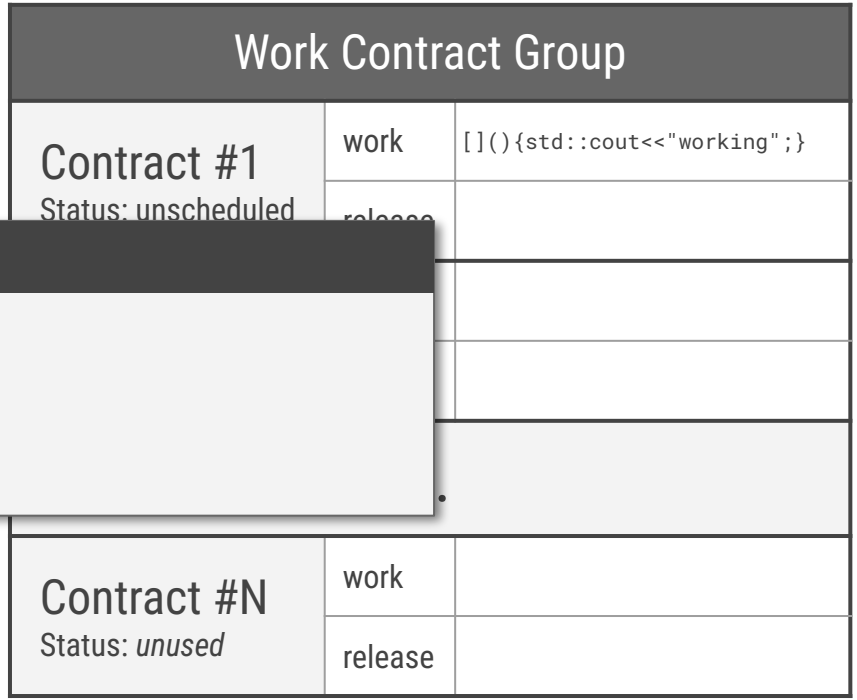
execute_next_contract() runs the next scheduled contract, then returns it to unscheduled.

Fundamentals: Basic Contract Execution

```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout <

// Don't forget to s
contract.schedule();

// Select and invoke
group.execute_next_contract();
```

Output

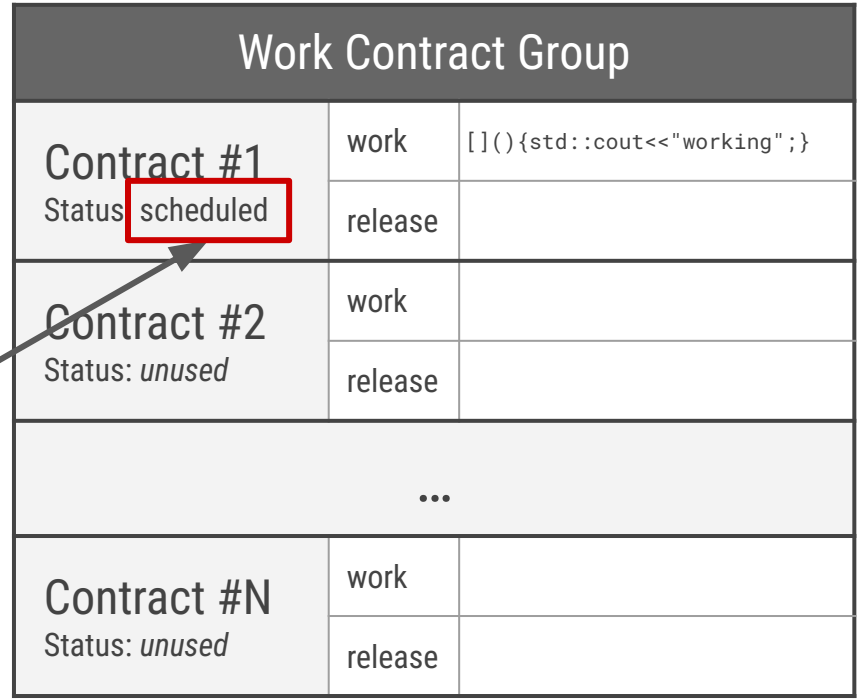
```
working
```



Fundamentals: Basic Contract Execution

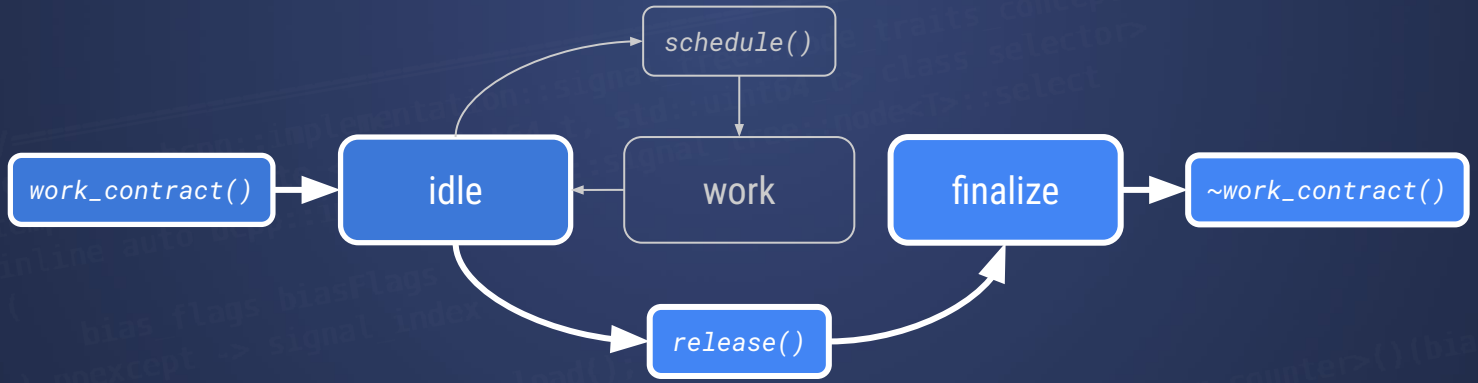
```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";},
    work_contract::initial_state::scheduled);
```

The initial state of the contract can also be specified during creation.



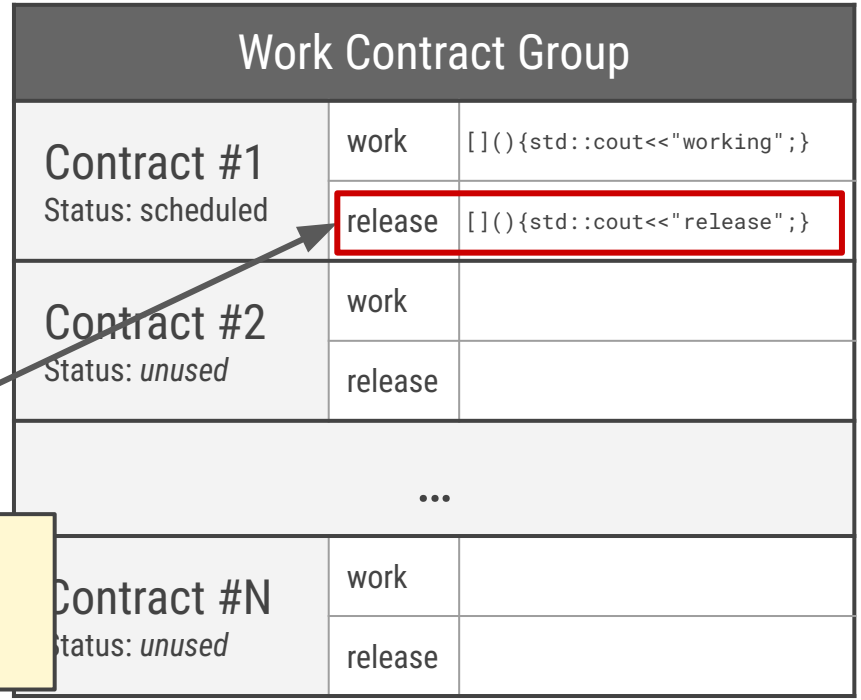
Work Contract Fundamentals

Contract Release



Fundamentals: Basic Contract Execution

```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";},
    [](){std::cout << "release";},
    work_contract::initial_state::scheduled);

// Run next scheduled contract
group.execute_next_contract();

// Release the c
contract.release

// Run next scheduled contract
group.execute_next_contract();
```



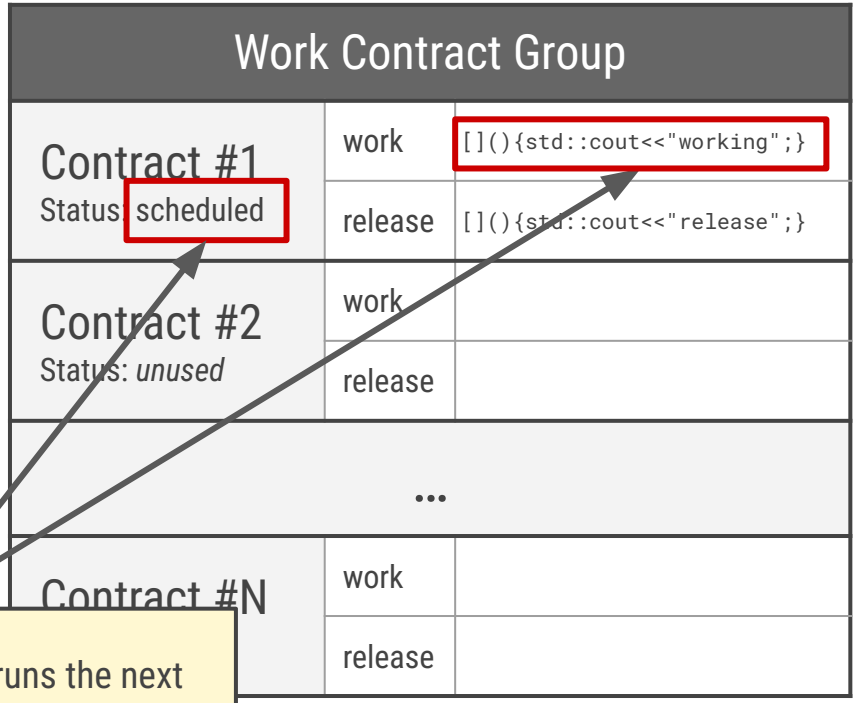
create_contract() can take a optional release callable. This callable will be run once at contract release.

