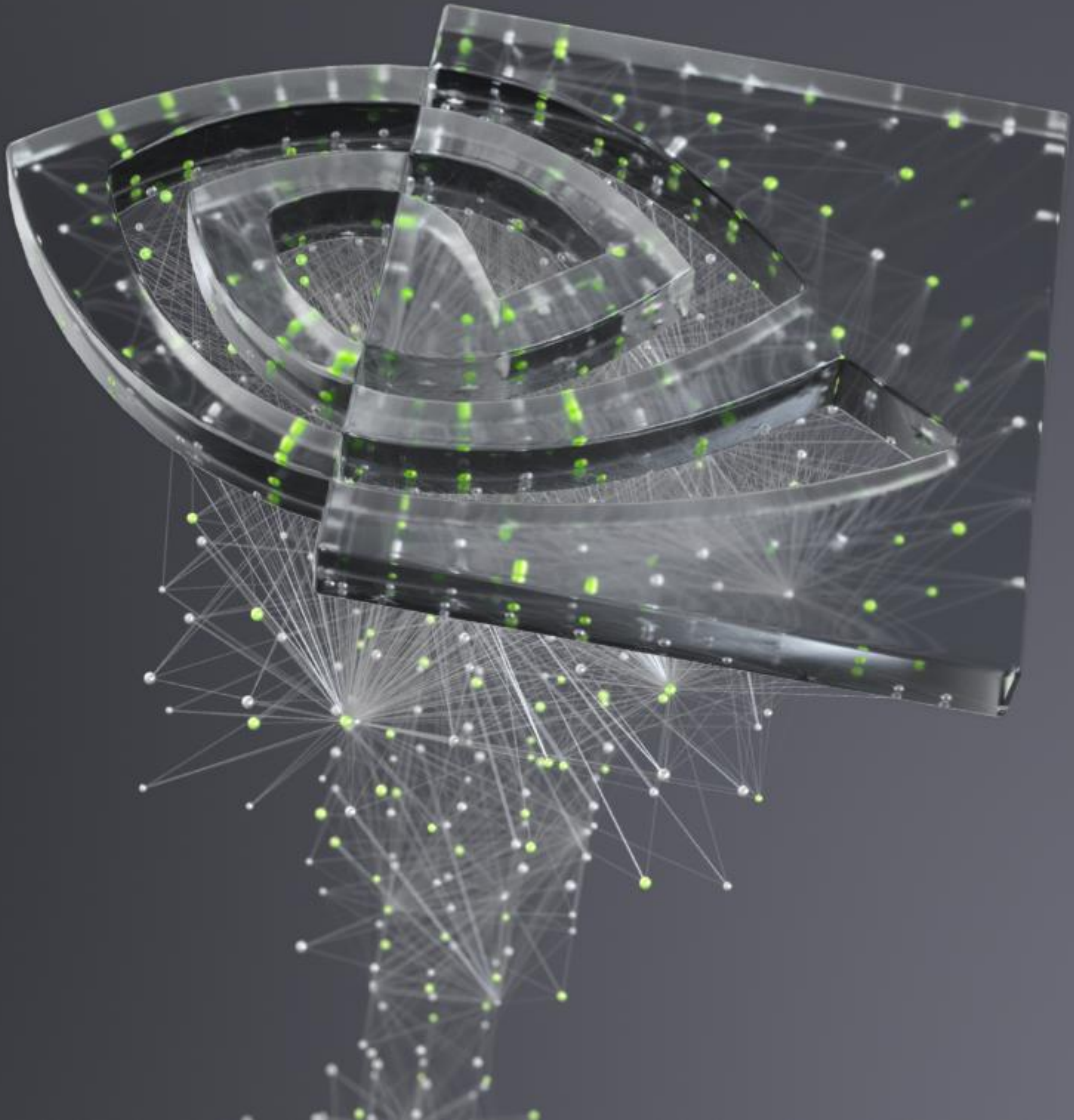
NVIDIA | ACADEMY



ASYNC EXECUTION IN STANDARD C++

Eric Niebler, September 21 2021

THE STANDARD EXECUTORS EFFORT

Mission: Deliver in standard C++…

… a portable abstraction of a compute resource,

… customizable async algorithms that are parameterized on that abstraction, and

… optionally, some standard compute resources:

- Event loop
- Static thread pool
- Access to the system execution context.

