Working with Asynchrony Generically:

A Tour of C++ Executors

ERIC NIEBLER

Cppcon
The C++ Conference
2021
October 24-29

TALK OUTLINE

Part 1:

1. Goals for the Executors proposal
2. Some simple examples, intro to senders
3. The lifecycle of an async operation
4. Under the hood of a concurrent operation
5. Implementing a simple algorithm
6. Senders and coroutines

Part 2:

1. Structured concurrency
2. Cancellation
3. An extended example

GOALS FOR THE EXECUTORS PROPOSAL

The vision: “An asynchronous analog of the STL”

- A full suite of standard async algorithms based on real-world requirements:

  E.g., `then`, `when_all`, `sync_wait`, `repeat`, `stop_when`, `timeout`, *etc*. (not all proposed yet)

- A standard set of abstractions (*aka*, concepts) derived from the algorithms

- Efficient interoperability with coroutines

- An open and extensible way to specify *where*, *how*, and *when* work should happen

  … including some standard ones: an event loop, portable access to the system execution context, nursery for spawned work

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

Example 1:
Launching concurrent work

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

libunifex: https://github.com/facebookexperimental/libunifex

EXAMPLE: LAUNCHING CONCURRENT WORK

```cpp
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();
}
```

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();
}
```

Use pipe syntax if you want to.

EXAMPLE: LAUNCHING CONCURRENT WORK

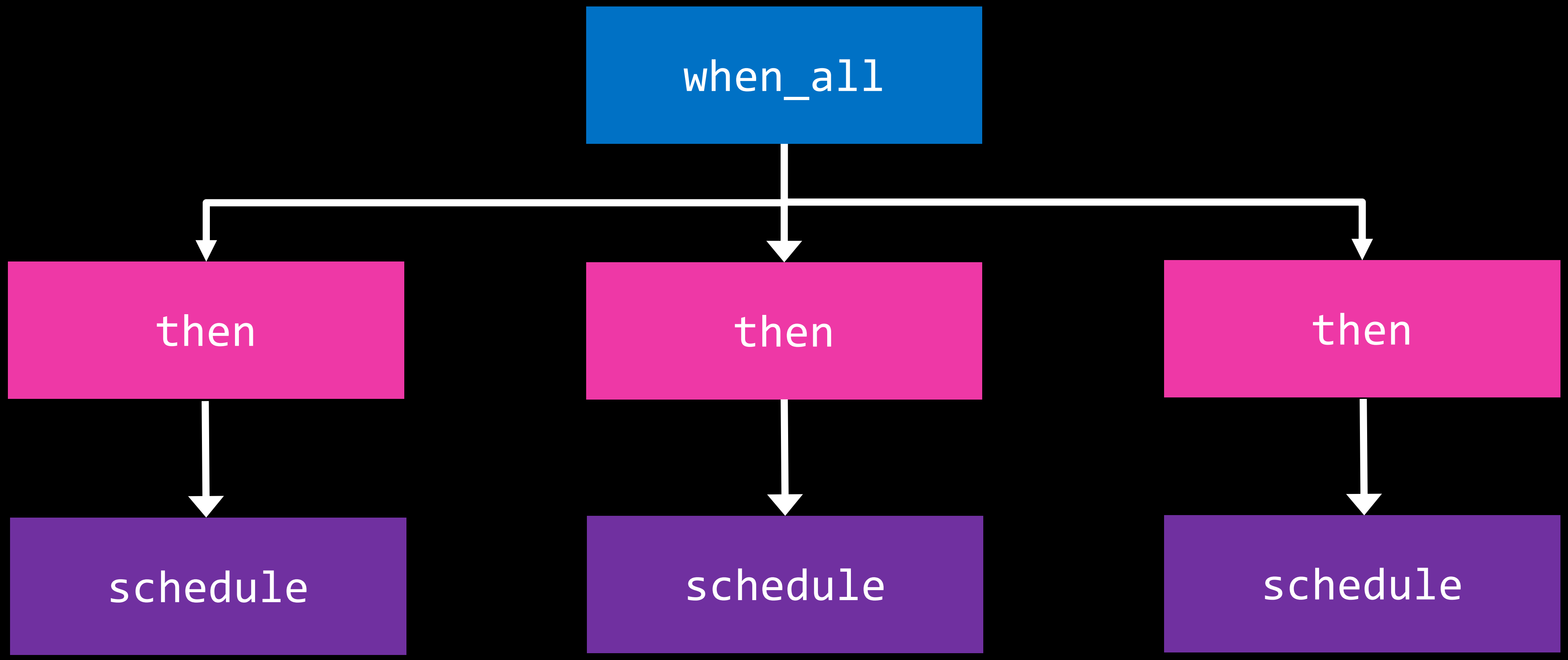
```
namespace ex = std::execution;

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();
}
```

Zero allocations here

EXAMPLE: LAUNCHING CONCURRENT WORK

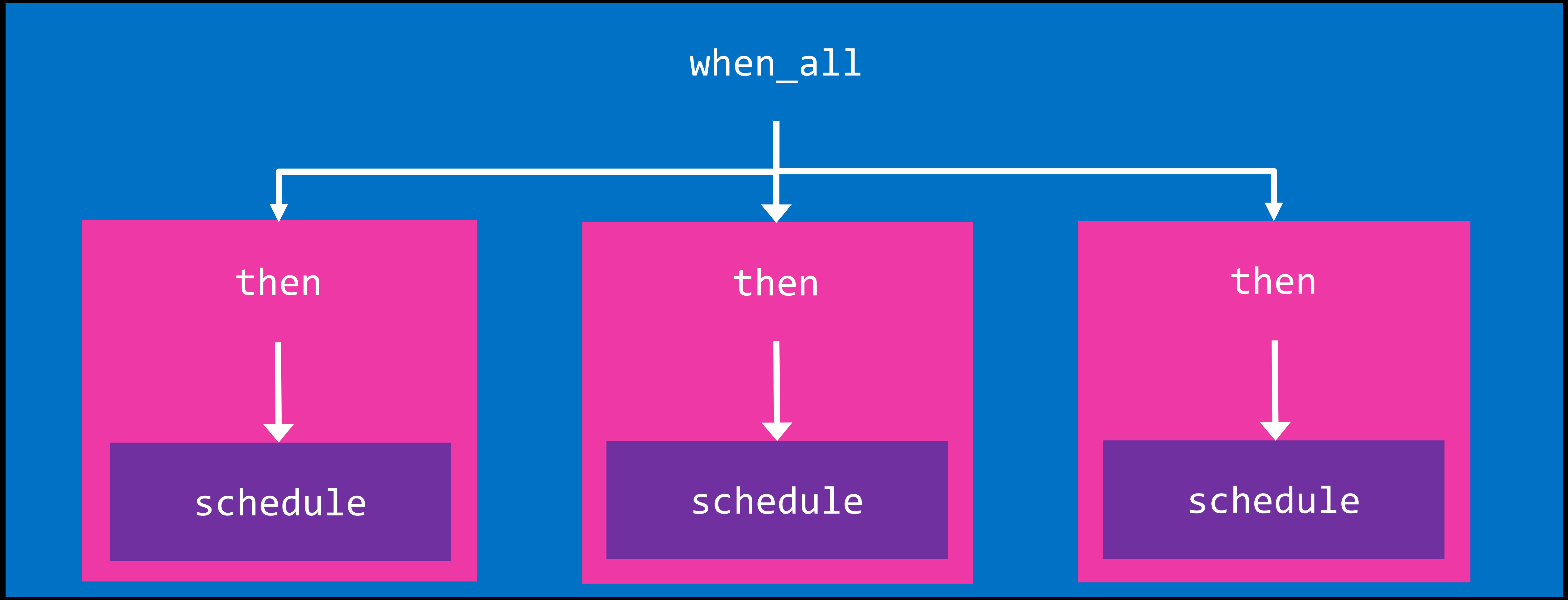
```cpp
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

SENDERS ARE EXPRESSION TEMPLATES



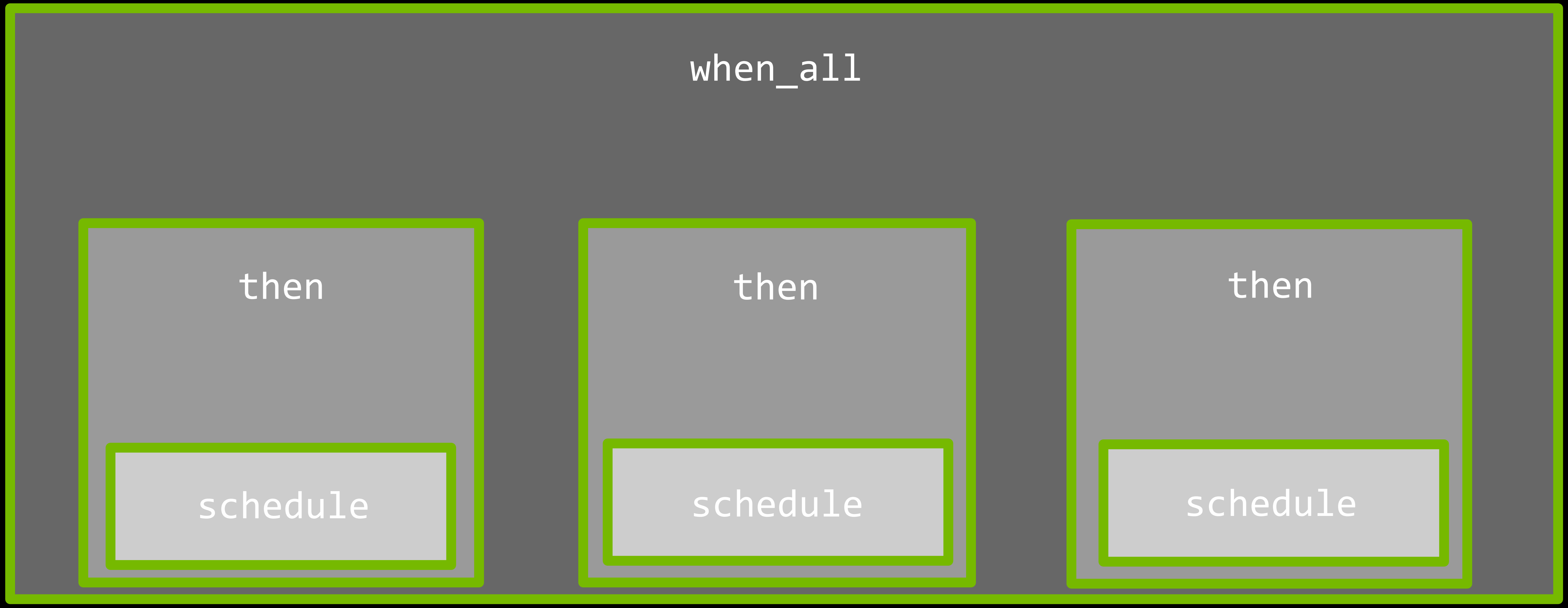

```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

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); })
  );
```

Example 2:
Transitioning execution context

EXAMPLE: TRANSITIONING EXECUTION CONTEXT

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

ex::sender auto accept_request();
ex::sender auto process_request(request_t);

extern unifex::static_thread_pool low_latency;
extern unifex::static_thread_pool workers;

ex::sender auto accept_and_process_requests() {
  return
      ex::on(low_latency.get_scheduler(), accept_request())
    | ex::transfer(workers.get_scheduler())
    | ex::then([](auto request) { process_request(request); })
    | unifex::repeat_effect();
}
```

Accept requests on low-latency threads. Process the requests on the worker threads.

EXAMPLE: TRANSITIONING EXECUTION CONTEXT