Fundamentals: Basic Contract Execution

Output

```
working
```

```
    tract(
    },
    []{std::cout << "release";},
    work_contract::initial_state::scheduled);

// Run next scheduled contract
group.execute_next_contract();

// Release the contract
contract.release();

// Run next scheduled contra
group.execute_next_contract
```



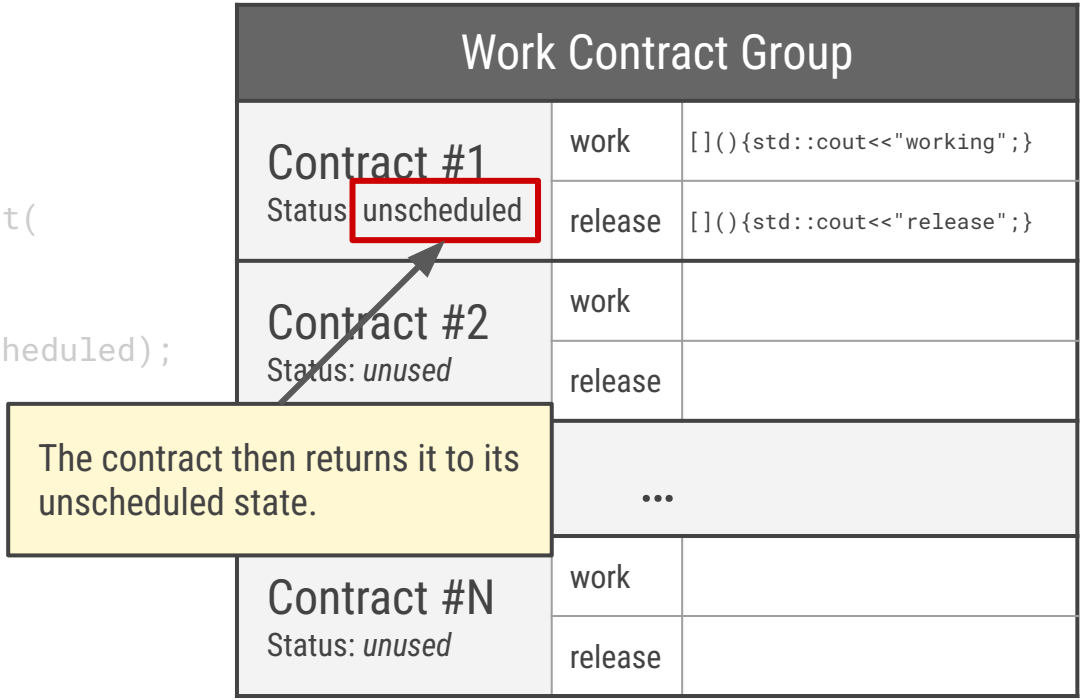
execute_next_contract() runs the next scheduled contract, as expected.

Fundamentals: Basic Contract Execution

Output

```
working
```

```
tract(
},
[](){std::cout << "release";},
work_contract::initial_state::scheduled);

// Run next scheduled contract
group.execute_next_contract();

// Release the contract
contract.release();

// Run next scheduled contract
group.execute_next_contract();
```



Fundamentals: Basic Contract Execution

Output

```
working
```

```
tract(
},
[](){std::cout << "release";},
work_contract::initial_state::scheduled);

// Run next scheduled contract
group.execute_next_contract();

// Release the contract
contract.release();

// Run next scheduled contract
group.execute_next_contract();
```

Work Contract Group		
Contract #1 Status: scheduled	work	`[](){std::cout<<"working";}`
	release	`[](){std::cout<<"release";}`
Contract #2 Status: *unused*	work	
	release	

Releasing a contract schedules the contract for its final run. At next run the release callable is called instead of the work callable.

Fundamentals: Basic Contract Execution

Output

```
working
release
```

```
tract(
},
[](){std::cout << "release";},
work_contract::initial_state::sc

// Run next scheduled contract
group.execute_next_contract();

// Release the contract
contract.release();

// Run next scheduled contract
group.execute_next_contract();
```

Work Contract Group

Contract #1 Status: released	work	`[](){std::cout<<"working";}`
	release	`[](){std::cout<<"release";}`
Contract #2	work	
Contract #N Status: *unused*	release	

When selected, the release callable is run rather than the work callable.

Note: As with the work callable, the release callable is asynchronous.

Fundamentals: Basic Contract Execution

```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){std::cout << "working";},
    [](){std::cout << "release";},
    work_contract::initial_state::scheduled);

// Run next sche
group.execute_ne

// Release the c
contract.release();

// Run next scheduled contract
group.execute_next_contract();
```

When the release callable completes the contract is invalid and becomes available for reuse.

Work Contract Fundamentals

Recurring Execution and Stateful Contracts

- Maintaining state across runs
- Self Scheduling

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};
```

Use a functor as the work contract

```
bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcep
    {
        if (n > 0)
            std::cout << n--
    }
};
```

Hopefully starting to look familiar:

- Create work contract group
- Create contract in that group with initial contract state ‘scheduled’
- Drive execution by looping over `execute_next_contract()`

```
bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

One new concept:

```
bool work_contract::is_valid() const noexcept;
```

Returns `true` for as long as the contract has not been destroyed and `false` thereafter

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};
```

What is the output?

```
bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

Output

3

(followed by an infinite loop)

Why ?

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
{
    group.execute_next_contract();
    c.schedule();
}
```

We could explicitly reschedule the contract after each execution …

Fundamentals: Stateful Contracts

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
{
    group.execute_next_contract();
    c.schedule();
}
```

Output

3
2
1
(followed by an infinite loop)

Now we have two problems:
Code smell ***and*** still broken!

Why?

Fundamentals: Self Scheduling

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
        this_contract::schedule();
        if (n <= 0)
            this_contract::release();
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, wor
while (c.is_valid())
    group.execute_next_contract();
```

Self Scheduling To The Rescue:

```
void this_contract::schedule() noexcept;
void this_contract::release() noexcept;
```

- Encapsulates the logic within the contract:
  - logic not scattered everywhere
- Eliminates external contract management:
  - not possible to schedule wrong contract
- Separation of concerns:
  - Executors are not concerned contract logic

Fundamentals: Self Scheduling

```
struct countdown
{
    int n;
    void operator()() noexcept
    {
        if (n > 0)
            std::cout << n-- << '\n';
        this_contract::schedule();
        if (n <= 0)
            this_contract::release();
    }
};


bcpp::work_contract_group group;
auto c = group.create_contract(countdown{3}, work_contract::initial_state::scheduled);
while (c.is_valid())
    group.execute_next_contract();
```

Output

3
2
1
(then contract is destroyed)

Self Scheduling: Takeaways …

Summary:

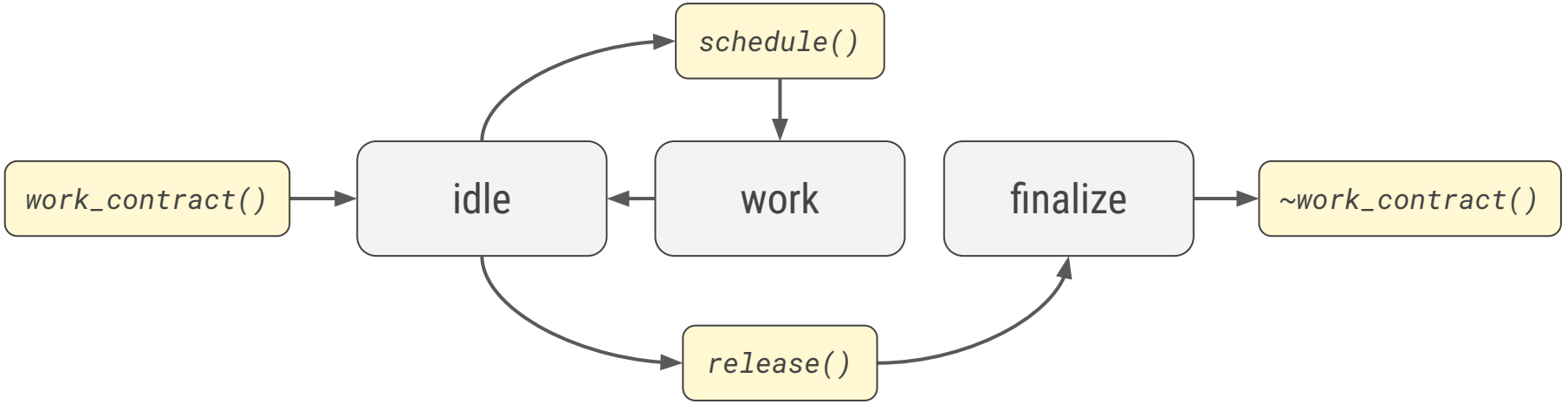
- **Work Contracts can maintain state across calls** - Obviously useful
- **Self Scheduling** - Significantly improves architecture
  - Eliminates coordination errors - no ‘wrong contract errors’
  - Separation of concerns - only the contract needs to be are of its own logic
  - Cleaner code - logic is localized and easier to follow/maintain
  - We could all use a little less “code smell” in our lives

Work Contract Fundamentals

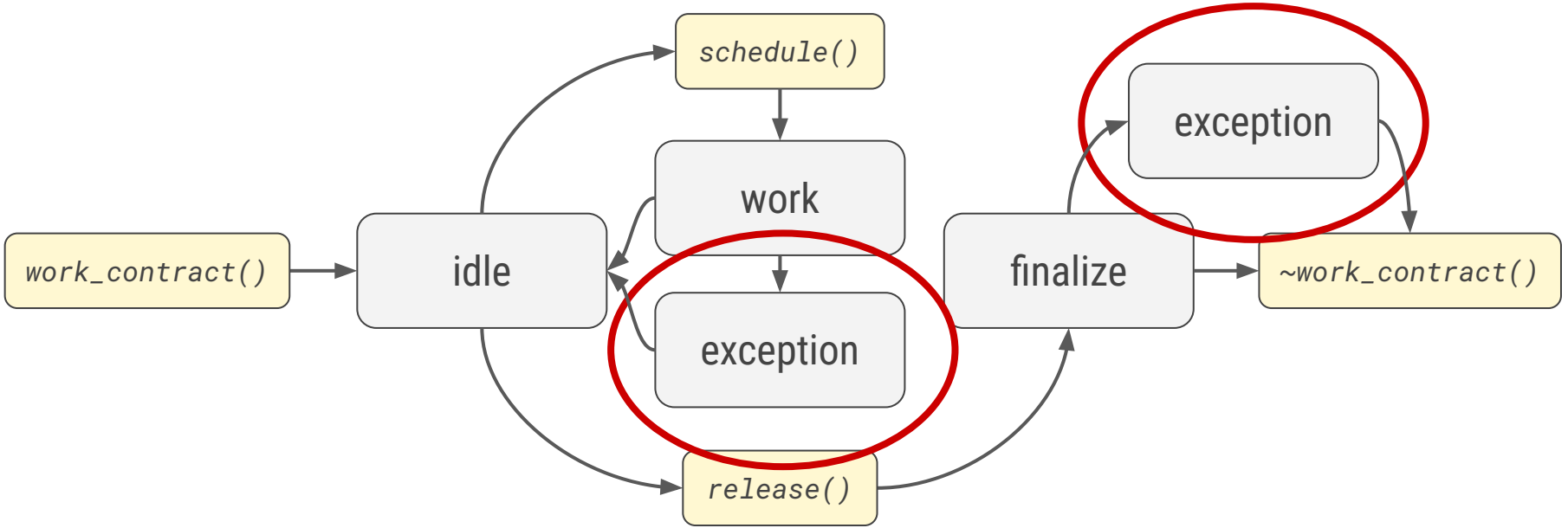
Exception Handling

- Handled inline (no propagation)
- Ensures continued execution

The Lifecycle of a Work Contract (Simplified)



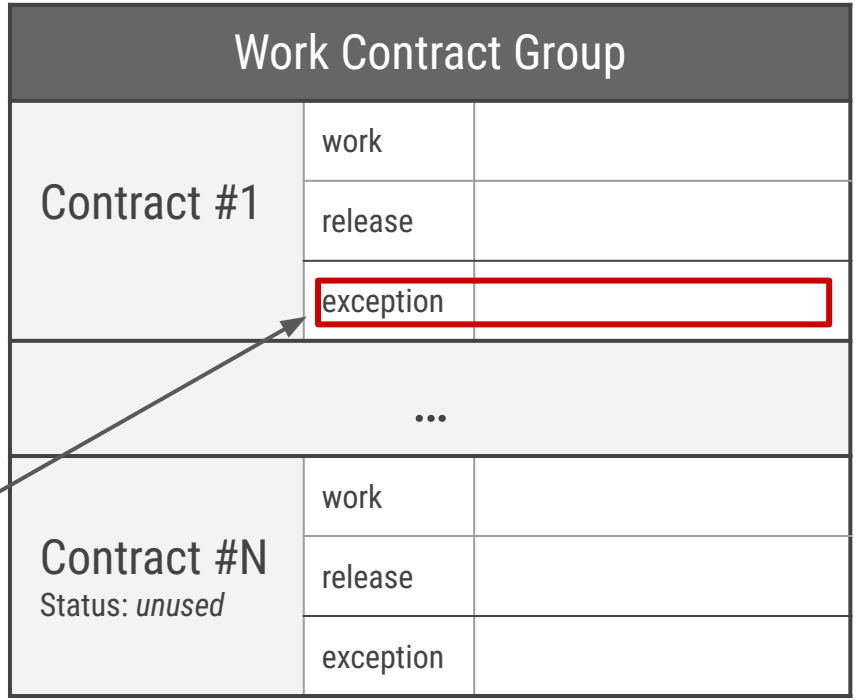
The Lifecycle of a Work Contract (With Exceptions)



Fundamentals: Exception Handling