Portable abstraction requirements:

- Allows launching work on arbitrary compute resources
- Gives a way to chain additional work in an optimal, customizable fashion (e.g., for GPU)
- Values, errors, and cancellation signals can be propagated through a chain
- Can block for work to finish

P2300: STD::EXECUTION

Proposes:

- A set of concepts that represent:
  - A handle to a compute resource (*aka*, `scheduler`)
  - A unit of lazy async work (*aka*, `sender`)
  - A completion handler (*aka*, `receiver`)
- A small, initial set of generic async algorithms:
  - *E.g.*, `then`, `when_all`, `sync_wait`, `let_*`
- Utilities for integration with C++20 coroutines

INTRO TO P2300: EXAMPLE

EXAMPLE: LAUNCHING CONCURRENT WORK

```
namespace ex = std::execution;

int compute_intensive(int);

int main() {
  unifex::static_thread_pool pool{8};
  ex::scheduler auto sched = pool.get_scheduler();

  ex::sender auto work =
    ex::when_all(
      ex::then(ex::schedule(sched), [] { return compute_intensive(0); }),
      ex::then(ex::schedule(sched), [] { return compute_intensive(1); }),
      ex::then(ex::schedule(sched), [] { return compute_intensive(2); })
    );

  auto [a, b, c] = std::this_thread::sync_wait( std::move(work) ).value();
}
```

Launch three tasks to execute concurrently on a custom execution context

EXAMPLE: LAUNCHING CONCURRENT WORK

```
namespace ex = std::execution;

int compute_intensive(int);

int main() {
  unifex::static_thread_pool pool{8};
  ex::scheduler auto sched = pool.get_scheduler();

  ex::sender auto work =
    ex::when_all(
      ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
    );

  auto [a, b, c] = std::this_thread::sync_wait( std::move(work) ).value();
}
```

Use pipe syntax if you want to.

EXAMPLE: LAUNCHING CONCURRENT WORK

```
namespace ex = std::execution;

int compute_intensive(int);

int main() {
  unifex::static_thread_pool pool{8};
  ex::scheduler auto sched = pool.get_scheduler();

  ex::sender auto work =
    ex::when_all(
      ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
    );

  auto [a, b, c] = std::this_thread::sync_wait( std::move(work) ).value();
}
```

P2300 proposes these concepts and algorithms, among others.

EXAMPLE: LAUNCHING CONCURRENT WORK

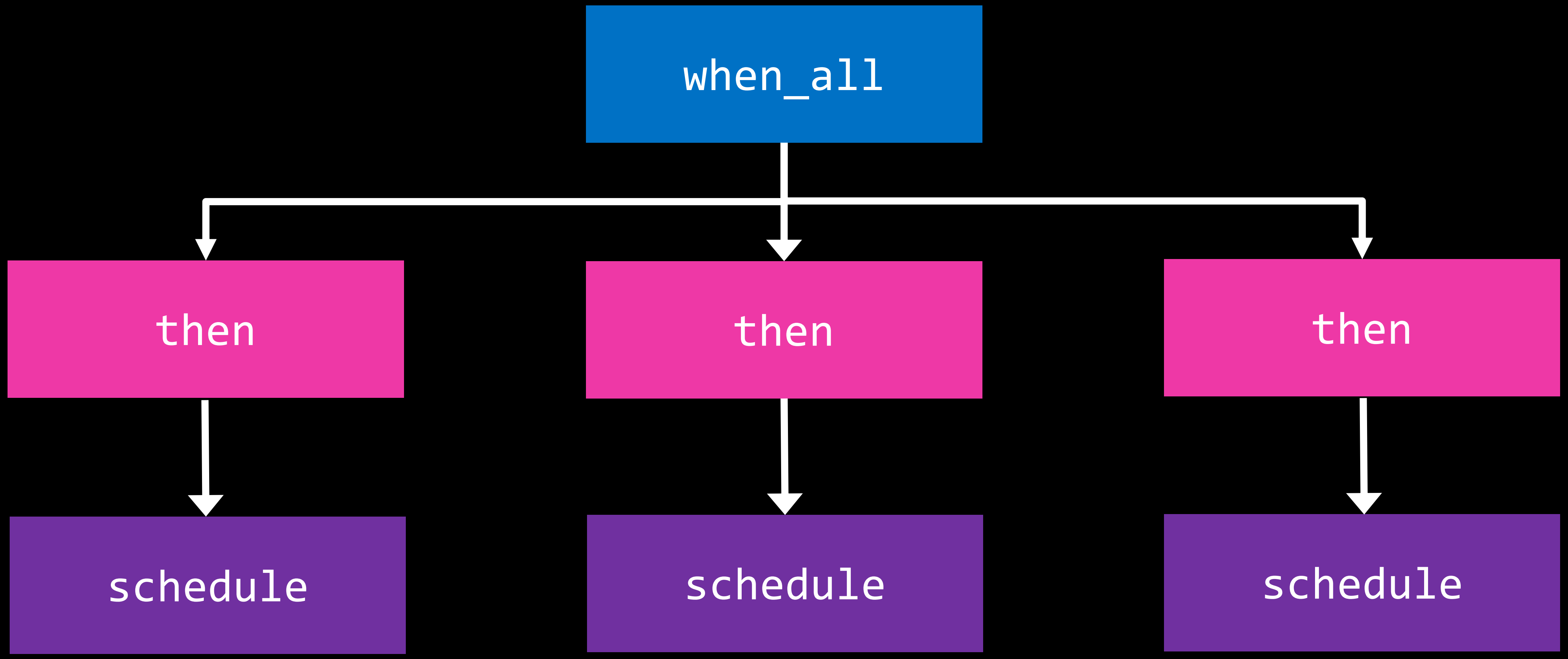
```
namespace ex = std::execution;

int compute_intensive(int);

int main() {
  unifex::static_thread_pool pool{8};
  ex::scheduler auto sched = pool.get_scheduler();

  ex::sender auto work =
    ex::when_all(
      ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
    );

  auto [a, b, c] = std::this_thread::sync_wait( std::move(work) ).value();
}
```