```
namespace ex = std::execution;

ex::sender auto accept_request();
ex::sender auto process_request(request_t);

extern unifex::static_thread_pool low_latency;
extern unifex::static_thread_pool workers;

unifex::task<void> accept_and_process_requests() {
  while (true) {
    auto request =
      co_await ex::on(low_latency.get_scheduler(), accept_request());

    co_await ex::on(workers.get_scheduler(), process_request(request));
  }
}
```

Or write it as a coroutine.

Schedulers produce senders

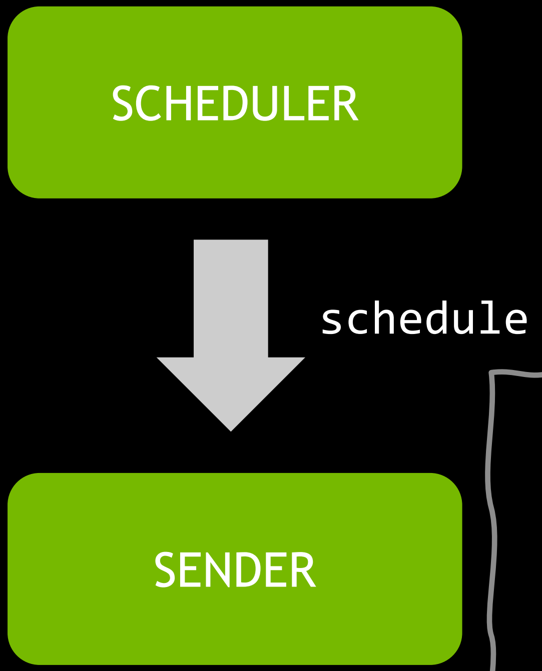
Generic async algorithms
accept and return senders

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

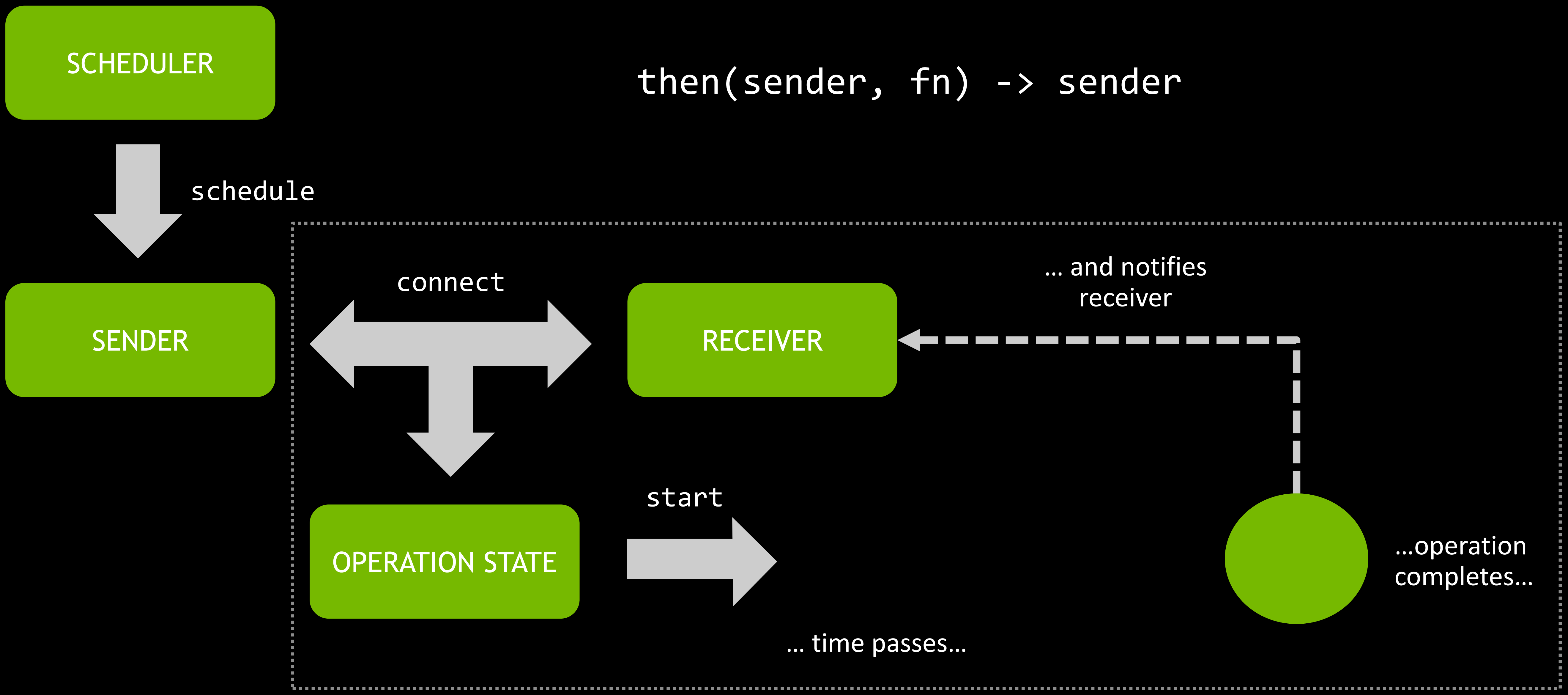
BASIC LIFETIME OF AN ASYNC OPERATION



`then(sender, fn) -> sender`

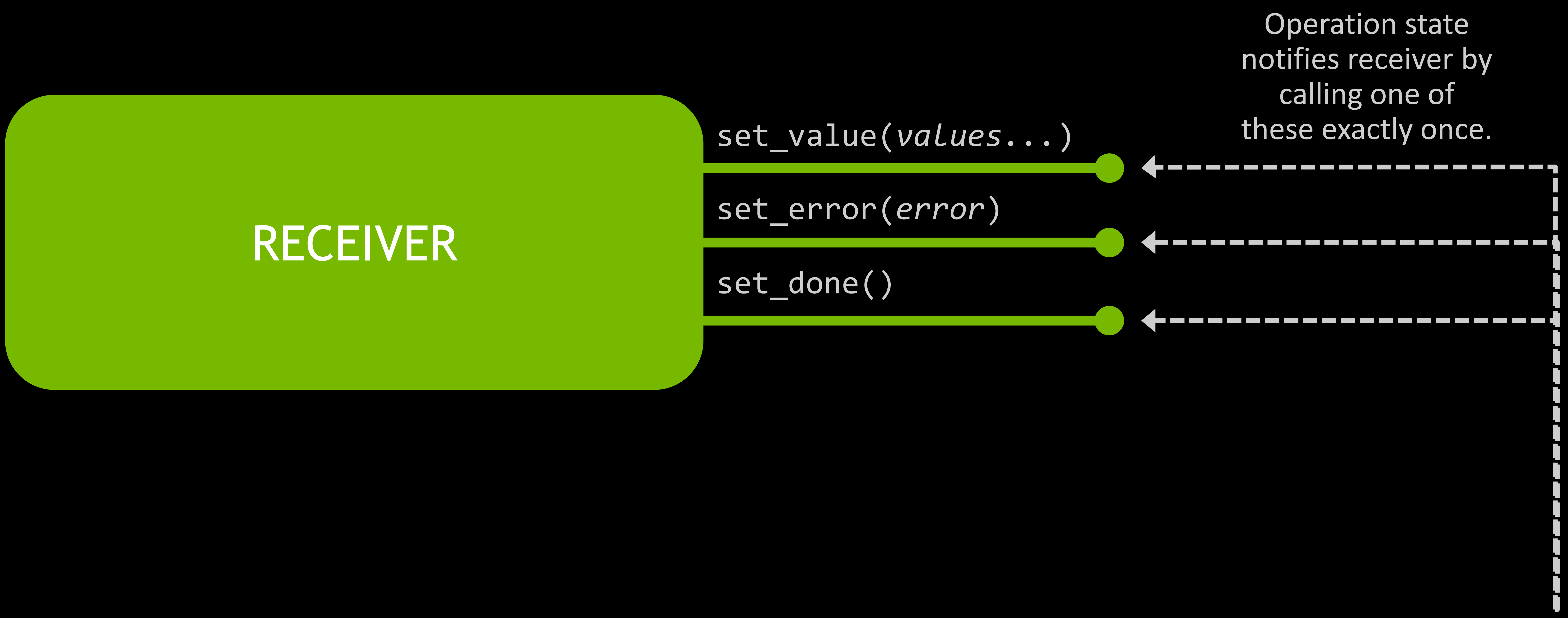
Implementation details of algorithms.
Magic happens here.

BASIC LIFETIME OF AN ASYNC OPERATION



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

SHAPE OF A RECEIVER



CONCEPTUAL BUILDING BLOCKS OF P2300

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

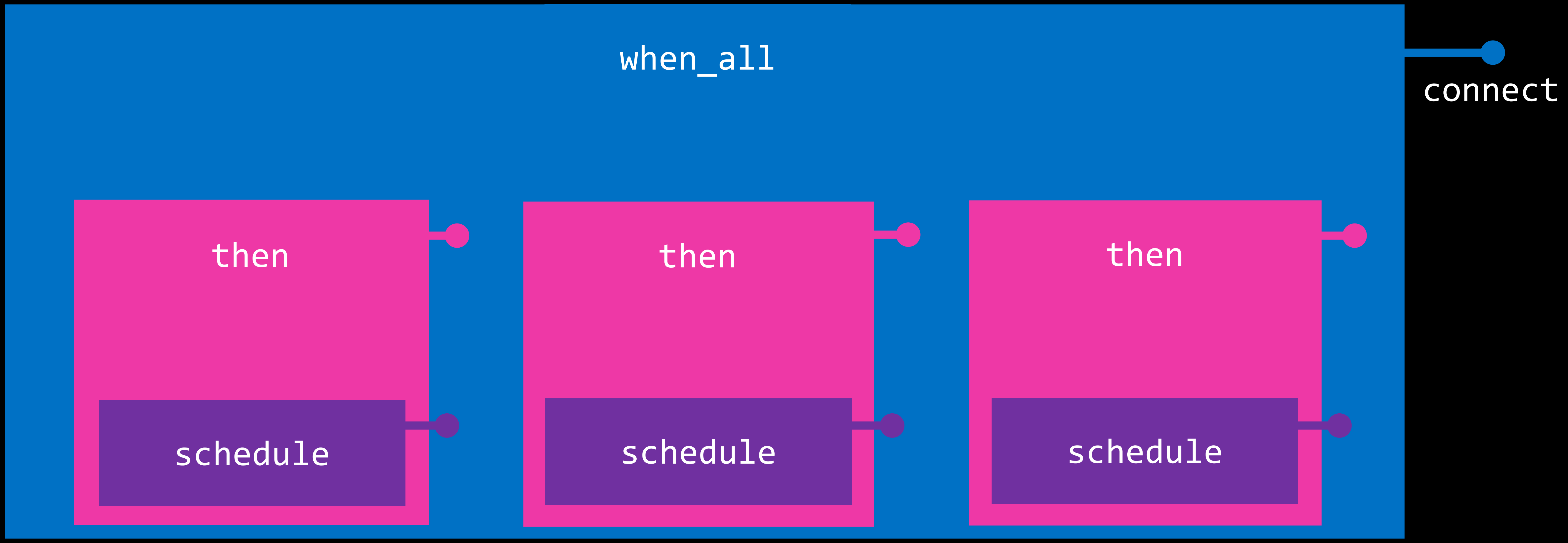
```
concept sender:
  connect(sender, receiver) → operation_state;
```

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

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

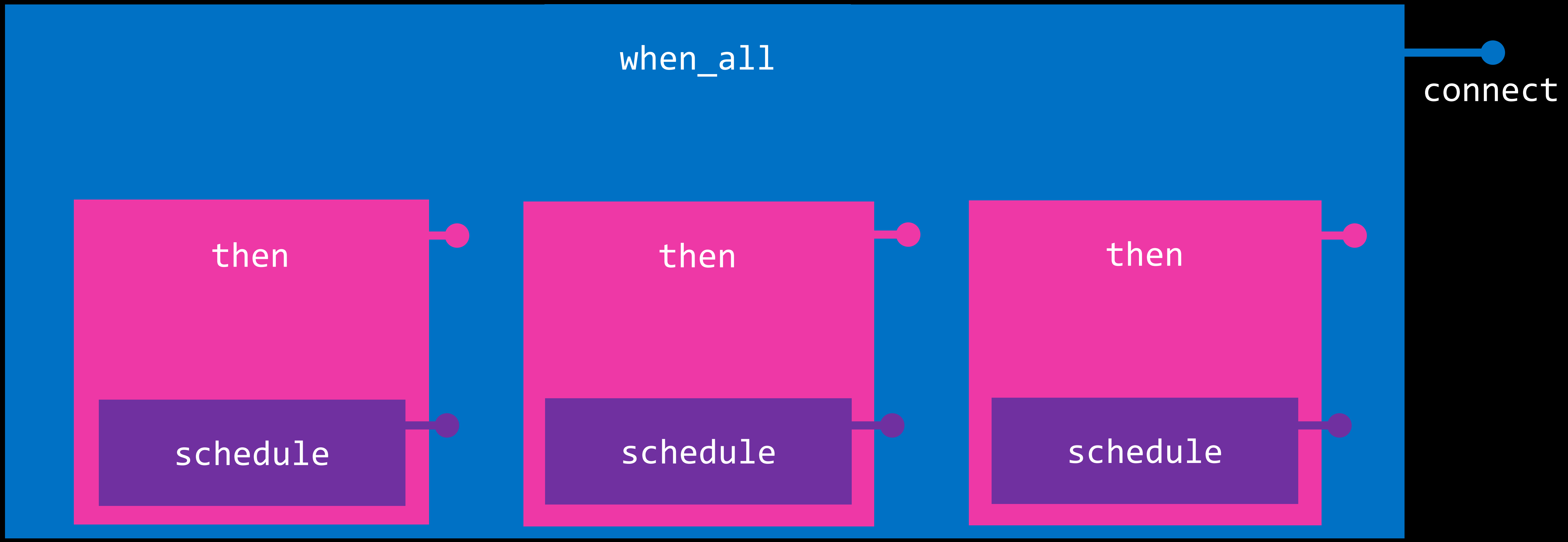
Under the hood of a concurrent operation

ALL OF THESE SENDERS IMPLEMENT CONNECT



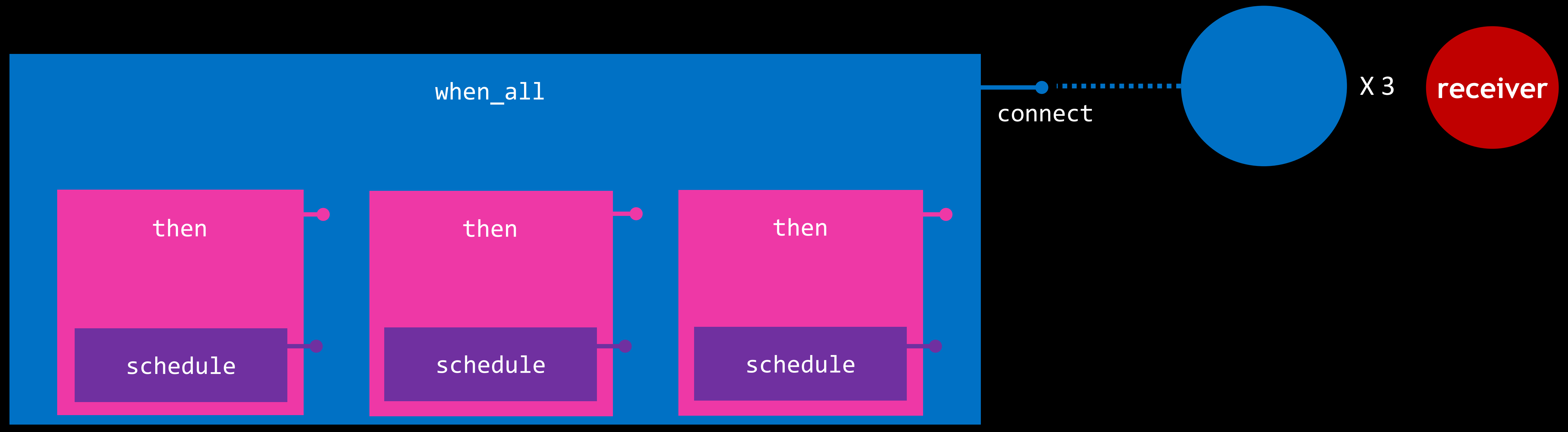