```
// Create a work contract group
bcpp::work_contract_group group;

// Create a work contract
auto contract = group.create_contract(
    [](){/*work*/},
    [](){/*release*/},
    [](std::exception_ptr){});
```

Exceptions during the work or release callables are caught and forwarded to an optional exception callable.



Work and Release Callables are user defined, therefore:

- Both callables are internally placed within a try/catch*
- If an exception handler was assigned at contract creation then it will be immediately invoked with the exception (no deferred exception propagation)
- The contract state is then restored as expected:
  - Work callable -> return contract to unscheduled (idle) state
  - Release callable -> contract destroyed
- If an exception handler was ***not*** assigned at contract creation then the exception is now rethrown.

Work Contract Fundamentals

Execution Guarantees

- Single-threaded per contract
- Parallelism across the group

Work Contract Groups are lock-free thread safe:

- Parallel calls to `work_contract_group::execute_next_contract()` are safe
  - Lock free - no additional synchronization is required
- Parallel execution of scheduled contracts is safe - no worries
- Individual Work Contracts are guaranteed to be executed single threaded
  - No additional synchronization is required
  - Only one thread will execute the contract at a time
  - This is very useful!

Overview of WC Ecosystem

- Work Contracts
  - Wait-Free O(1) scheduling (with scheduling coalescence)
  - Repeatable execution
  - Async one-shot finalization
  - Self Scheduling
  - Stateful
  - Inline exception handling (no exception propagation required)
  - Guaranteed single threaded execution (without synchronization overhead)

- Work Contract Groups
  - Lock-free contract selection - O(1) expected, O(n) worst case