Zero allocations here

EXAMPLE: LAUNCHING CONCURRENT WORK

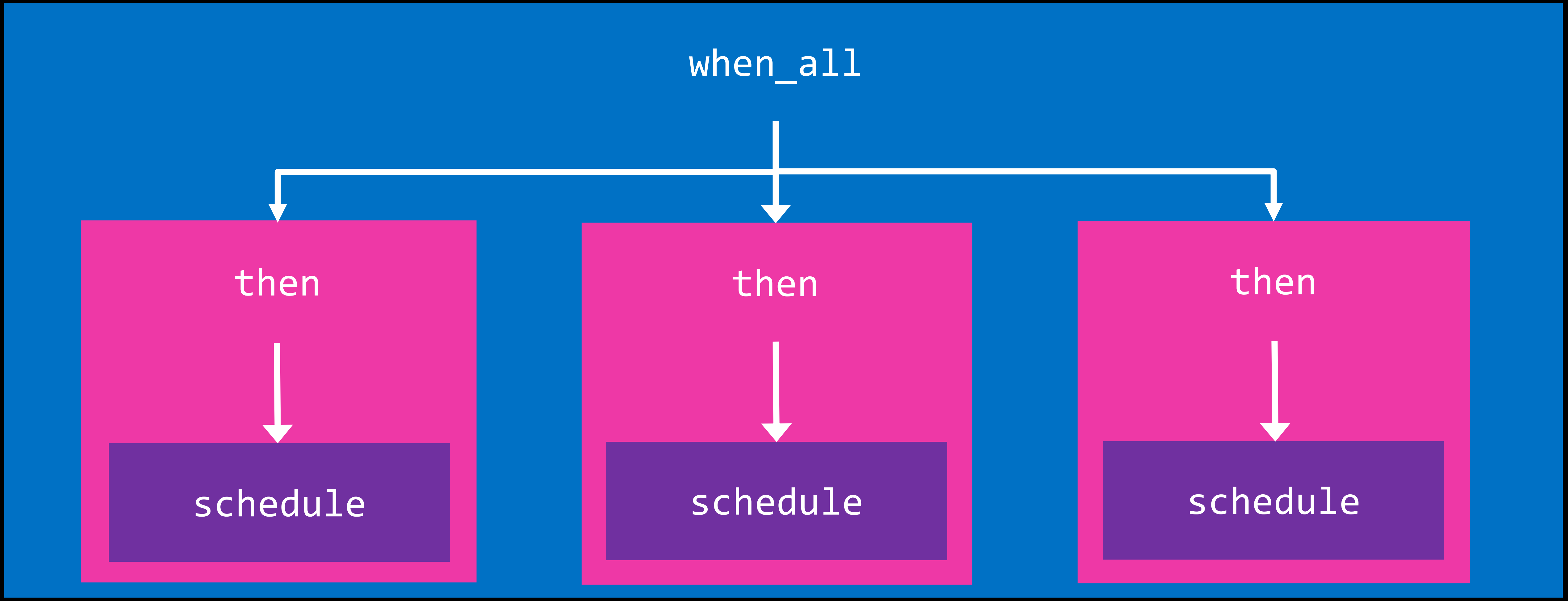
```
namespace ex = std::execution;

int compute_intensive(int);

int main() {
  unifex::static_thread_pool pool{8};
  ex::scheduler auto sched = pool.get_scheduler();

  ex::sender auto work =
    ex::when_all(
      ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
      ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
    );

  auto [a, b, c] = std::this_thread::sync_wait( std::move(work) ).value();
}
```