```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

ALL OF THESE SENDERS IMPLEMENT CONNECT

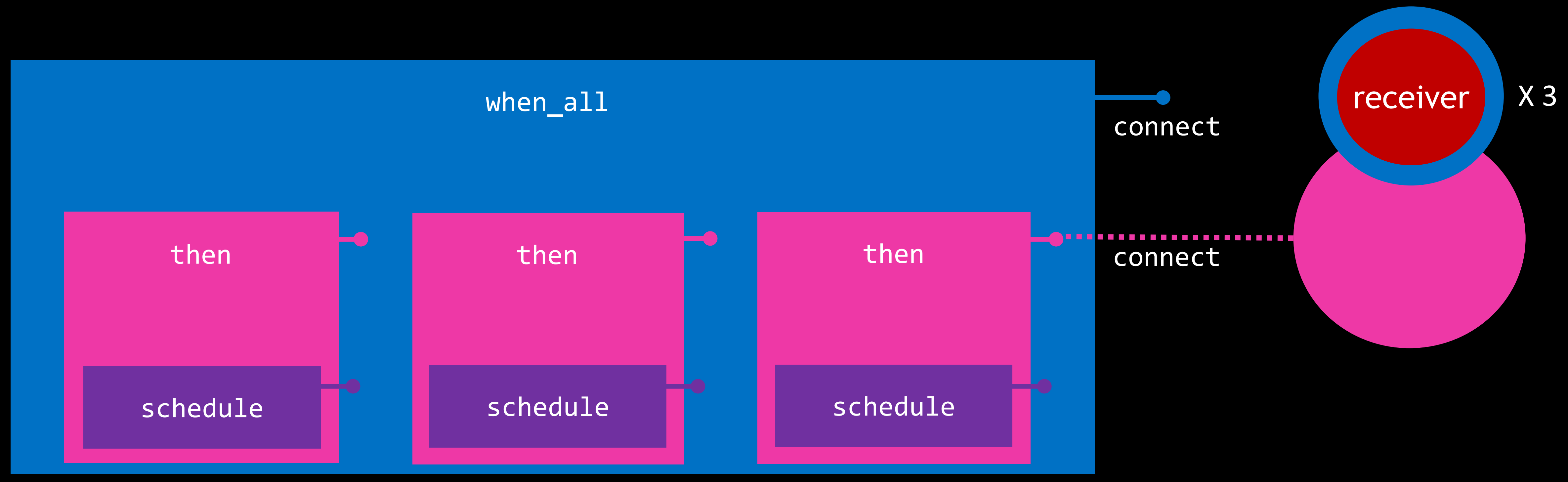



```
ex::sender auto work =
  ex::when_all(
    ex::schedule(sched) | ex::then([] { return compute_intensive(0); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(1); }),
    ex::schedule(sched) | ex::then([] { return compute_intensive(2); })
  );
```