  - Low-Latency - parallel contract selection and execution
  - Effortless exceptionally fair task scheduling (without coordination overhead)
  - Thread Contention minimized to near zero
  - Scales linearly with number of cores

Blah Blah Blah - I’m Bored!

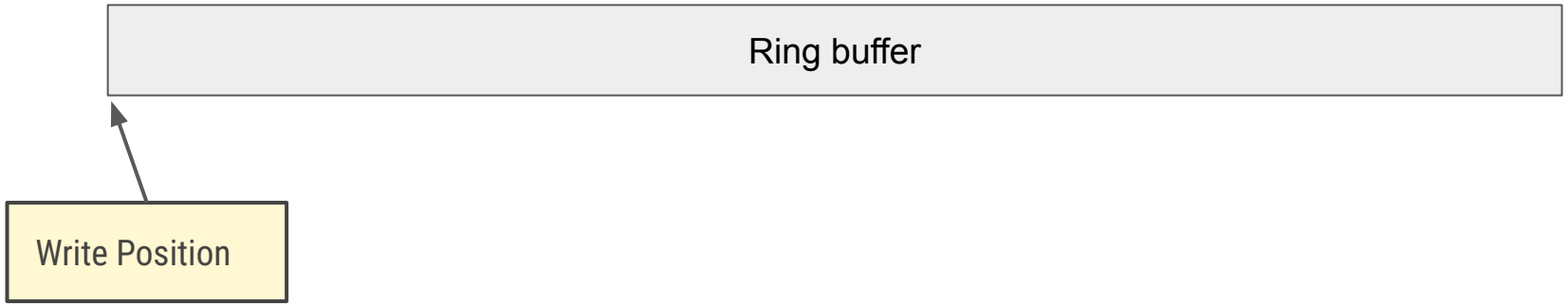
Let’s Build Something Real

- Building a high performance IPC “networking” component:
  - A (very) quick high level into to Multicast Queues
  - Building a multicast queue producer and consumer
  - Building the mcq_socket (receiver only)
  - Building the mcq_poller
  - Scale (massively) with Work Contracts
  - Live Demo (fingers crossed!)

Multicast Queue: High Level Overview

A multicast “mcast” queue is a ring buffer:

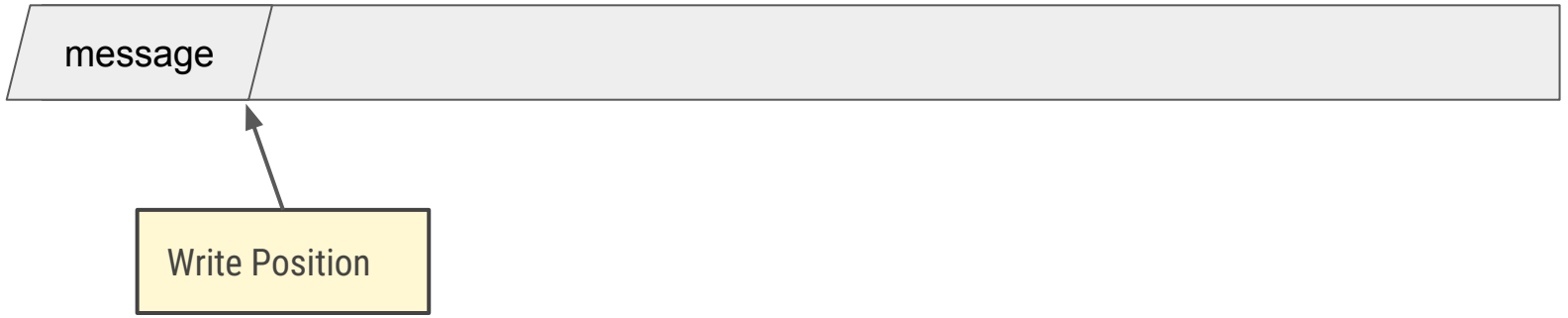
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

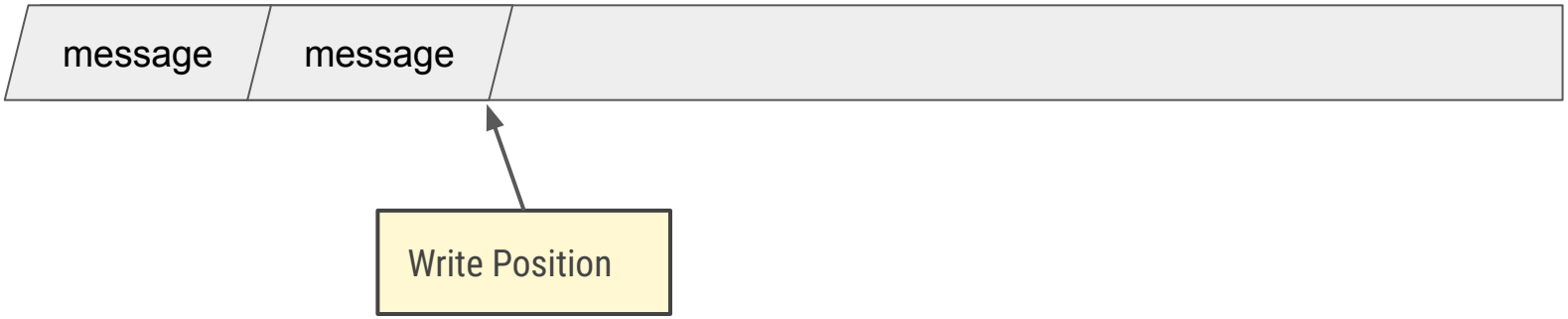
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

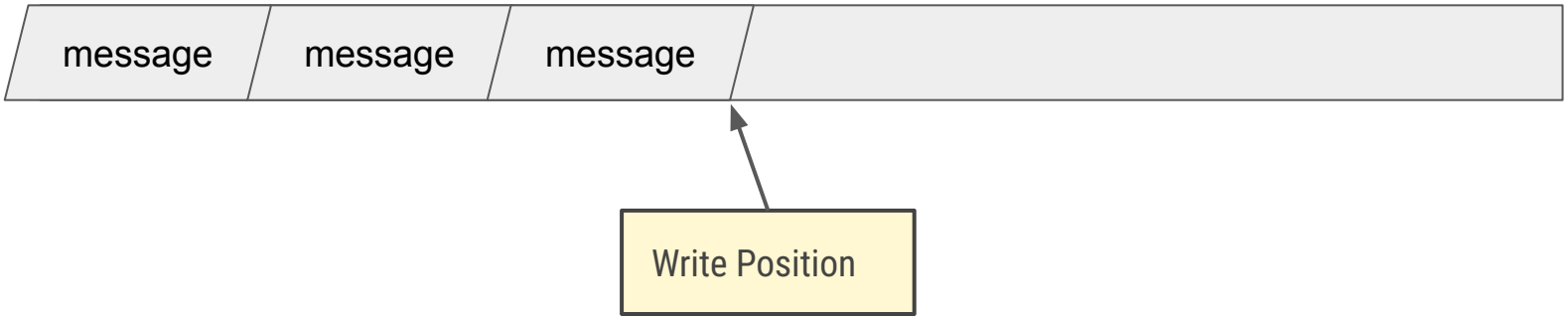
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

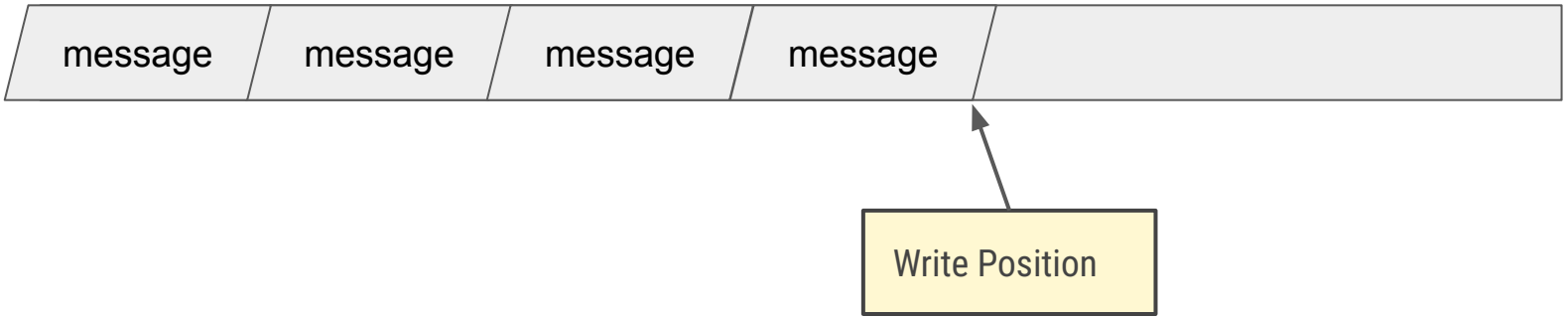
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

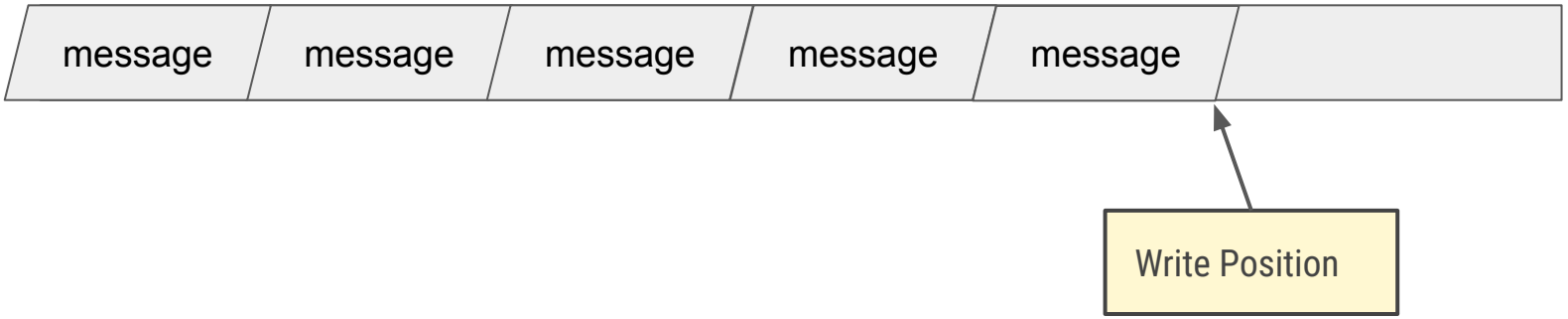
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

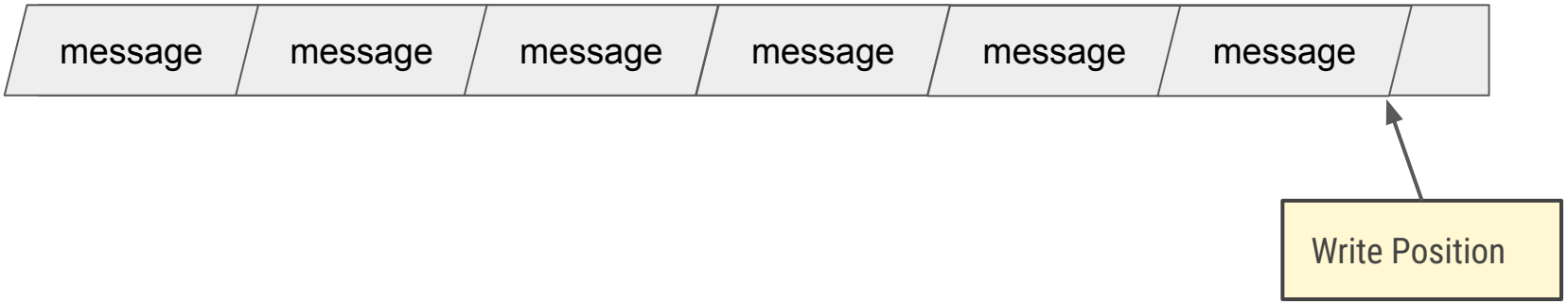
- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer



Multicast Queue: The Producer

A multicast “mcast” queue is a ring buffer:

- Write messages into the buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer

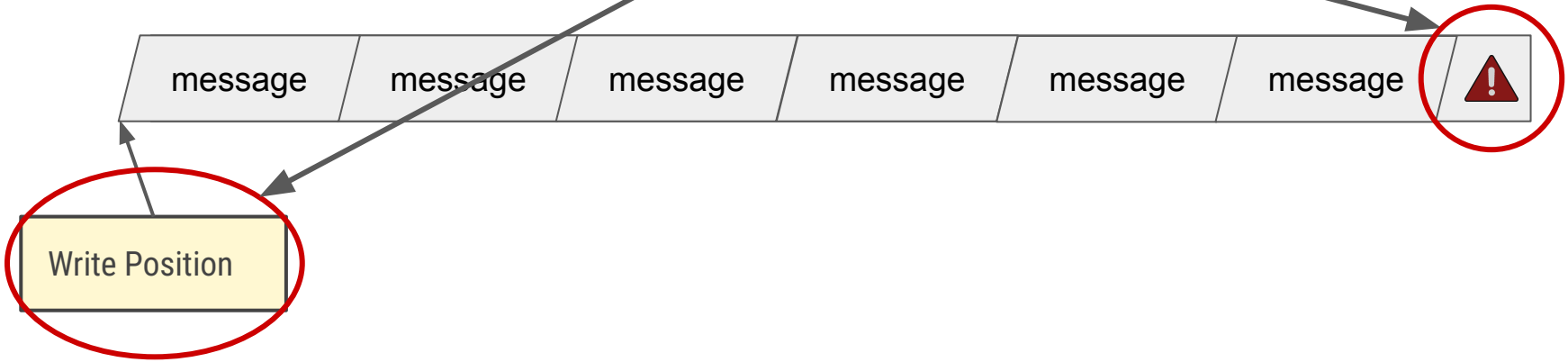


Multicast Queue: The Producer

A multicast “mcast” (

- Write messages
- When at end of buffer resume writing at start of buffer

> If sufficient space remaining for the next message. Move the write position to the beginning of the buffer.

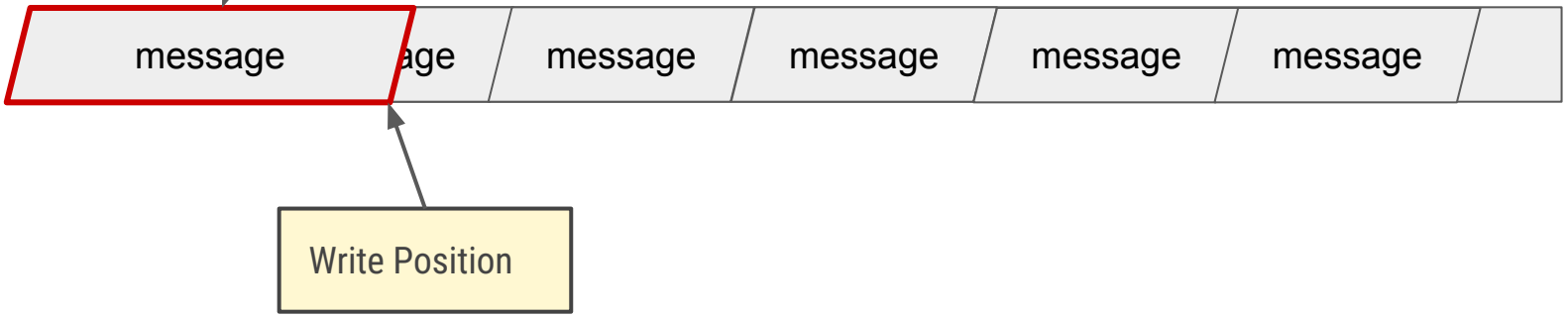