SENDERS ARE EXPRESSION TEMPLATES



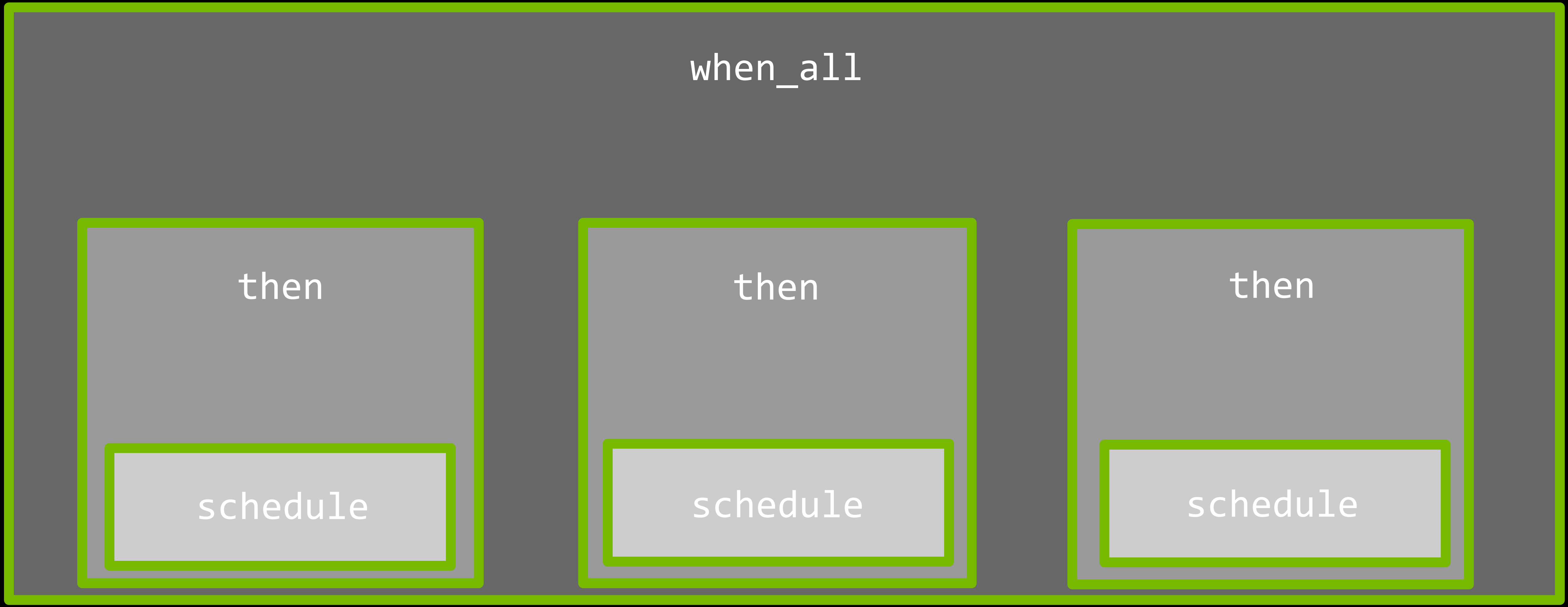
```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

SENDERS ARE EXPRESSION TEMPLATES