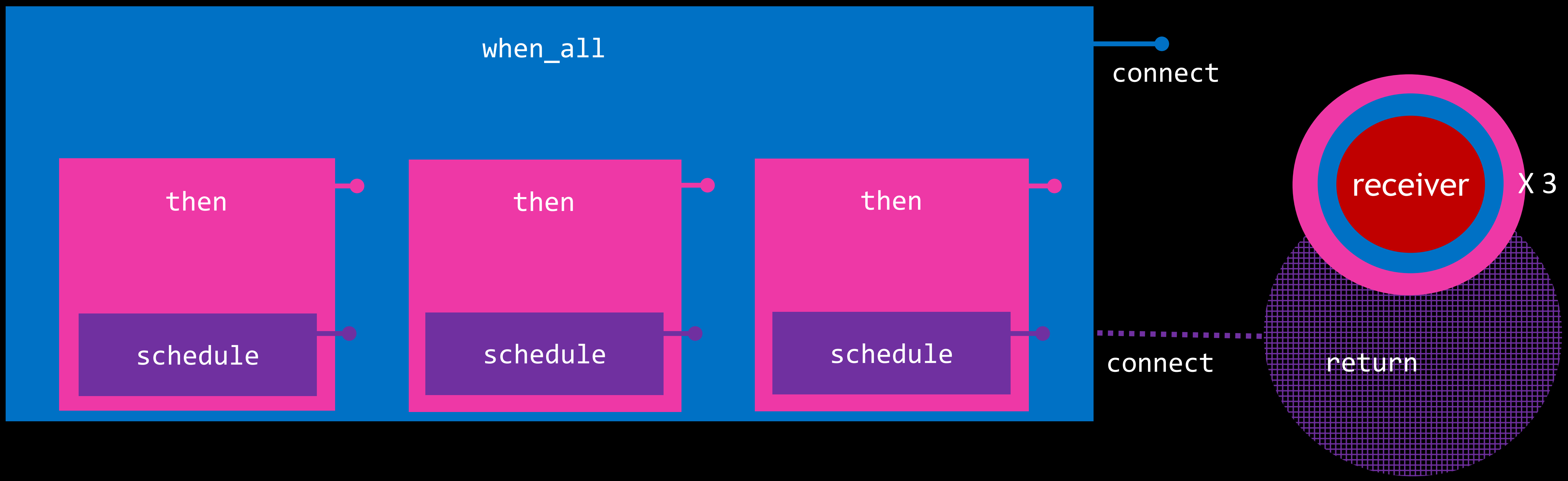
CONNECT ENRICHES RECEIVER AND RECURSES INTO CHILDREN



CONNECT ENRICHES RECEIVER AND RECURSES INTO CHILDREN

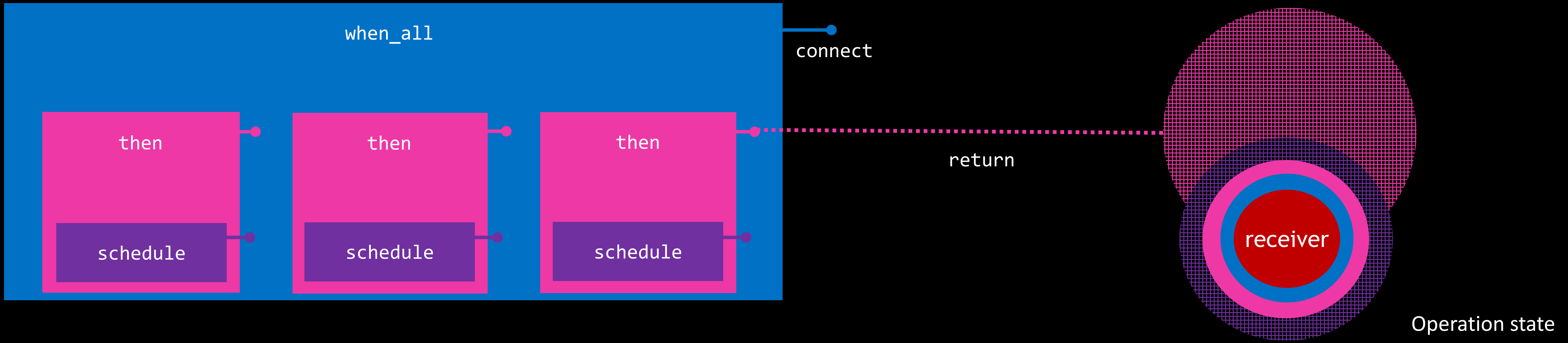


CONNECT RETURNS AN OPERATION STATE

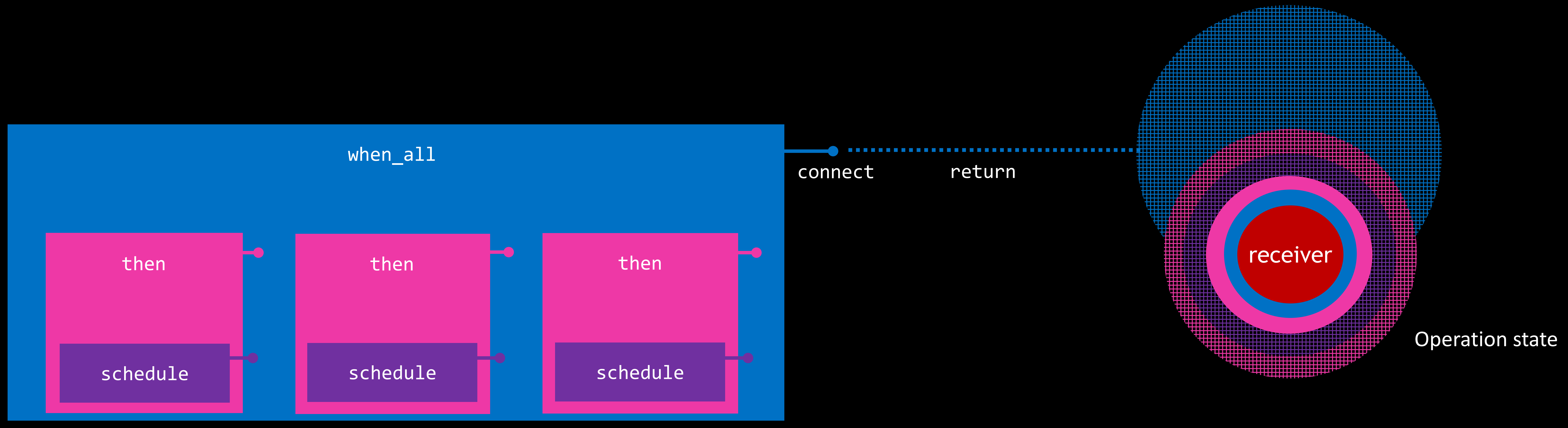


Operation state

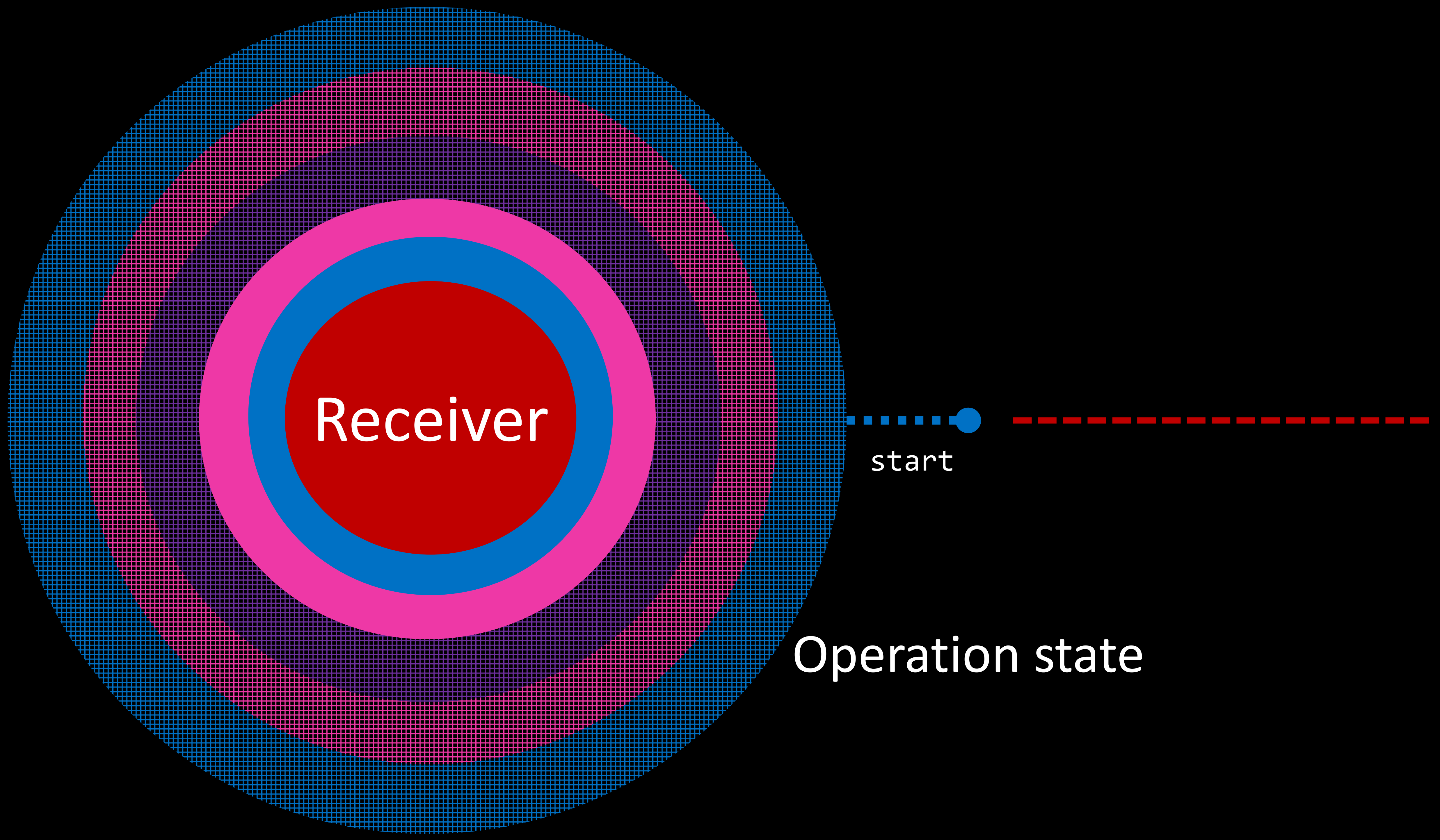
CONNECT RETURNS AN OPERATION STATE



CONNECT RETURNS AN OPERATION STATE



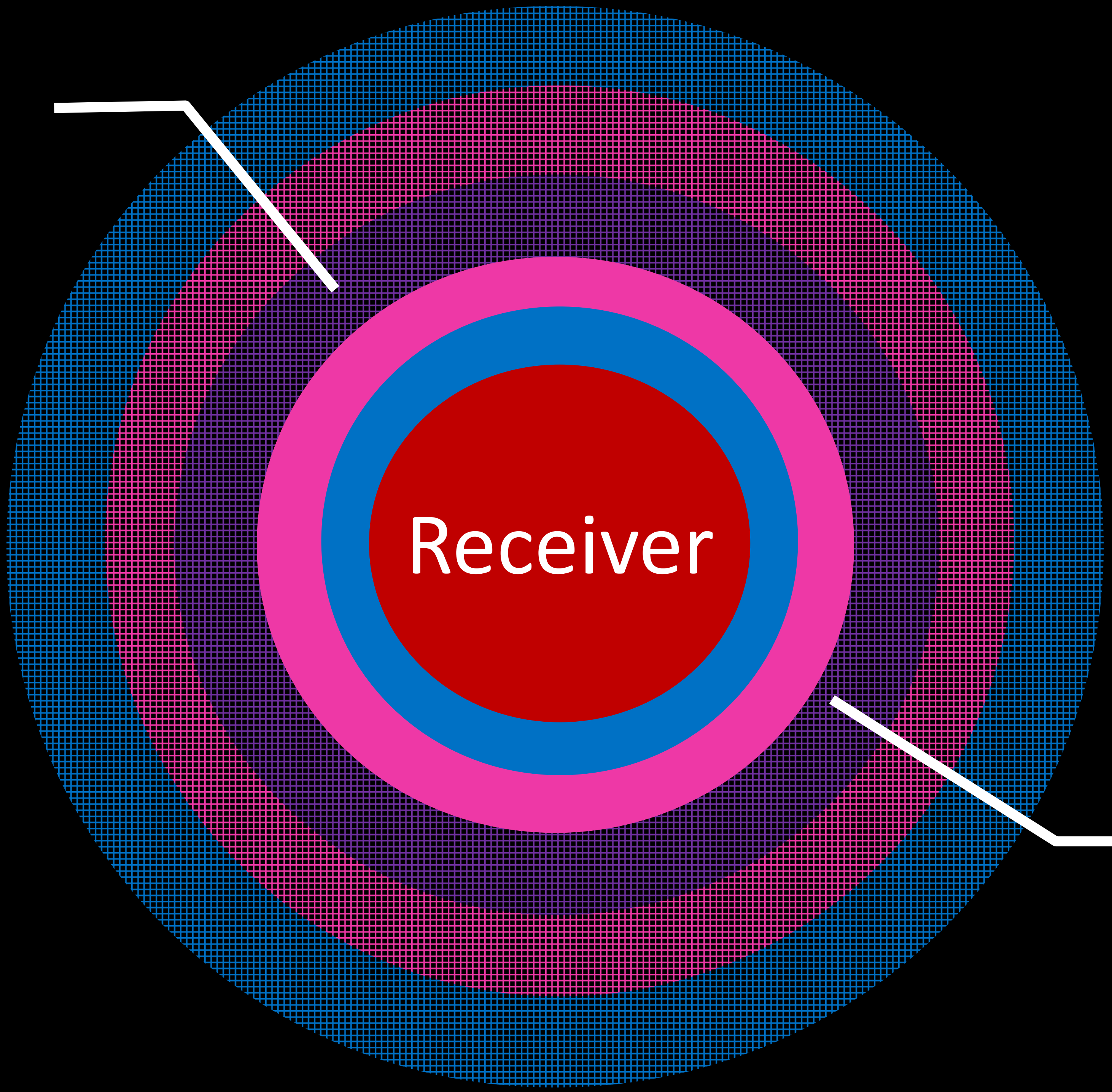
OPERATIONS EXECUTE OUTSIDE-IN



OPERATIONS EXECUTE OUTSIDE-IN

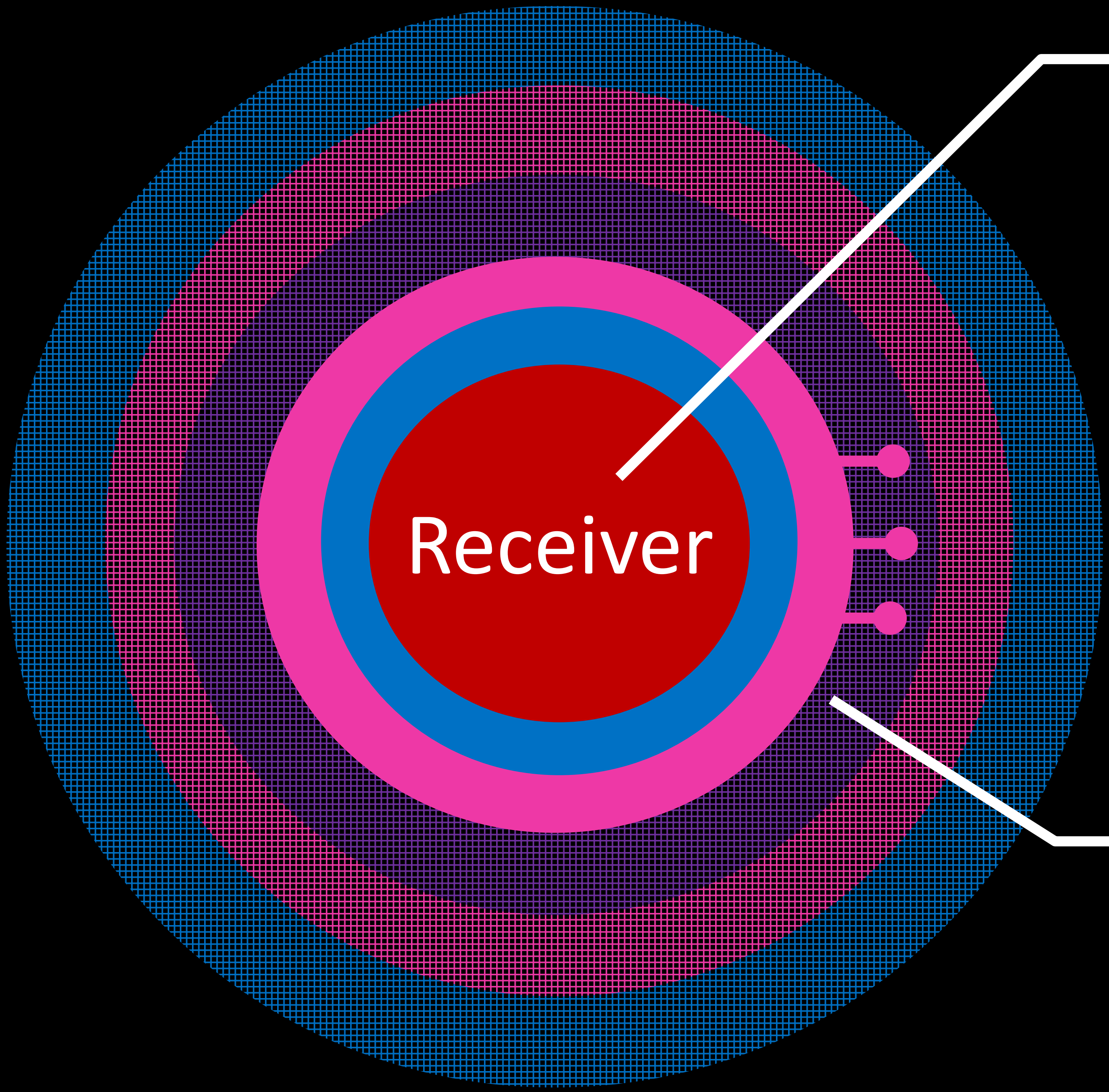
Work is scheduled on the thread pool

Time passes....