Multicast Queue: The Producer

a ring buffer:

- buffer starting at beginning of buffer
- When at end of buffer resume writing at start of buffer

Next message overwrites existing data starting at front of buffer.



Things To Notice:

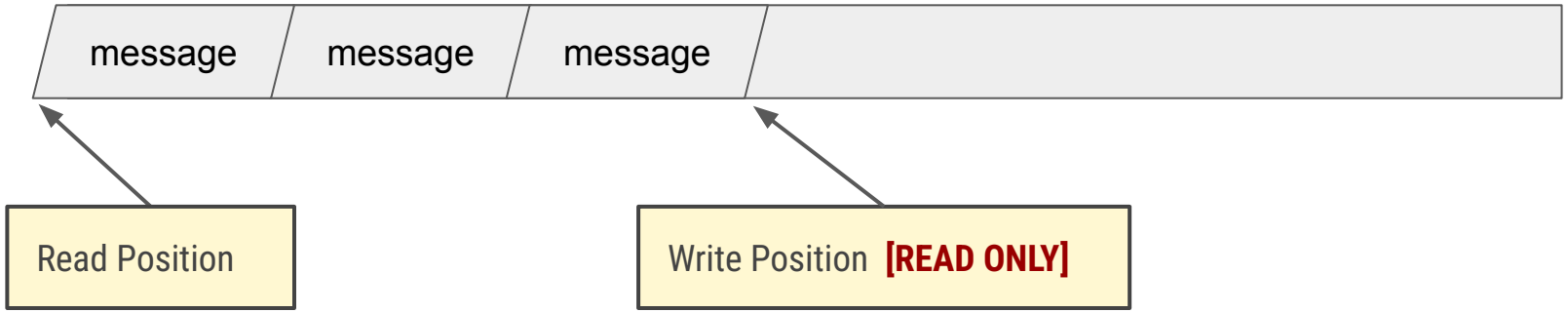
- There is no “read address”
  - The multicast queue producer only maintains the “write address”
  - If there were a “read address” then this would be a SPSC queue
  - Multicast queues are SPMC
  - Each consumer must maintain its own “read address”

- Upon producer wrap around existing data is overwritten
  - Multicast message delivery is never guaranteed
  - If consumer falls behind (gets lapped) then they lose messages and must recover by resyncing with the producer (many solutions to this)

Multicast Queue: The Consumer

Each consumer maintains its own read position

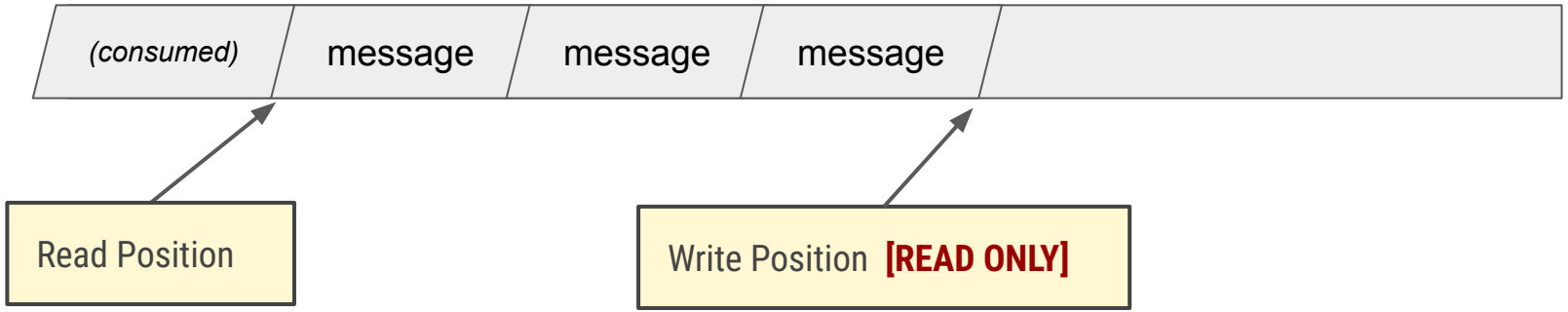
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

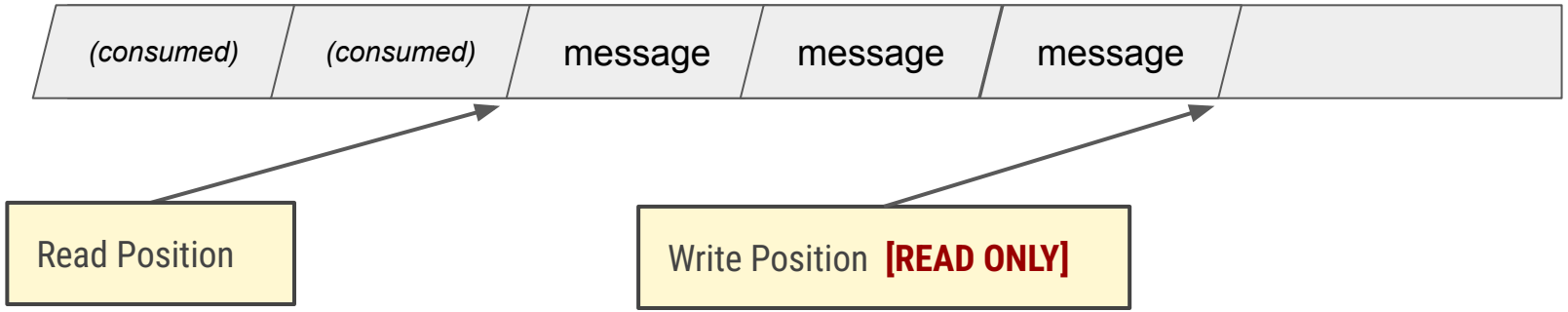
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

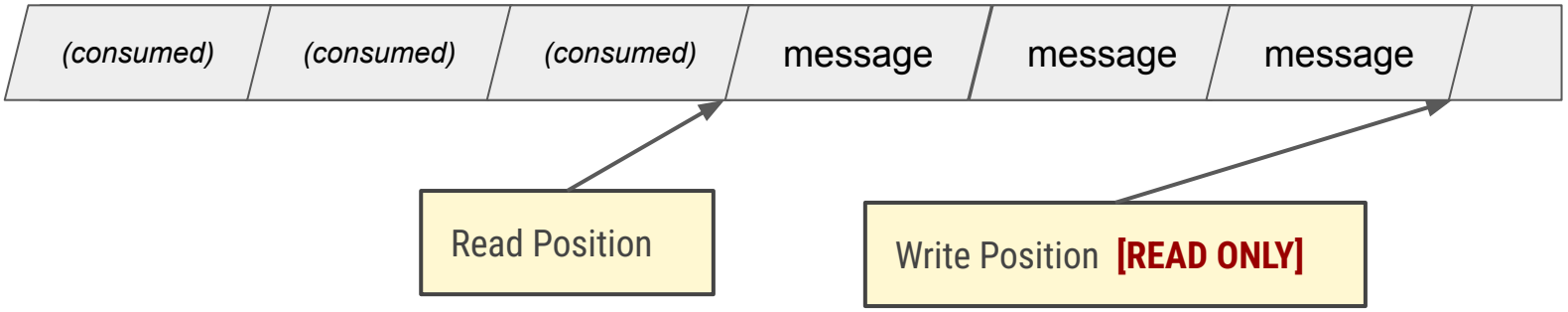
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

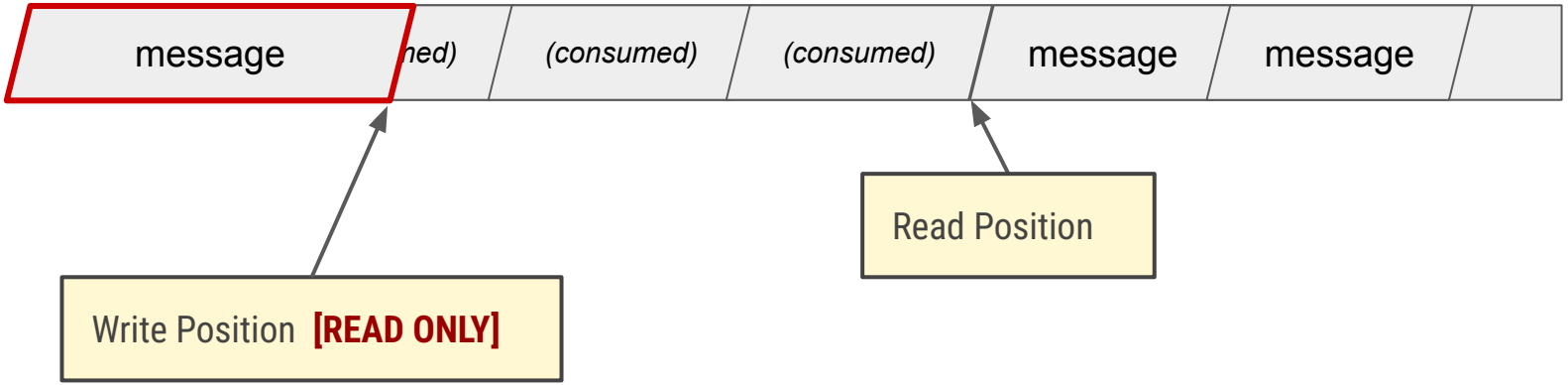
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

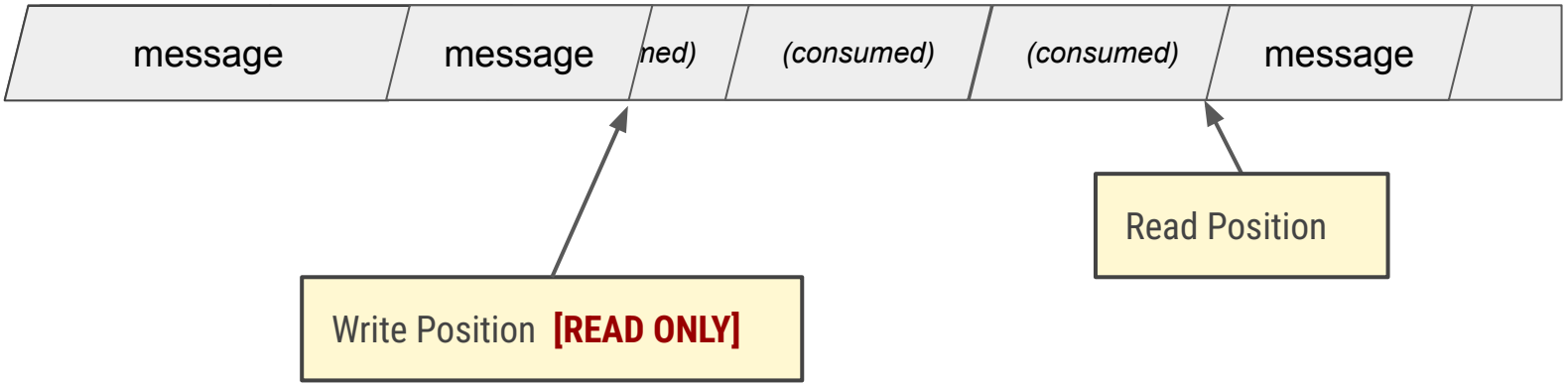
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

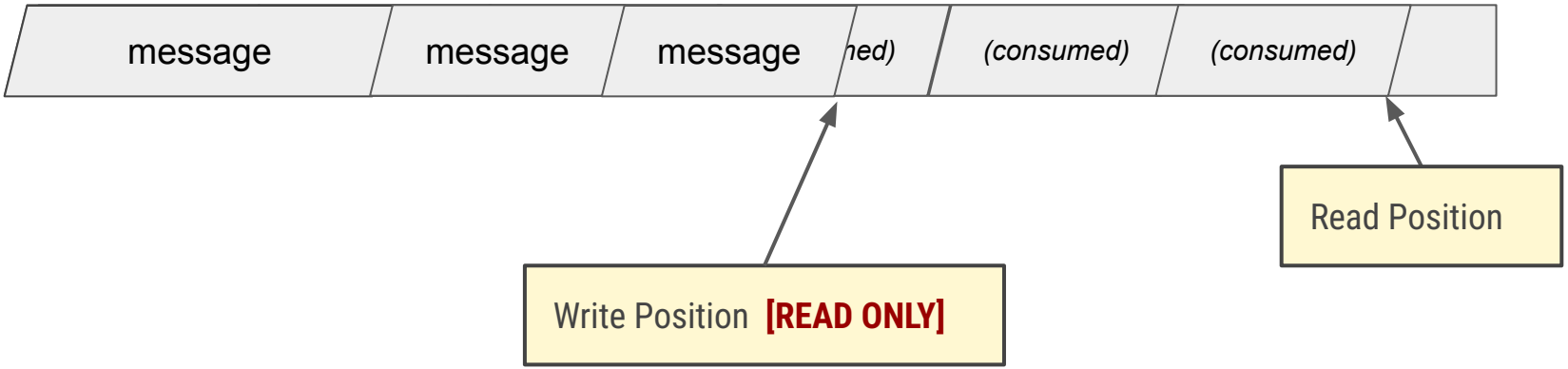
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

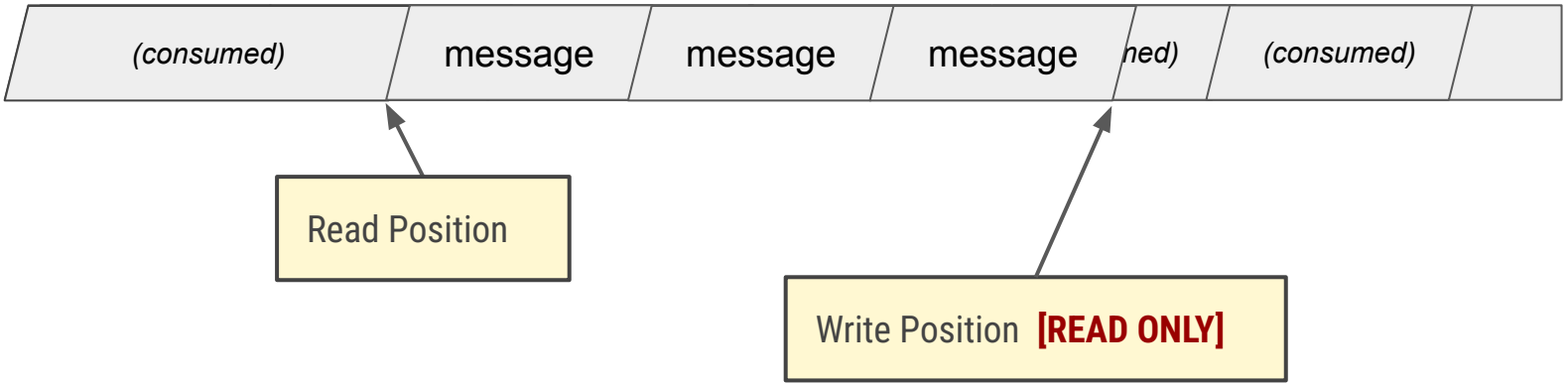
- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The Consumer

Each consumer maintains its own read position

- Consumers start reading from the front of the buffer
- Consume one message at a time and advance their read position
- Wrap to front of buffer when end is reached - just like producer does



Multicast Queue: The (minimal) producer API and example