```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

THESE EXPRESSIONS ARE ALL SENDERS



```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

Schedulers produce senders

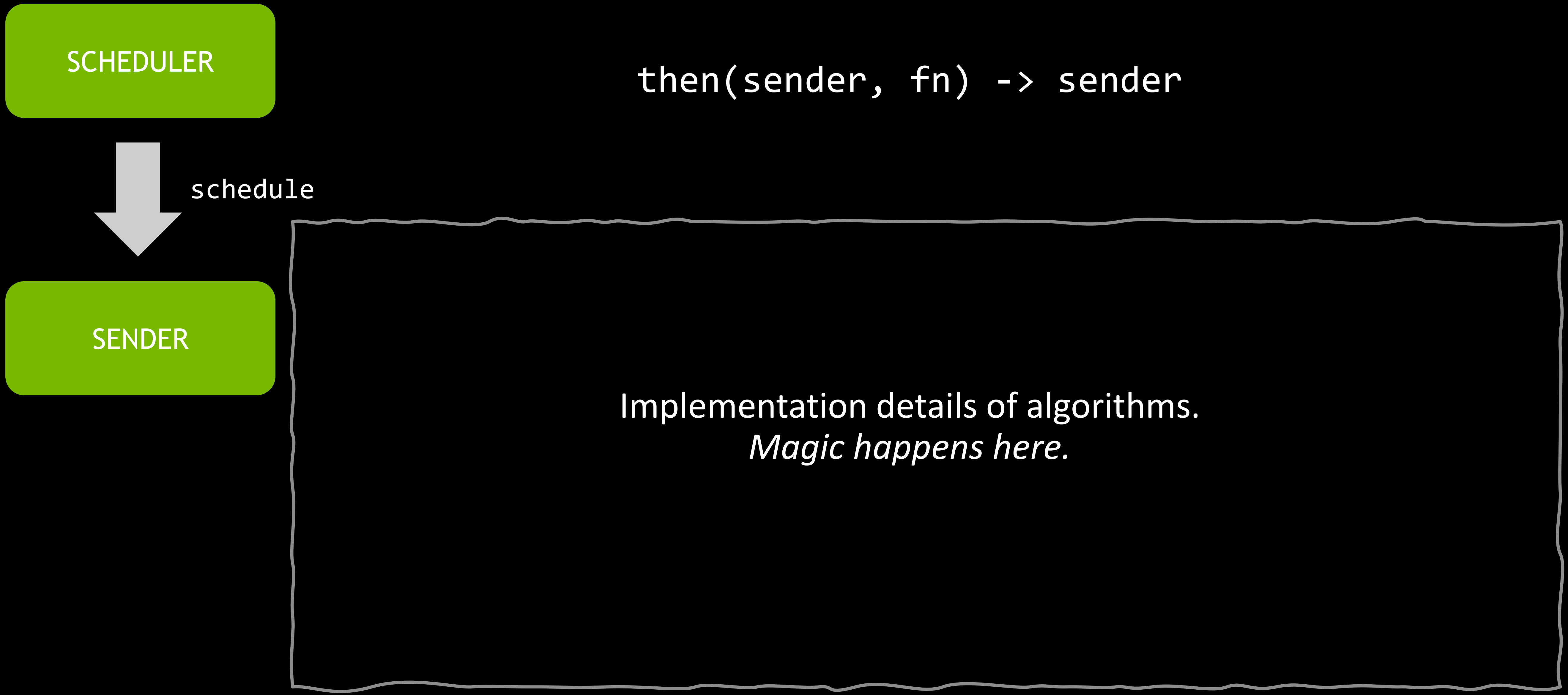
Generic async algorithms accept and return senders

SENDER ADAPTORS OF STD::EXECUTION

Sender adaptors	Returns a sender that…
`then(sender, fn) → sender`	… transforms the result value of the operation with a function.
`upon_[error|done](sender, fn) → sender`	… transforms the error and done signals with a function.
`let_[value|…](sender, fn) → sender`	… passes the result of input sender to function, which returns new sender.
`when_all(senders...) → sender`	… completes when all the input senders complete.