A thread in the pool picks up the work and executes it

OPERATIONS EXECUTE OUTSIDE-IN



Inner-most receiver is notified that the work is complete

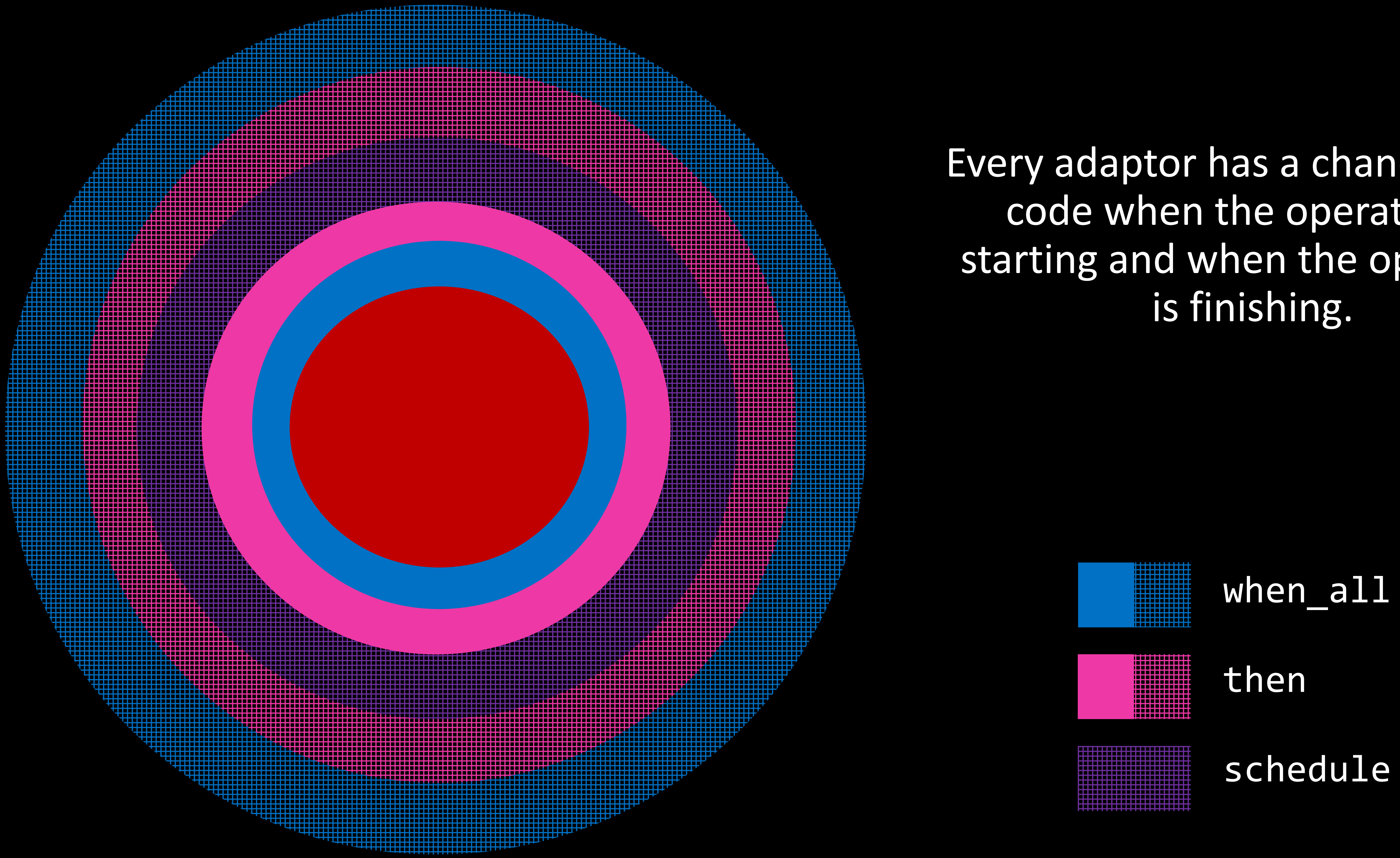
`set_value`

`set_error`

`set_done`

A thread in the pool picks up the work and executes it

OPERATIONS EXECUTE OUTSIDE-IN



Every adaptor has a chance to run code when the operation is starting and when the operation is finishing.

Senders nest

Receivers nest

Operation states nest

IMPLEMENTING AN ALGORITHM:
THEN

ALGORITHM EXAMPLE: THEN

```
namespace ex = std::execution;

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

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

ALGORITHM EXAMPLE: THEN

```
template<ex::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 ex::receiver_of<R, std::invoke_result_t<F, As...>>
  void set_value(As&&... as) &&
  {
     ex::set_value((R&&) r_, std::invoke((F&&) f_, (As&&) as...));
  }

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

ALGORITHM EXAMPLE: THEN

```
namespace ex = std::execution;

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



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

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

```
template<ex::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 ex::receiver_of<R, std::invoke_result_t<F, As...>>
  void set_value(As&&... as) &&
  {
     ex::set_value((R&&) r_, std::invoke((F&&) f_, (As&&) as...));
  }

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

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

SENDERS 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 [cbytes] = std::this_thread::sync_wait( read_socket_async(s, buff) ).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:
template< class T >
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 the done signal 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()`.

FLASHBACK: TRANSITIONING EXECUTION CONTEXT

If accept_request() completes by calling set_done(), the rest of the coroutine doesn't execute.

```
unifex::task<void> accept_and_process_requests() {
  while (true) {
    auto request =
      co_await ex::on(low_latency.get_scheduler(), accept_request());

    co_await ex::on(workers.get_scheduler(), process_request(request));
  }
}
```

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();
}
```

EXAMPLE: LAUNCHING CONCURRENT WORK

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

int compute_intensive(int);

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

  auto compute = [=](int i) -> unifex::task<int> {
      co_await ex::schedule(sched) | unifex::complete_inline();
      co_return compute_intensive(i);
    };

  ex::sender auto work = ex::when_all(compute(0), compute(1), compute(2));

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

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.

COMING UP IN THE NEXT HOUR:

Structured concurrency

Cancellation support in sender/receiver

Extended example: Sender/receiver and ranges

ADDITIONAL RESOURCES

P2300R2: “std::execution”:
https://wg21.link/P2300R2

Libunifex:
https://github.com/facebookexperimental/libunifex

Working with Asynchrony Generically:

A Tour of C++ Executors

ERIC NIEBLER

Cppcon
The C++ Conference

2021
October 24-29

SUMMARY FROM PART 1

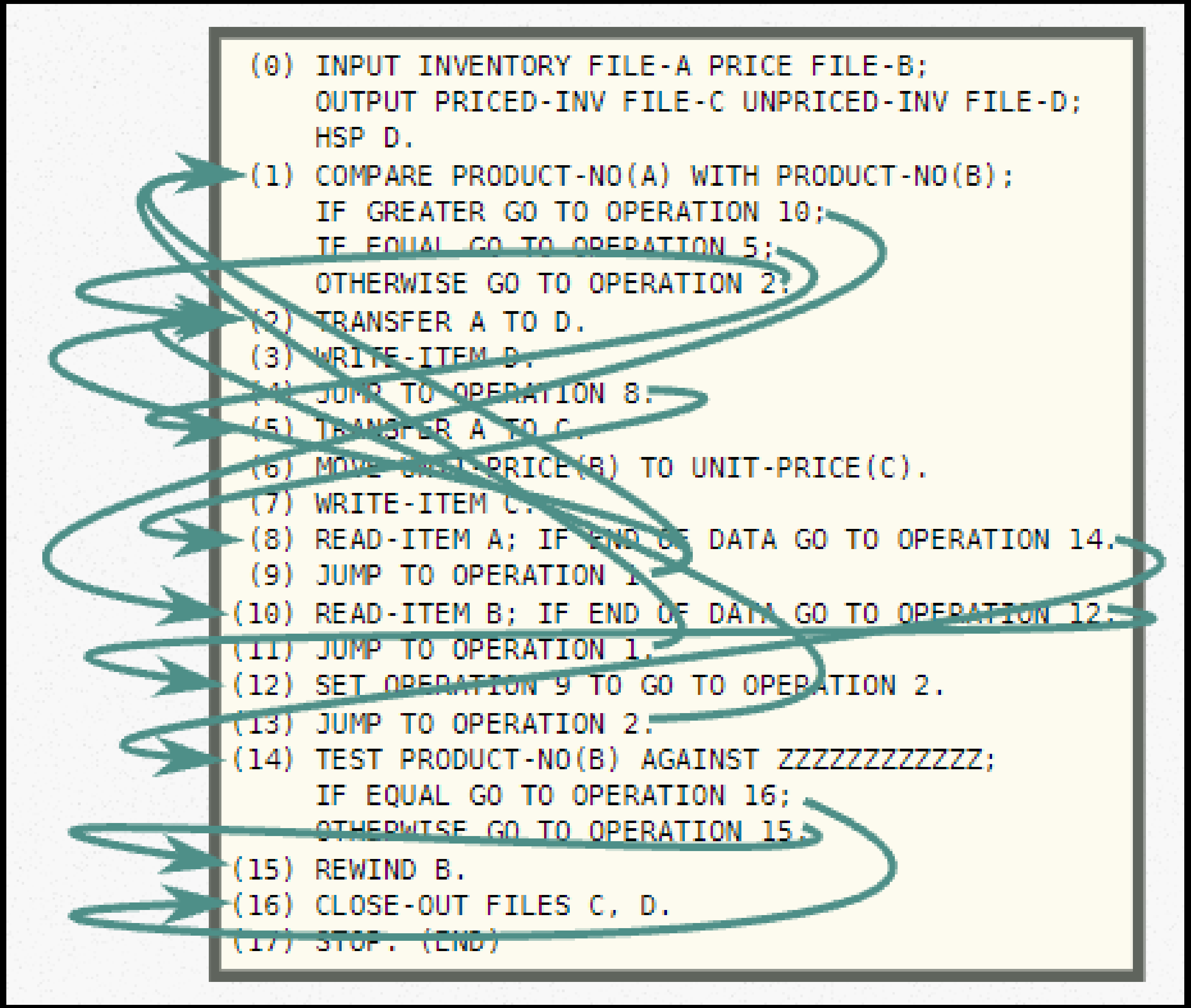
1. Vision: “An asynchronous analogue of the STL”
2. Some simple examples, intro to senders
3. The lifecycle of an async operation with sender/receiver
4. Under the hood of a composite concurrent operation
5. Implementing a simple algorithm: `then()`
6. Senders and coroutines

WHAT’S COMING IN PART 2

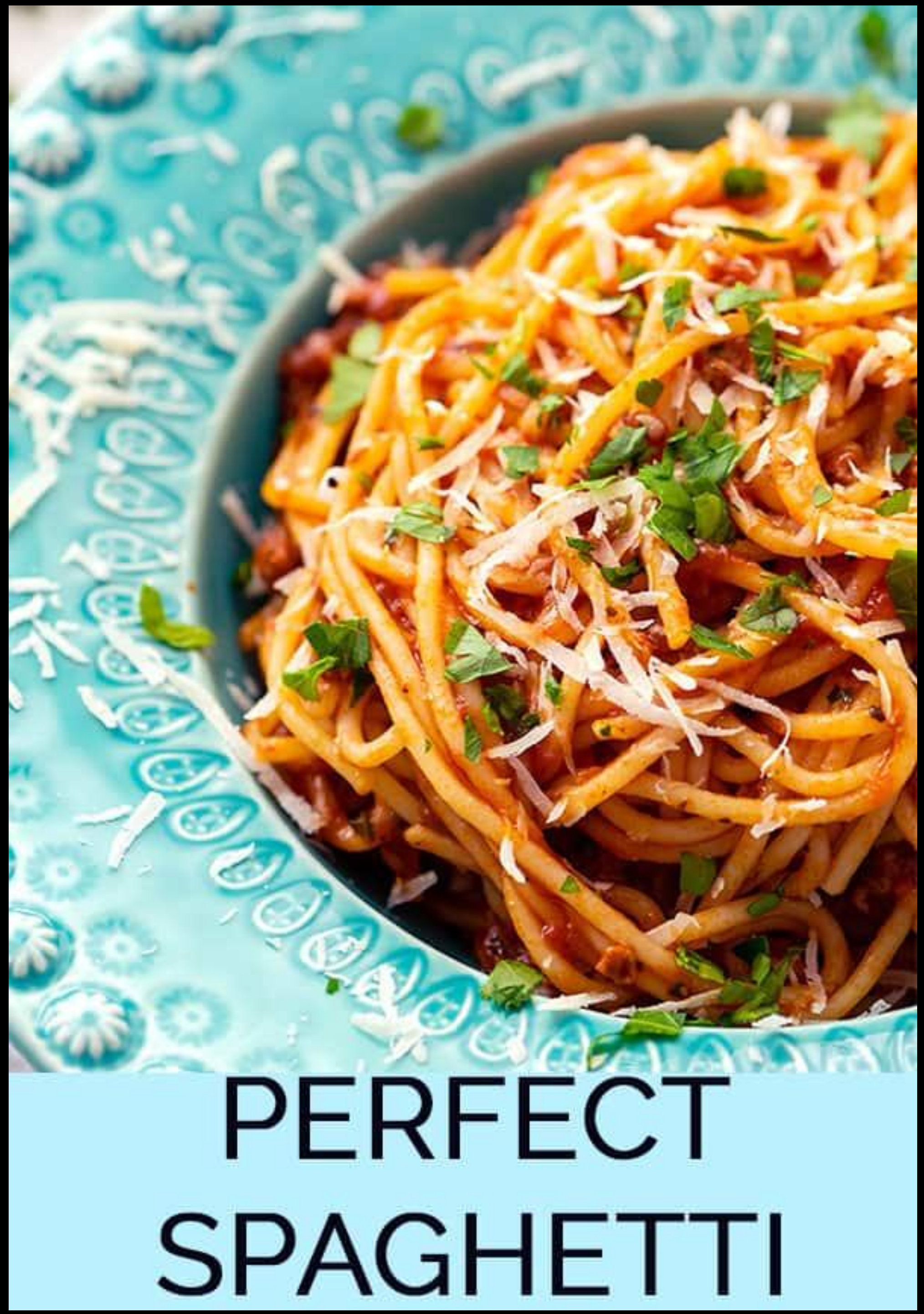
1. Structured concurrency
2. Cancellation support in the sender/receiver abstraction
3. An extended example

Structured concurrency

IN THE BEGINNING … WAS GOTO