```
class mcq_producer
{
public:
    void push(std::span<char const>) noexcept;
}; // class mcq_producer
```

```
mcqProducer.push("Guess what?");
mcqProducer.push("Chicken butt!");
```

Multicast Queue: The (minimal) consumer API and example


```
class mcq_consumer
{
public:
    std::int64_t pop(std::span<char> &) noexcept;
}; // class mcq_consumer
```

```
std::array<char, 2048> buf;
auto bytesAvailable = 0;
do
{
    std::span message(buf.data(), buf.size());
    bytesAvailable = mcqConsumer.pop(message);
    if (not message.empty())
        std::cout << std::string_view(message.data(), message.size()) << '\n';
} while (bytesAvailable > 0);
```

Multicast Queue: Without Concurrency

LIME TRADING

```
// some other process is the producer and just blasts out messages
for (auto i = 0; i < 1000000; ++i)
   mcqProducer.push(std::format("message # {}", i));
```

```
// some consumer process
std::array<char, 2048> buf;
while (true)
{
    std::span message(buf.data(), buf.size());
    if (mcqConsumer.pop(message); not message.empty())
        std::cout << std::string_view(message.data(), message.size()) << '\n';
}
```

Adding Concurrency

And Asynchronicity …

- Creating mcq_socket - a socket like API
  - mcq_socket::receive()
  - Asynchronous messaging

Multicast Queue: Multicast Queue Socket (message receiver)


```
class mcq_socket{
public:
   using message_handler = std::function<void(std::span<char const>)>;
   using loss_handler = std::function<void(std::uint64_t, std::uint64_t)>;

   mcq_socket
   (
      std::string_view shmPath,              // shared memory for the mcast queue
      message_handler messageHandler,        // callback for received messages (because async)
      loss_handler lossHandler               // callback for lost messages due to mcast lapping
   );

   void receive();                           // pops message from mcast receiver and
                                             // uses callback message_handler callback

private:

   mcq_consumer        mcqReceiver_;        // our multicast queue consumer
   message_handler     messageHandler_;     // callback for messages received
   loss_handler        lossHandler_;        // callback for message loss
};
```

Multicast Queue: Multicast Queue Socket (message receiver)


```
mcq_socket::mcq_socket
(
    std::string_view shmPath,
    message_handler messageHandler,
    loss_handler lossHandler
):
    mcqConsumer_(shmPath), messageHandler_(messageHandler), lossHandler_(lossHandler)
{
}


void mcq_socket::receive
(
    // try to receive next message from the mcq consumer and forward it through the handler
)
{
    std::array<char, 2048> buf;
    std::span message(buf.data(), buf.size());
    if (mcqConsumer_.pop(message); not message.empty())
        messageHandler_(message);
}
```

Multicast Queue: Multicast Queue Socket (message receiver)

Previously we had …

```
std::array<char, 2048> buf;
while (true)
{
    std::span message(buf.data(), buf.size());
    if (mcqConsumer.pop(message); not message.empty())
        std::cout << std::string_view(message.data(), message.size()) << '\n';
}
```

Now we have …

```
mcq_socket mcqSocket("path_to_shm",
   [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
   [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";});

while (true)
   mcqSocket.receive();
```

Multicast Queue: Multicast Queue Socket (message receiver)

Previously we had …

```
std::array<char, 2048> buf;
while (true)
{
    std::span message(buf.data(), mcqConsumer.pop(buf.data()));
    if (mcqConsumer.pop(buf.data()))
        std::cout << std::string_view(message.data(), message.size()) << '\n';
}
```

But we don't want to loop like this!
We want asynchronous!

```
mcq_socket mcqSocket("path_to_shm",
   [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
   [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";});

while (true)
   mcqSocket.receive();
```

Multicast Queue: Adding Work Contracts

```
class mcq_socket{
public:
   using message_handler = std::function<void(std::span<char const>)>;
   using loss_handler = std::function<void(std::uint64_t, std::uint64_t)>;

   mcq_socket
   (
      std::string_view shmPath,              // shared memory for the mcast queue
      message_handler messageHandler,
      loss_handler lossHandler,
      work_contract_group & wcg
   );

private:

   void receive();

   mcq_consumer         mcqReceiver_;        // our multicast queue consumer
   message_handler      messageHandler_;     // callback for messages received
   loss_handler         lossHandler_;        // callback for message loss
   work_contract        workContract_;
};
```

Add `work_contract_group&` to ctor
Add a `work_contract` to the class
Move `receive()` to `private`

Multicast Queue: Adding Work Contracts

LIME TRADING

```
mcq_socket::mcq_socket
(
    std::string_view shmPath,
    message_handler messageHandler,
    loss_handler lossHandler,
    work_contract_group & workContractGroup
):
    mcqConsumer_(shmPath), messageHandler_(messageHandler), lossHandler_(lossHandler),
    workContract_(workContractGroup.create_contract([this](){this->receive();}))
{
}


void mcq_socket::receive()
{
    std::array<char, 2048> buf;
    std::span message(buf.data(), buf.size());
    if (mcqConsumer_.pop(message); not message.empty())
        messageHandler_(message);
}
```

Multicast Queue: Adding Work Contracts

```
mcq_socket::mcq_socket
(
    std::string_view shmPath,
    message_handler messageHandler,
    loss_handler lossHandler,
    work_contract_group & workContractGroup
):
    mcqConsumer_(shmPath), messageHandler_(messageHandler), lossHandler_(lossHandler),
    workContract_(workContractGroup.create_contract([this](){this->receive();}))
{
}


void mcq_socket::receive()
{
    std::array<char, 2048> buf;
    std::span message(buf.data()
    if (mcqConsumer_.pop(message
        messageHandler_(message);
}
```

```
class mcq_socket : non_copyable, non_movable{...};
```

Happy now? It's *slideware*! (^:

Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;

mcq_socket mcqSocket("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup);

std::jthread worker([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    });

while (not exit)
{
    // do main thread kind of stuff ...
}

worker.request_stop();
worker.join();
```