`on(scheduler, sender) → sender`	… starts the input sender in the context of the input scheduler.
`into_variant(sender) → sender`	… packages all possible results of input sender into a variant of tuples.
`bulk(sender, size, fn) → sender`	… launches a bulk operation.
`split(sender) → sender`	… permits multiple receivers to be connected (forks the execution graph).
`done_as_optional(sender) → sender`	… commutes the done signal into a nullopt.
`done_as_error(sender, error) → sender`	… commutes the done signal into an error.

SENDER/RECEIVER CONTROL FLOW

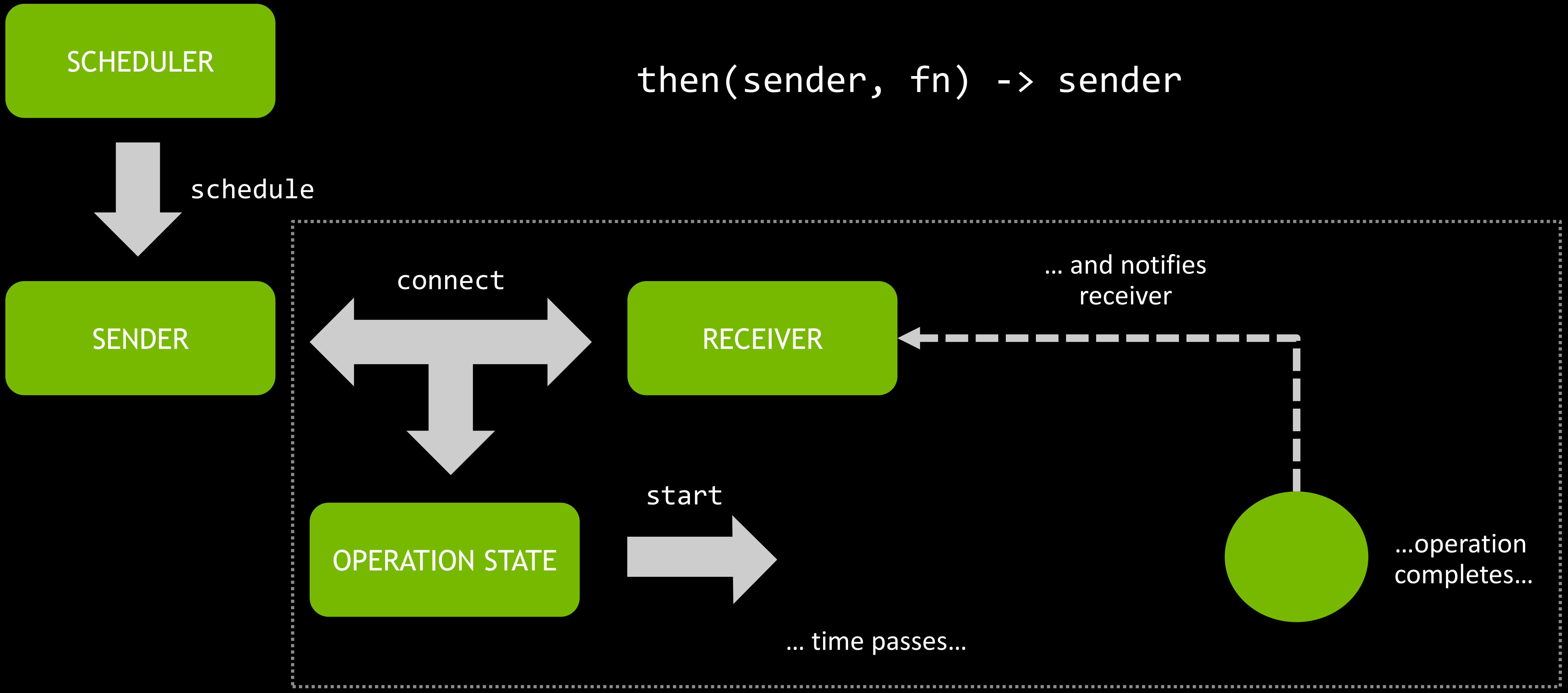
BASIC LIFETIME OF AN ASYNC OPERATION



`then(sender, fn) -> sender`

Implementation details of algorithms.
Magic happens here.

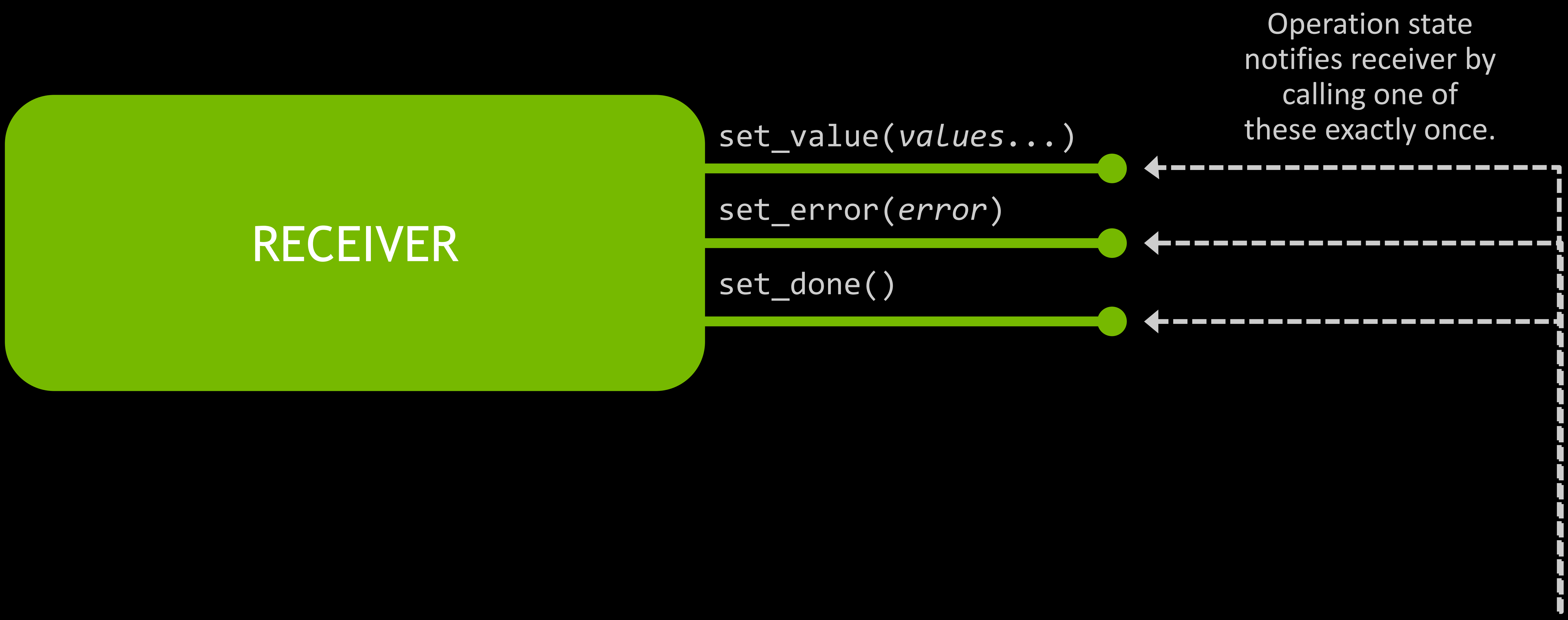
BASIC LIFETIME OF AN ASYNC OPERATION