```
(0)  INPUT INVENTORY FILE-A PRICE FILE-B;
     OUTPUT PRICED-INV FILE-C UNPRICED-INV FILE-D;
     HSP D.
(1)  COMPARE PRODUCT-NO(A) WITH PRODUCT-NO(B);
     IF GREATER GO TO OPERATION 10;
     IF EQUAL GO TO OPERATION 5;
     OTHERWISE GO TO OPERATION 2;
(2)  TRANSFER A TO D.
(3)  WRITE-ITEM D.
(4)  JUMP TO OPERATION 8.
(5)  TRANSFER A TO C.
(6)  MOVE UNIT-PRICE(B) TO UNIT-PRICE(C).
(7)  WRITE-ITEM C.
(8)  READ-ITEM A; IF END OF DATA GO TO OPERATION 14.
(9)  JUMP TO OPERATION 1.
(10) READ-ITEM B; IF END OF DATA GO TO OPERATION 12.
(11) JUMP TO OPERATION 1.
(12) SET OPERATION 9 TO GO TO OPERATION 2.
(13) JUMP TO OPERATION 2.
(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;
     IF EQUAL GO TO OPERATION 16;
     OTHERWISE GO TO OPERATION 15.
(15) REWIND B.
(16) CLOSE-OUT FILES C, D.
(17) STOP. (END)
```

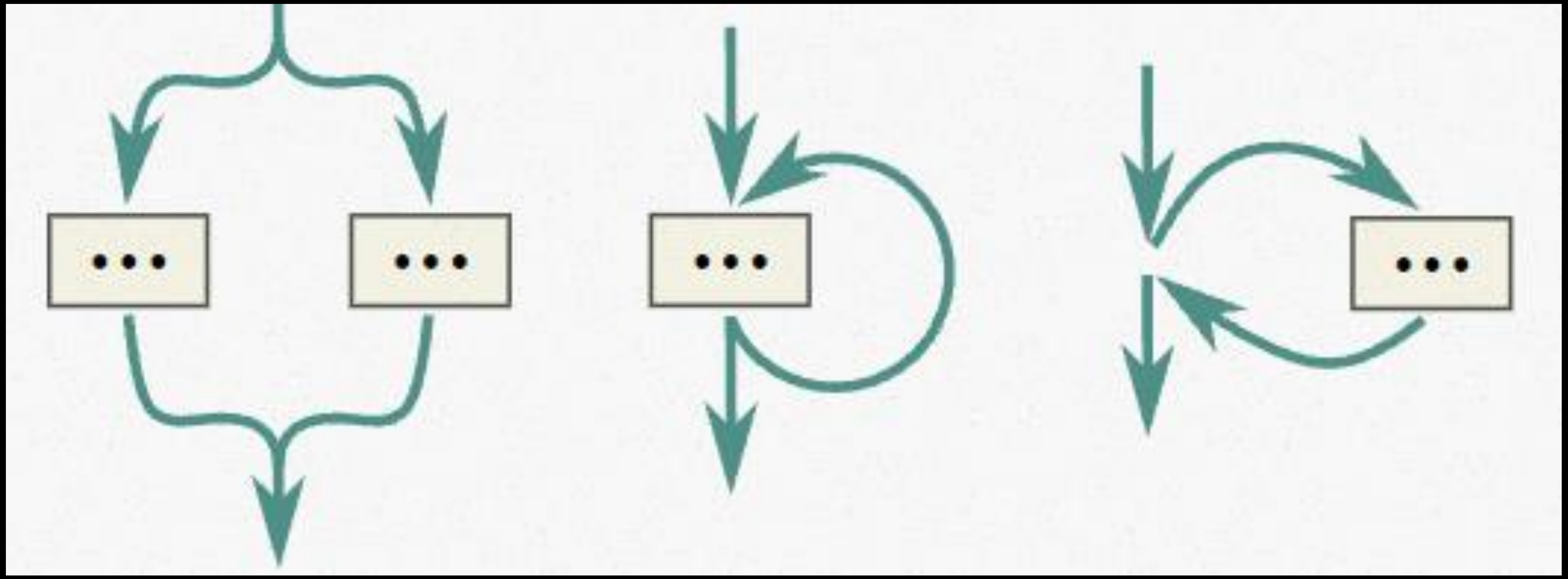


From: "Go Statement considered harmful," by Nathaniel Smith

IN THE BEGINNING … WAS ~~GOTO~~ STRUCTURED PROGRAMMING

Structured control flow constructs have a single entry and a single exit, permitting them to be treated like a black box.

goto is an *unstructured* control flow construct



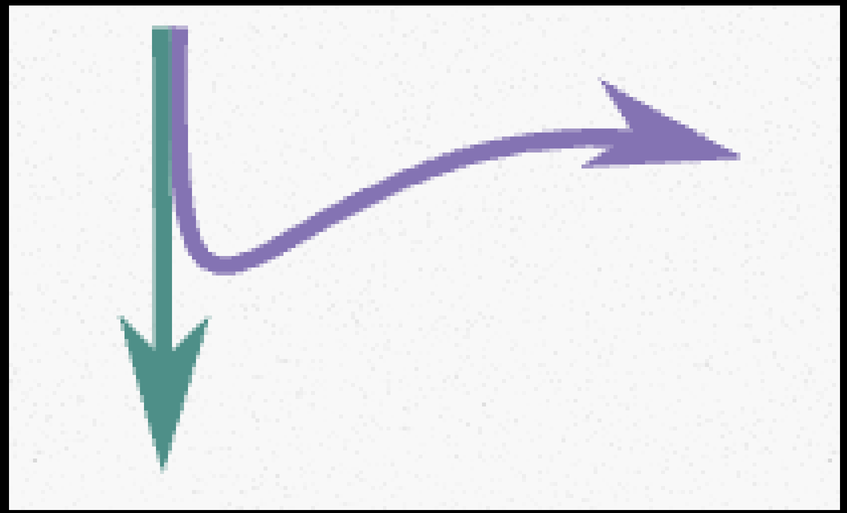
conditional loop function call

goto

FIRE-AND-FORGET TASK MODELS ARE *UNSTRUCTURED*

Fire-and-forget work is like goto.

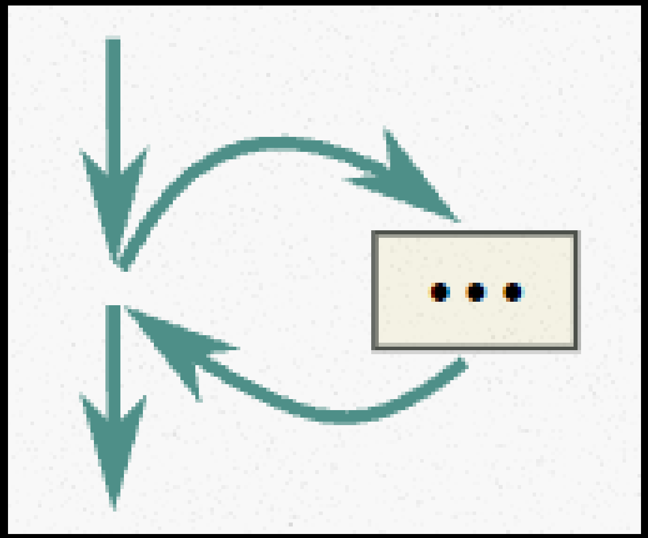
```
void compute_helper_async();

void compute_async(Executor ex)
{
  /* ... */

  // Spawn helper in specified context
  // w/ an executor like in ASIO and NetTS
  ex.execute(&compute_helper_async);

  /* ... */
}
```



execute

COROUTINES ARE *STRUCTURED* CONCURRENCY

```
// This is a coroutine:
task<int> compute_helper_async();

task<void> compute_async()
{
  /* ... */

  int i = co_await compute_helper_async();

  /* ... */
}
```



coroutine

Awaiting a coroutine has a single entry and a single exit.

COROUTINES ARE *STRUCTURED* CONCURRENCY

```cpp
// This is a coroutine:
task<int> compute_helper_async();

task<void> compute_async()
{
  /* ... */

  int i = co_await compute_helper_async();

  /* ... */
}
```

Activation of callee coroutine….

…is wholly nested within activation of the caller coroutine.

COROUTINES ARE *STRUCTURED* CONCURRENCY

```cpp
// This is a coroutine:
task<int> compute_helper_async();

task<void> compute_async()
{
  /* ... */

  int i = co_await compute_helper_async();

  /* ... */
}
```

Activations nest.

Scopes nest.

Lifetimes of locals nest.

RAII works.

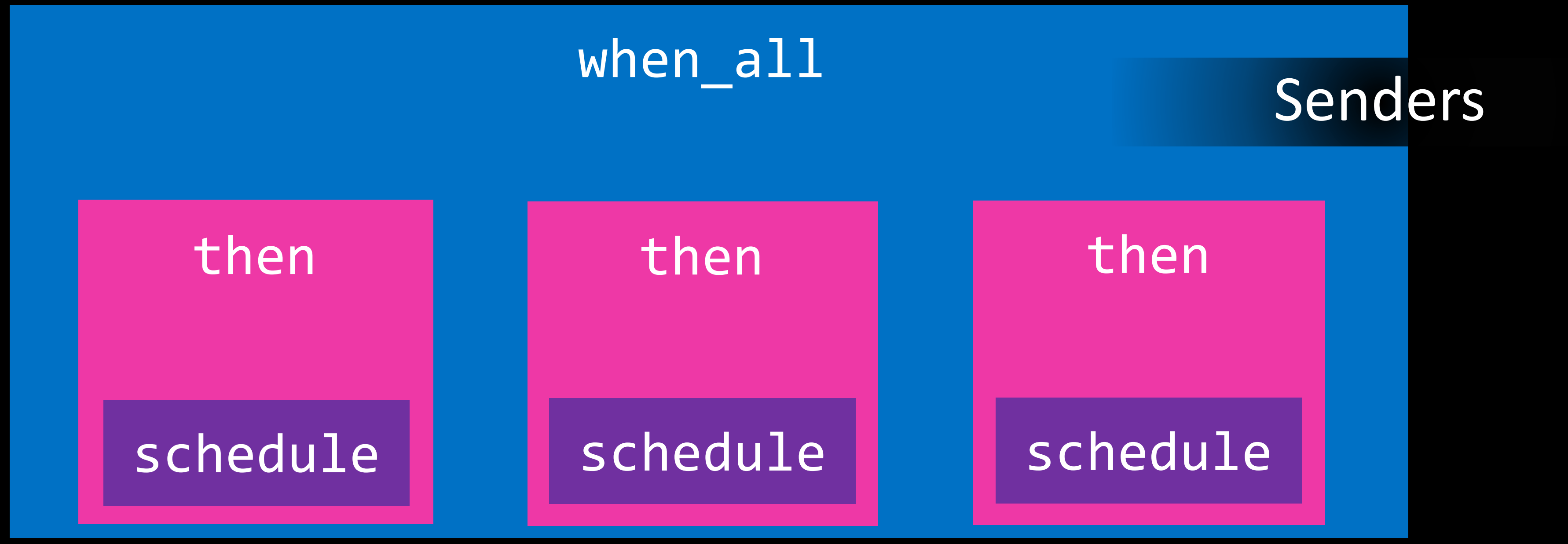
COROUTINES ARE *STRUCTURED* CONCURRENCY

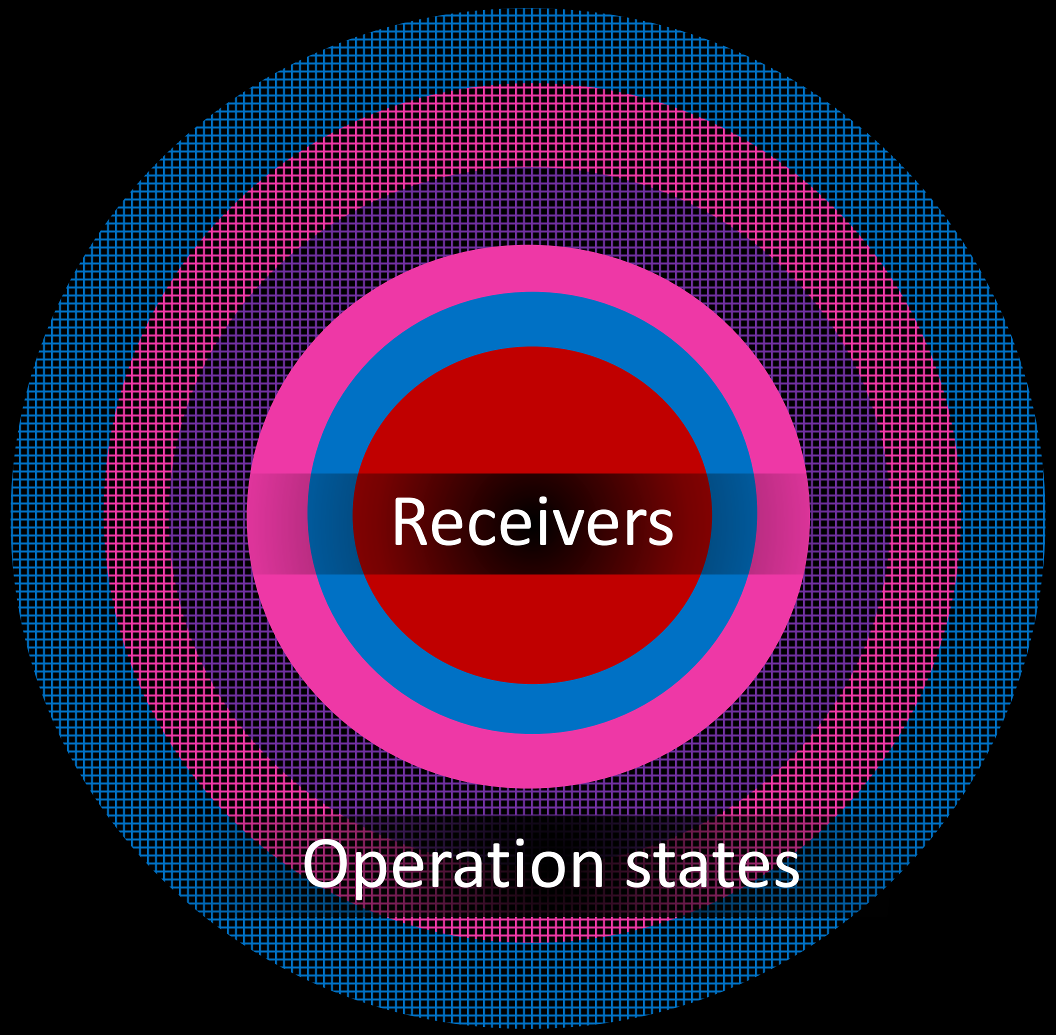
```
// This is a coroutine:
task<int> compute_helper_async(int& data);