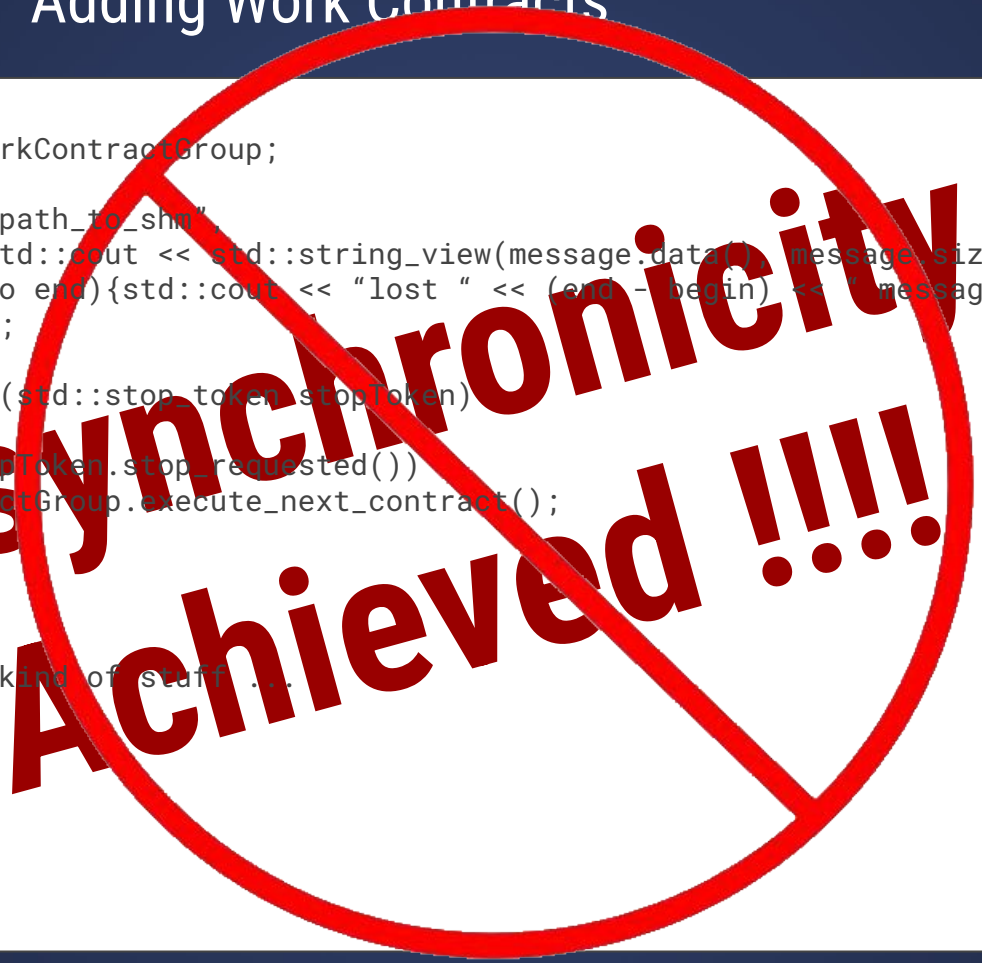
Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;

mcq_socket mcqSocket("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup);

std::jthread worker([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    });

while (not exit)
{
    // do main thread kind of stuff ...
}

worker.request_stop();
worker.join();
```

Asynchronicity Achieved !!!!

Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;

mcq_socket mcqSocket("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup);

std::jthread worker([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    });

while (not exit)
{
    // do main thread kind of stuff ...
}

worker.request_stop();
worker.join();
```



Multicast Queue: Self Scheduling To The Rescue!

```
void mcq_socket::receive()
{
    std::array<char, 2048> buf;
    auto bytesRemaining = 0;

    std::span message(buf.data(), buf.size());
    if (bytesRemaining = mcqConsumer_.pop(message); not message.empty())
        messageHandler_(message);
    if (bytesRemaining > 0)
        this_contract::schedule();
}
```

Hurray for self scheduling!

Multicast Queue: Self Scheduling To The Rescue!

```
void mcq_socket::receive()
{
    std::array<char, 2048> buf;
    auto bytesRemaining = 0;

    std::span message(buf.data(), buf.size());
    if (bytesRemaining = mcq_consumer_.pop(message); not message.empty())
        messageHandler_(message);
    if (bytesRemaining > 0)
        this_contract::schedule();
}
```

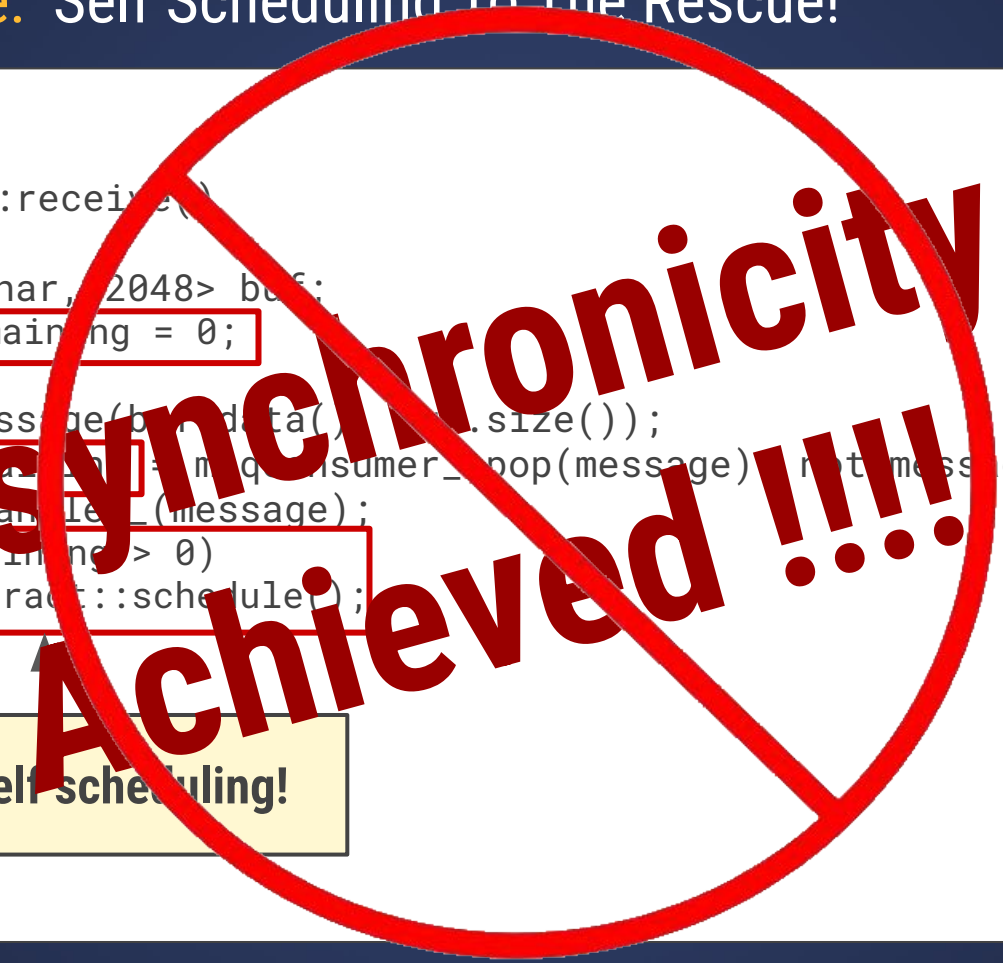
Asynchronicity Achieved !!!!

Hurray for self scheduling!

Multicast Queue: Self Scheduling To The Rescue!

```
void mcq_socket::receive()
{
    std::array<char, 2048> buf;
    auto bytesRemaining = 0;

    std::span message(buf.data(), buf.size());
    if (bytesRemaining = mcq_consumer_.pop(message); not message.empty())
        messageHandler_(message);
    if (bytesRemaining > 0)
        this_contract::schedule();
}
```

Hurry for self scheduling!



Multicast Queue: External Contract Scheduling (via poller)

```
class mcq_socket{
public:
   mcq_socket
   (
      std::string_view shmPath,              // sh
      message_handler messageHandler,        // c
      loss_handler lossHandler,              // callback for lost messages due to mcast lapping
      work_contract_group & wcg,
      mcq_poller & poller
   );

private:
   void receive();
   void on_select() noexcept {workContract_.schedule();}

   mcq_consumer         mcqReceiver_;         // our multicast queue consumer
   message_handler      messageHandler_;      // callback for messages received
   loss_handler         lossHandler_;         // callback for message loss
   work_contract        workContract_;
   mcp_poller &         poller_;
};
```

We need to register with a poller. When poller selects this socket it invokes
`void mcq_socket::on_select() noexcept;`

Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;
mcq_poller mcqPoller;

mcq_socket mcqSocket("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup, mcqPoller);

std::jthread worker([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    });

while (not exit)
    mcqPoller.poll();

worker.request_stop();
worker.join();
```

Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;
mcq_poller mcqPoller;

mcq_socket mcqSocket("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup, mcqPoller);

std::jthread worker([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    });

while (not exit)
    mcqPoller.poll();

worker.request_stop();
worker.join();
```

Victory !!!

Multicast Queue: Adding Work Contracts

```
work_contract_group workContractGroup;
mcq_poller mcqPoller;

std::vector<std::unique_ptr<mcq_socket>> sockets(num_sockets);
for (auto & socket : sockets)
    socket = std::make_unique<mcq_socket>("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup, mcqPoller);

std::vector<std::jthread> workers(num_workers);
for (auto & worker : workers)
    worker = std::move(std::jthread([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    }));

while (not exit)
    mcqPoller.poll();

for (auto & worker : workers){worker.request_stop(); worker.join();}
```

Multicast Queue: Adding Work Contracts

LIME TRADING

```
work_contract_group workContractGroup;
mcq_poller mcqPoller;

std::vector<std::unique_ptr<mcq_socket>> sockets(num_sockets);
for (auto & socket : sockets)
    socket = std::make_unique<mcq_socket>("path_to_shm",
    [](auto message){std::cout << std::string_view(message.data(), message.size()) << '\n';},
    [](auto begin, auto end){std::cout << "lost " << (end - begin) << " messages\n";},
    workContractGroup, mcqPoller);

std::vector<std::jthread> workers(num_workers);
for (auto & worker : workers)
    worker = std::move(std::jthread([](std::stop_token stopToken)
    {
        while (not stopToken.stop_requested())
            workContractGroup.execute_next_contract();
    }));

while (not exit)
    mcqPoller.poll();

for (auto & worker : workers){worker.request_stop(); worker.join();}
```

Where’s the Beef???!!!