```
then(sender, fn) -> sender
```

Implementation details of some algorithm; e.g., `then`.

SHAPE OF A RECEIVER


RECEIVER
set_value(*values...*)
set_error(*error*)
set_done()
Operation state notifies receiver by calling one of these exactly once.

CONCEPTUAL BUILDING BLOCKS OF P2300

```
concept scheduler:
  schedule(scheduler) → sender;
```

```
concept sender:
  connect(sender, receiver) → operation_state;
```

```
concept receiver:
  set_value(receiver, Values...) → void;
  set_error(receiver, Error) → void;
  set_done(receiver) → void;
```

```
concept operation_state:
  start(operation_state) → void;
```

ALGORITHM EXAMPLE: THEN

ALGORITHM EXAMPLE: THEN

```cpp
namespace stdexec = std::execution;

template<stdexec::sender S, class F>
stdexec::sender auto then(S s, F f)
{
  return _then_sender{{}, (S&&) s, (F&&) f};
}
```

ALGORITHM EXAMPLE: THEN

```
  return _then_sender{{}, (S&&) s, (F&&) f};
}

template<stdexec::sender S, class F>
struct _then_sender : stdexec::sender_base
{
  S s_;
  F f_;

  // Implement connect(sender, receiver) -> operation_state:
  template<stdexec::receiver R>
    requires stdexec::sender_to<S, _then_receiver<R, F>>
  decltype(auto) connect(R r) &&
  {
    return stdexec::connect(
        (S&&) s_, _then_receiver<R, F>{(R&&) r, (F&&) f_});
  }
};
```

ALGORITHM EXAMPLE: THEN

```
template<receiver R, class F>
struct _then_receiver
{
  R r_;
  F f_;

  // Implement set_value(Values...) by invoking the callable and
  // passing the result to the inner receiver:
  template<class... As>
    requires stdexec::receiver_of<R, std::invoke_result_t<F, As...>>
  void set_value(As&&... as) &&
  {
     stdexec::set_value((R&&) r_, std::invoke((F&&) f_, (As&&) as...));
  }

  // Implement set_error(Values...) and set_done() as pass-throughs:
  template<class Err>
    requires stdexec::receiver<R, E>
  void set_error(Err&& err) && noexcept
  {
     stdexec::set_error((R&&) r_, (Err&&) err);
  }
```

ALGORITHM EXAMPLE: THEN

```cpp
namespace stdexec = std::execution;

template<stdexec::sender S, class F>
stdexec::sender auto then(S s, F f)
{
  return _then_sender{{}, (S&&) s, (F&&) f};
}



template<stdexec::sender S, class F>
struct _then_sender : stdexec::sender_base
{
  S s_;
  F f_;

  // Implement connect(sender, receiver) -> operation_state:
  template<stdexec::receiver R>
    requires stdexec::sender_to<S, _then_receiver<R, F>>
  decltype(auto) connect(R r) &&
  {
    return stdexec::connect(
        (S&&) s_,
        _then_receiver<R, F>{(R&&) r, (F&&) f_});
  }
};
```

```cpp
template<receiver R, class F>
struct _then_receiver
{
  R r_;
  F f_;

  // Implement set_value(Values...) by invoking the callable and
  // passing the result to the inner receiver:
  template<class... As>
    requires stdexec::receiver_of<R, std::invoke_result_t<F, As...>>
  void set_value(As&&... as) &&
  {
    stdexec::set_value((R&&) r_, std::invoke((F&&) f_, (As&&) as...));
  }