task<void> compute_async()
{
  int data = 0;

  int i = co_await compute_helper_async(data);

  /* ... */
}
```

Because of the nested scopes, it's safe to pass locals by reference to callees...

... no dynamic allocation or reference counting needed.

SENDER/RECEIVER IS ALSO STRUCTURED CONCURRENCY





Activations nest.
Scopes nest.
Lifetimes of locals nest.
RAII works.

NO DETACHED COMPUTATION ALLOWED

schedule

then

schedule

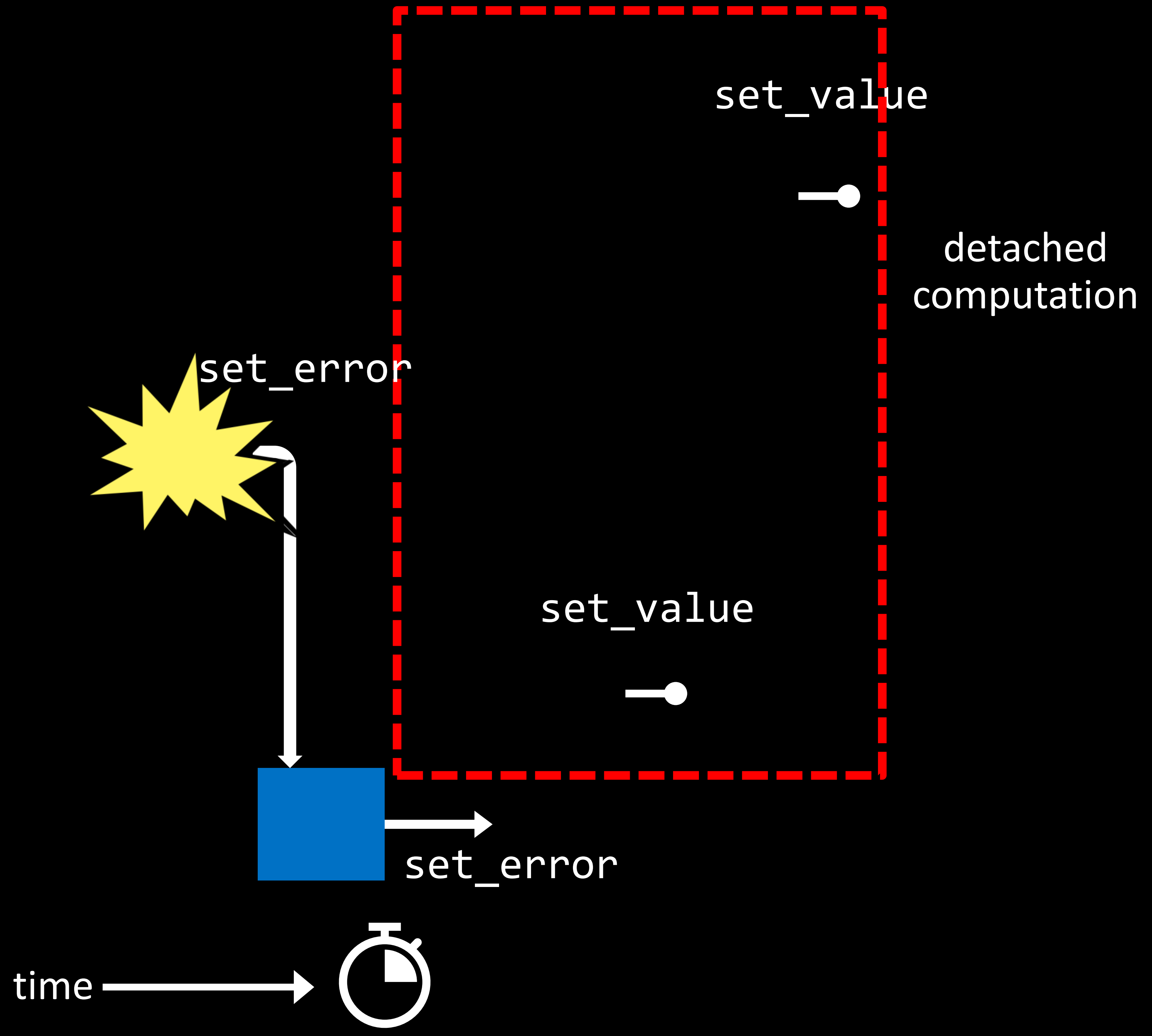
then

schedule

then

when_all

A thread in the thread pool picks up this work and executes it.

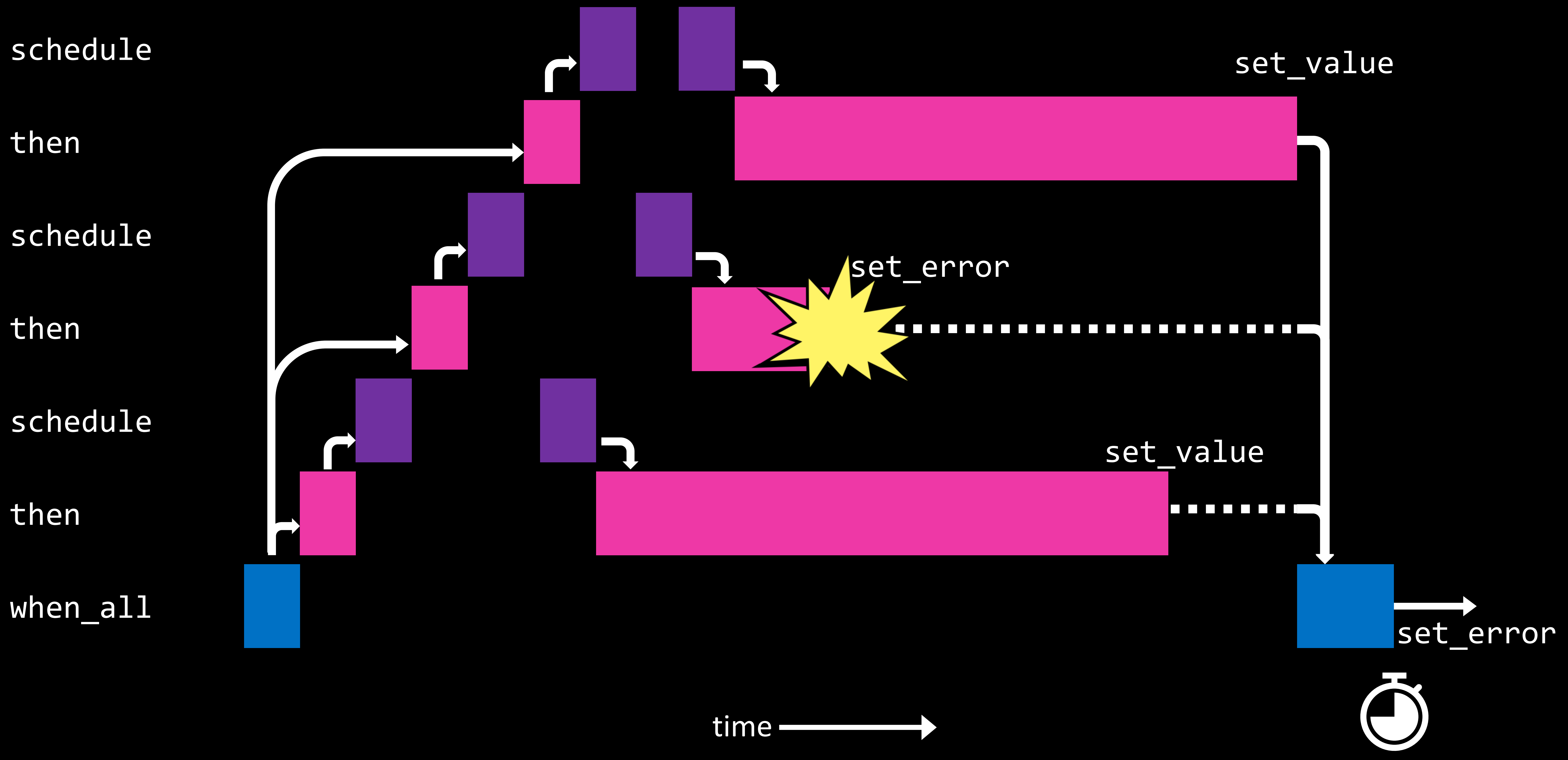


WHY IS DETACHED COMPUTATION BAD?

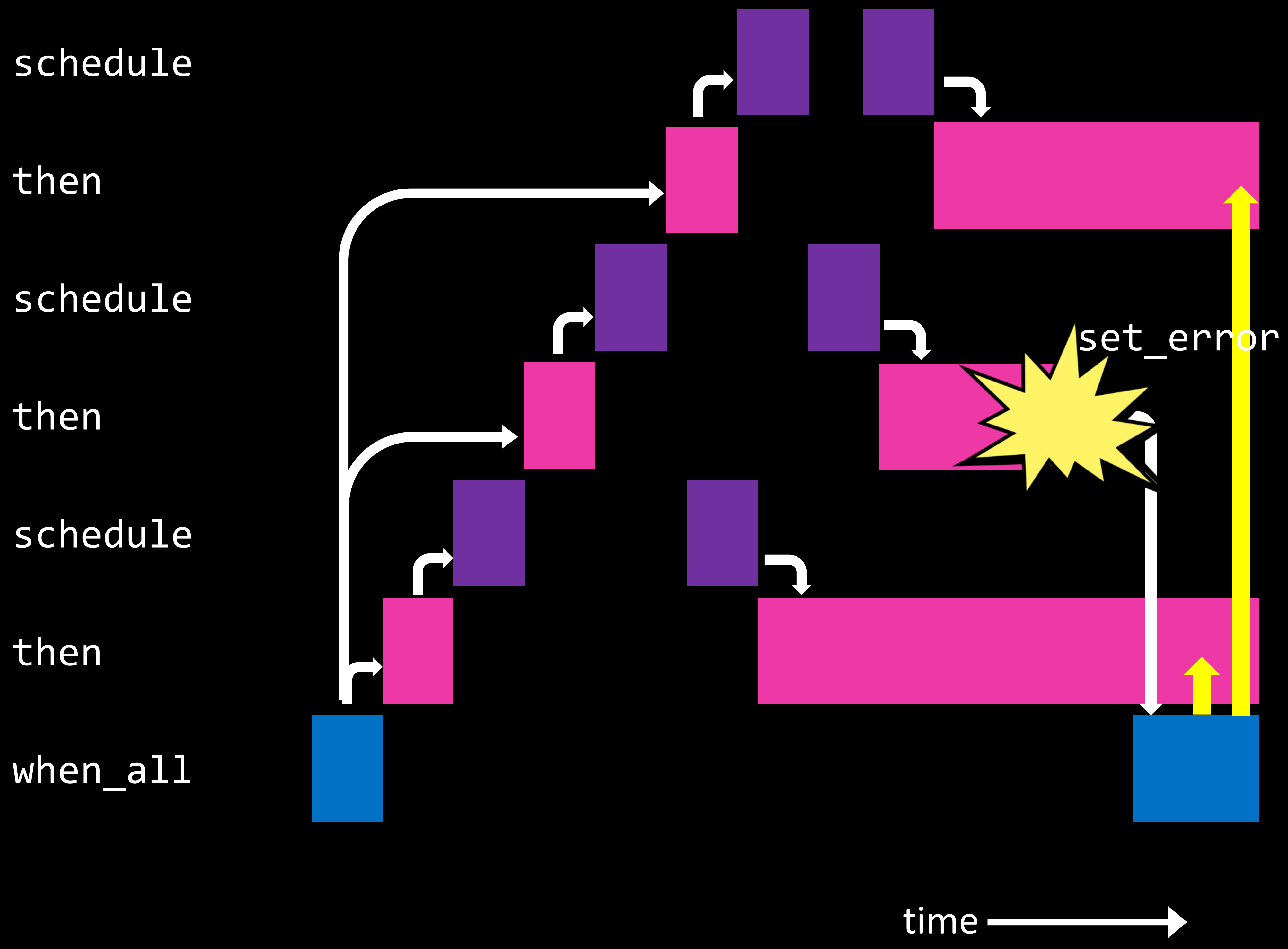
```
int compute_helper(int& data);