Live Demo

```
Executor thread count: 4,  concurrent mutlicast readers: 256

Producer: msg sent = 11000168, send rate: 1.95 MB msg/sec, aggregate message received = 2473.22 MB, aggregate message loss = 0

socket 1: msg recv: 10150611, msg loss: 0        socket 2: msg recv: 10151783, msg loss: 0        socket 3: msg recv: 10150926, msg loss: 0
socket 4: msg recv: 10137060, msg loss: 0        socket 5: msg recv: 10149454, msg loss: 0        socket 6: msg recv: 10151534, msg loss: 0
socket 7: msg recv: 10157358, msg loss: 0        socket 8: msg recv: 10162030, msg loss: 0        socket 9: msg recv: 10158798, msg loss: 0
socket 10: msg recv: 10159342, msg loss: 0       socket 11: msg recv: 10182817, msg loss: 0       socket 12: msg recv: 10149678, msg loss: 0
socket 13: msg recv: 10156398, msg loss: 0       socket 14: msg recv: 10154702, msg loss: 0       socket 15: msg recv: 10154382, msg loss: 0
socket 16: msg recv: 10159598, msg loss: 0       socket 17: msg recv: 10153870, msg loss: 0       socket 18: msg recv: 10153198, msg loss: 0
socket 19: msg recv: 10157006, msg loss: 0       socket 20: msg recv: 10151566, msg loss: 0       socket 21: msg recv: 10154190, msg loss: 0
socket 22: msg recv: 10154894, msg loss: 0       socket 23: msg recv: 10158894, msg loss: 0       socket 24: msg recv: 10151246, msg loss: 0
socket 25: msg recv: 10152686, msg loss: 0       socket 26: msg recv: 10154862, msg loss: 0       socket 27: msg recv: 10152974, msg loss: 0
socket 28: msg recv: 10153806, msg loss: 0       socket 29: msg recv: 10158488, msg loss: 0       socket 30: msg recv: 10150993, msg loss: 0
socket 31: msg recv: 10149809, msg loss: 0       socket 32: msg recv: 10154673, msg loss: 0       socket 33: msg recv: 10156849, msg loss: 0
socket 34: msg recv: 10147985, msg loss: 0       socket 35: msg recv: 10153425, msg loss: 0       socket 36: msg recv: 10151345, msg loss: 0
socket 37: msg recv: 10150321, msg loss: 0       socket 38: msg recv: 10158993, msg loss: 0       socket 39: msg recv: 10158129, msg loss: 0
socket 40: msg recv: 10144945, msg loss: 0       socket 41: msg recv: 10147249, msg loss: 0       socket 42: msg recv: 10147505, msg loss: 0
socket 43: msg recv: 10150353, msg loss: 0       socket 44: msg recv: 10154449, msg loss: 0       socket 45: msg recv: 10151473, msg loss: 0
socket 46: msg recv: 10157969, msg loss: 0       socket 47: msg recv: 10154385, msg loss: 0       socket 48: msg recv: 10152977, msg loss: 0
socket 49: msg recv: 10154449, msg loss: 0       socket 50: msg recv: 10155985, msg loss: 0       socket 51: msg recv: 10162001, msg loss: 0
socket 52: msg recv: 10156839, msg loss: 0       socket 53: msg recv: 10150865, msg loss: 0       socket 54: msg recv: 10157969, msg loss: 0
socket 55: msg recv: 10155722, msg loss: 0       socket 56: msg recv: 10158660, msg loss: 0       socket 57: msg recv: 10159114, msg loss: 0
socket 58: msg recv: 10154571, msg loss: 0       socket 59: msg recv: 10149803, msg loss: 0       socket 60: msg recv: 10157131, msg loss: 0
socket 61: msg recv: 10156081, msg loss: 0       socket 62: msg recv: 10160019, msg loss: 0       socket 63: msg recv: 10161649, msg loss: 0
socket 64: msg recv: 10153811, msg loss: 0       socket 65: msg recv: 10116571, msg loss: 0       socket 66: msg recv: 10114619, msg loss: 0
socket 67: msg recv: 10117819, msg loss: 0       socket 68: msg recv: 10125755, msg loss: 0       socket 69: msg recv: 10118203, msg loss: 0
socket 70: msg recv: 10116507, msg loss: 0       socket 71: msg recv: 10110235, msg loss: 0       socket 72: msg recv: 10108187, msg loss: 0
socket 73: msg recv: 10109883, msg loss: 0       socket 74: msg recv: 10109307, msg loss: 0       socket 75: msg recv: 10077299, msg loss: 0
socket 76: msg recv: 10120187, msg loss: 0       socket 77: msg recv: 10112553, msg loss: 0       socket 78: msg recv: 10122252, msg loss: 0
socket 79: msg recv: 10112876, msg loss: 0       socket 80: msg recv: 10107738, msg loss: 0       socket 81: msg recv: 10112812, msg loss: 0
socket 82: msg recv: 10114348, msg loss: 0       socket 83: msg recv: 10111404, msg loss: 0       socket 84: msg recv: 10114188, msg loss: 0
socket 85: msg recv: 10112940, msg loss: 0       socket 86: msg recv: 10110092, msg loss: 0       socket 87: msg recv: 10108009, msg loss: 0
socket 88: msg recv: 10116620, msg loss: 0       socket 89: msg recv: 10114636, msg loss: 0       socket 90: msg recv: 10113900, msg loss: 0
socket 91: msg recv: 10116268, msg loss: 0       socket 92: msg recv: 10114155, msg loss: 0       socket 93: msg recv: 10110048, msg loss: 0
socket 94: msg recv: 10115424, msg loss: 0       socket 95: msg recv: 10117408, msg loss: 0       socket 96: msg recv: 10113024, msg loss: 0
socket 97: msg recv: 10110560, msg loss: 0       socket 98: msg recv: 10118043, msg loss: 0       socket 99: msg recv: 10115264, msg loss: 0
socket 256: msg recv: 10105819, msg loss: 0
```

The Demo: Source Code Walkthrough

LIME TRADING

```
int main(int argc,char** argv)
{
    // start up a poller which will be used to poll any multicast queues created in this demo.
    mcq_poller poller;
    std::jthread pollerThread([&](auto stopToken)
            {
                while (not stopToken.stop_requested())
                    poller.poll();
            });

    // assign a unique path to the shared memory where our multicast queue will reside
    auto shmPath = std::format("bcpp.{}", std::chrono::system_clock::now().time_since_epoch());

    // start up the multicast queue producer thread
    std::atomic<bool> producerReady = false;
    std::jthread producer([&](auto stopToken)
             {producer_function(stopToken, shmPath, producerReady, begin);});
    while (not producerReady)
        ;
```

The Demo: Source Code Walkthrough

LIME TRADING

```
int main(int argc,char** argv)
{

   // start up a poller which will be used to poll any multicast queues created in this demo.
   mcq_poller poller;
   std::jthread pollerThread([&](auto stopToken)
           {
               while (not stopToken.stop_requested())
                   poller.poll();
           });

   // assign a unique path to the shared memory where our multicast queue will reside
   auto shmPath = std::format("bcpp.{}", std::chrono::system_clock::now().time_since_epoch());

   // start up the multicast queue producer thread
   std::atomic<bool> producerReady = false;
   std::jthread producer([&](auto stopToken)
            {producer_function(stopToken, shmPath, producerReady, begin);});
   while (not producerReady)
       ;
```

The Demo: Source Code Walkthrough

LIME TRADING

```
bcpp::work_contract_group workContractGroup;

// create a bunch of multicast queue receiver sockets (multicast queue consumers)
std::vector<std::unique_ptr<mcq_socket>> messageReceivers(numConsumers);
for (auto index = 0; auto & messageReceiver : messageReceivers)
{
    messageReceiver = std::move(std::make_unique<mcq_socket>(
        mcq_socket::event_handlers{.receiveHandler_ = on_receive_message,
        .lossHandler_ = on_receive_loss},
        shmPath, index++, workContractGroup, poller));
}

// create the thread pool which will power the work contract group
std::vector<std::jthread> workerThreads(numWorkers);
for (auto & executor : workerThreads)
        executor = std::move(std::jthread([&](auto stopToken)
        {
            while (not stopToken.stop_requested())
                workContractGroup.execute_next_contract();
        }));
```

The Demo: Source Code Walkthrough

LIME TRADING

```
bcpp::work_contract_group workContractGroup;

// create a bunch of multicast queue receiver sockets (multicast queue consumers)
std::vector<std::unique_ptr<mcq_socket>> messageReceivers(numConsumers);
for (auto index = 0; auto & messageReceiver : messageReceivers)
{
    messageReceiver = std::move(std::make_unique<mcq_socket>(
        mcq_socket::event_handlers{.receiveHandler_ = on_receive_message,
        .lossHandler_ = on_receive_loss},
        shmPath, index++, workContractGroup, poller));
}
```

```
// create the thread pool which will power the work contract group
std::vector<std::jthread> workerThreads(numWorkers);
for (auto & executor : workerThreads)
        executor = std::move(std::jthread([&](auto stopToken)
        {
            while (not stopToken.stop_requested())
                workContractGroup.execute_next_contract();
        }));
```

The Demo: Source Code Walkthrough

LIME TRADING

```
// signal all to begin after short delay to ensure threads are all started up
std::this_thread::sleep_for(std::chrono::milliseconds(100));
begin = true;

// Run until Ctrl-C
while (not terminate)
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
```

“If you can’t explain it simply, then you don’t understand it well enough.”

- Albert Einstein

Work Contracts:

Fast relief for blocking pain and tail-latency flare-ups

“For years, mutex dependency kept me from enjoying my life to the fullest. But then I discovered Work Contracts. Now I’m wait-free, and my concurrency has never been better! Thanks, Work Contracts!”

Warning: Do not take Work Contracts if you are allergic to fast code, believe that mutexes build character, or have a history of wait-free anxiety. Side effects may include a sudden loss of contention, increased productivity spikes, accusations that you photoshopped your flattened latency tails, an urge to do more performance profiling and an irrational disdain for blocking calls. In rare cases, codebases may achieve zero contention. If you experience any of these symptoms, then good for you - keep going. Ask your project manager if Work Contracts is right for you.

Resources:

Source Code:

github.com/buildingcpp/work_contract

Lime Trading:

Lime.co

Contact:

wc@michael-maniscalco.com

Work Contract Talk From CppCon 2024:

www.youtube.com/watch?v=oj-_v[ZNMVw