  // Implement set_error(Values...) and set_done() as pass-throughs:
  template<class Err>
    requires stdexec::receiver<R, E>
  void set_error(Err&& err) && noexcept
  {
    stdexec::set_error((R&&) r_, (Err&&) err);
  }

  void set_done() && noexcept
  {
    stdexec::set_done((R&&) r_);
  }
};
```

SENDER/RECEIVER AND COROUTINES

AWAITABLES AS SENDERS

```
// This is a coroutine:
unifex::task<int> read_socket_async(socket, span<char, 1024>);


int main()
{
  socket s = /*...*/;
  char buff[1024];

  auto read = read_socket_async(s, buff);

  auto [cbytes] =
      std::this_thread::sync_wait( std::move( read ) ).value();
}
```

All awaitable types are senders and can be passed to any async algorithm that accepts a sender.

No extra allocation or synchronization is required.

SENDERS AS AWAITABLES

```
// This is a coroutine:
unifex::task<int> read_socket_async(socket, span<char, 1024>);


unifex::task<void> concurrent_read_async(socket s1, socket s2)
{
  char buff1[1024];
  char buff2[1024];

  auto [cbytes1, cbytes2] =
    co_await std::execution::when_all(
        read_socket_async(s1, buff1),
        read_socket_async(s2, buff2)
    ) | into_tuple();

  /*...*/
}
```

Most senders can be made awaitable in a coroutine type trivially.

No extra allocation or synchronization is necessary.

SENDERS AS AWAITABLES

```
// This is a coroutine task type:
struct task
{
  struct promise_type :
    std::execution::with_awaitable_senders<promise_type>
  {
    /*...*/
```

To make senders awaitable within a coroutine type, derive its promise type from with_awaitable_senders.

COROUTINES AND CANCELLATION

- If an awaited sender completes by calling `set_done()`, it behaves as though an uncatchable "exception" has been thrown. (The stack of awaiting coroutines is unwound.)
- The cancellation exception is "caught" by applying a sender adaptor that translates "done" into a value or an error before awaiting the sender; *e.g.*, with:
  - `std::execution::done_as_optional()`
  - `std::execution::done_as_error()`.
- When the cancellation exception reaches an awaitable/sender boundary, it is automatically translated back into a call of `set_done()`.

SENDER/RECEIVER AND COROUTINES

By returning a sender, an async function puts the choice of whether to use coroutines or not in the hands of the caller.

SENDER/RECEIVER FIELD EXPERIENCE

SENDER/RECEIVER FIELD EXPERIENCE

Sender/receiver as specified in P2300R2 is currently being used in the following shipping Facebook products:

- Facebook Messenger on iOS, Android, Windows, and macOS
- Instagram on iOS and Android
- Facebook on iOS and Android
- Portal
- An internal Facebook product that runs on Linux

It’s safe to say the number of monthly users of sender/receiver-based async APIs number in the billions.

“NVIDIA is fully invested in P2300 senders/receivers; we believe it is the correct basis for portable asynchronous and heterogeneous execution in Standard C++ across all platforms, including CPUs, GPUs, and other accelerators, both manufactured by us and others. We are developing an implementation P2300 that we plan to ship in production sometime next year.”

SENDER/RECEIVER EXPERIMENTATION AT BLOOMBERG

Dietmar Kuehl has been teaching sender/receiver internally at Bloomberg with positive result.

He is experimenting with sender/receiver socket-based networking. His code is public at https://github.com/dietmarkuehl/kuhllib/ and includes an echo server example.

"… Different and new: yes. Complex? I don't think it is more complex than what is already being used. Especially with coroutine integration I think an "end user" use will be as simple as any of the other alternatives but a "library implementer" use, packaging up solution ("algorithms"), remains possible - unlike any of the other approaches I'm aware of."

SUMMARY

SUMMARY: THE STANDARD EXECUTORS EFFORT

Mission: Deliver in standard C++…

… a portable abstraction of a compute resource, ✓

… customizable async algorithms that are parameterized on that abstraction, and ✓

… optionally, some standard compute resources:

- Event loop
- Static thread pool
- Access to the system execution context.

Portable abstraction requirements:

- Allows launching work on arbitrary compute resources ✓
- Gives a way to chain additional work work in a optimal, customizable fashion (e.g., for GPU) ✓
- Values, errors, and cancellation signals can be propagated through a chain ✓
- Can block for work to finish ✓
- Can work with zero allocations ✓
- Tight integration with C++20 coroutines ✓

ADDITIONAL RESOURCES

P2300R2: “std::execution”:

https://wiki.edg.com/pub/Wg21telecons2021/LibraryEvolutionWorkingGroup/P2300R2.html

Libunifex:

https://github.com/facebookexperimental/libunifex