void compute()
{
  int data = 0;
  int result = compute_helper(data);

  /* ... */
}
```

What would happen if `compute()` could return after calling `compute_helper()` but before `compute_helper()` returned?

CONCURRENCY MUST BE JOINED!



CONCURRENCY MUST BE JOINED!



In structured concurrency, deep support for cooperative cancellation is essential for good performance.

CANCELLATION IN SENDER/RECEIVER

CANCELLATION SUPPORT IN C++20

Built on top of C++20's `std::stop_token` support

Calling code…

- … declares a `std::stop_source` and calls `get_token()` on it to get a stop token,
- … passes the token to an async operation when launching it, and
- … calls `request_stop()` on the stop source when it wants the async operation to stop early.

Async code can either…

- … poll the token periodically to see if stop has been requested, or…
- … register a callback to be executed when the caller requests stop.

cppreference.com Create account Sea

Page Discussion View Edit Hi

C++ / Standard Library headers

Standard library header <stop_token>

This header is part of the thread support library.

Classes

`stop_token` (C++20)	an interface for querying if a `std::jthread` cancellation request has been made (class)
`stop_source` (C++20)	class representing a request to stop one or more `std::jthread`s (class)
`stop_callback` (C++20)	an interface for registering callbacks on `std::jthread` cancellation (class template)
`nostopstate_t` (C++20)	placeholder type for use in stop_source constructor (class)

Constants

`nostopstate` (C++20)	a std::nostopstate_t instance for use in stop_source constructor (constant)

Synopsis

```
namespace std {
  // class stop_token
  class stop_token;

  // class stop_source
  class stop_source;

  // no-shared-stop-state indicator
  struct nostopstate_t {
    explicit nostopstate_t() = default;
  };
  inline constexpr nostopstate_t nostopstate{};

  // class stop_callback
  template<class Callback>
  class stop_callback;
}
```

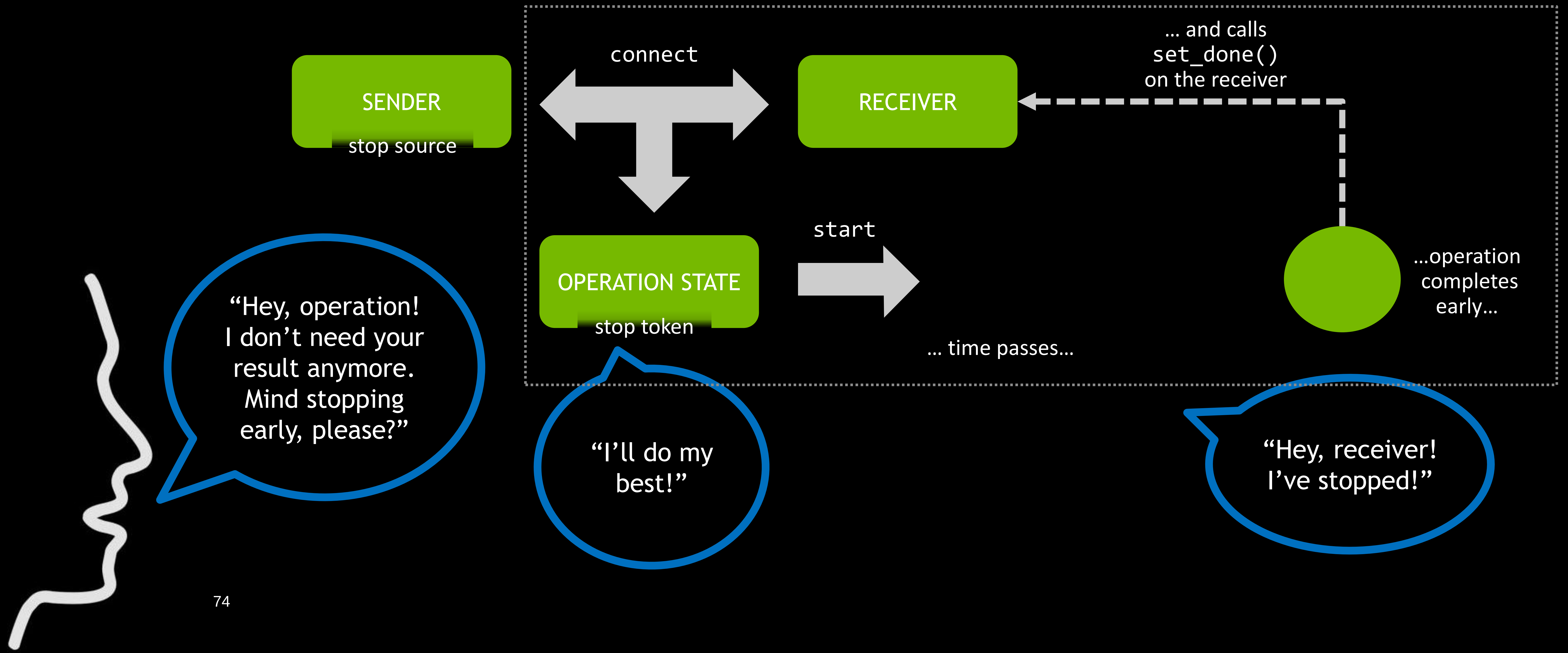
https://en.cppreference.com/w/cpp/header/stop_token

CANCELLATION IN SENDER/RECEIVER

Some senders have stop *sources*.

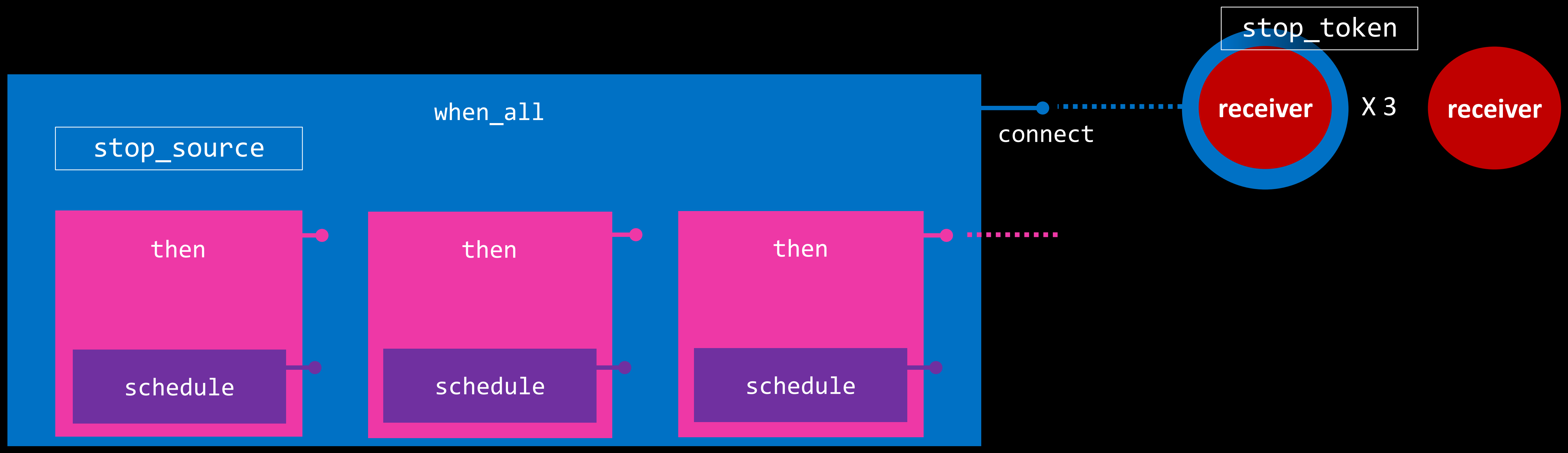
They pass stop *tokens* to the operation state.

Cancellation details are internal to algorithms.



CANCELLATION IS A THREE-WAY HANDSHAKE

SENDERS PASS STOP TOKENS TO CHILDREN VIA RECEIVERS



SENDER/RECEIVER AND CANCELLATION

- Orchestrating cancellation is the job of the *algorithms*, not the user.
- Only algorithms that introduce concurrency need to handle cancellation.
- Cancellation is *asynchronous* and *cooperative*.
- Cancellation is *best-effort*

  No guarantee an operation will stop promptly or at all
- Dedicated algorithms capture common cancellation patterns
  - *e.g.*: `unifex::stop_when()`

EXAMPLE: UNIFEX::STOP_WHEN

```
// unifex::stop_when(
//     sender auto task,
//     sender auto condition) -> sender auto;
//
// stop_when takes two senders:
//   1. task:      the async operation to perform
//   2. condition: the async stopping condition
//
// When one operation finishes, it cancels the other. It completes
// with the result of `task` when both operations complete.

// process input in a loop until the user interrupts:
sender auto work_loop =
    unifex::stop_when(
        unifex::repeat_effect( processInput() ),
        userInterrupt()
    );
```

AN EXTENDED EXAMPLE:
SENDERS, COROUTINES, AND RANGES, OH MY!

But first, a sad ☹ story about a boy and the greatest keyboard ever made….


IBM

The mission:

Write a program that monitors the entire system for keyboard events and plays **Model M** clicky sounds.

MODEL M SIMULATOR: STRATEGY

1. Model a key click as a sender
2. Model keyboard input as a range of senders
3. Model interrupt (e.g., Ctrl-C) as a sender
4. Asynchronously transform range of senders into clicky noises until interrupt sender completes.
5. ???
6. Profit!

Step 1:

Model key click as a sender

MODEL KEYCLICK AS SENDER: STRATEGY

1. Assume system API for registering keyboard callback
2. Write a keyboard *sender* and *op state* such that:
   - ...the sender's `connect()` returns the op state wrapping the user's receiver.
   - ...the op state's `start()` places a callback that completes the receiver in a global.
3. Register a keyboard callback that reads the completion info out of the global and completes it if it's not null.

KEYCLICK SENDER

```cpp
// Type-erased receiver waiting for a keyclick:
struct pending_completion {
  virtual void complete(char) = 0;
  virtual ~pending_completion() {}
};

// Global registration of next completion:
std::atomic<pending_completion*> pending_completion_{nullptr};

// Keyboard input callback to register with the system:
static void on_keyclick(char ch) {
  auto* current = pending_completion_.exchange(nullptr);
  if (current != nullptr) {
    current->complete(ch);
  }
}
```

KEYCLICK SENDER

```
// Sender that completes with the next keyclick
// read from stdin:
struct keyclick_sender : _sender_of<char>
{
  auto connect(unifex::receiver_of<char> auto rec) {
    return keyclick_operation{std::move(rec)};
  }
};

keyclick_sender read_keyclick() {
  return {};
}
```

KEYCLICK SENDER

```
// Operation state for a single key click:
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;

  explicit keyclick_operation(Rec rec) : rec_(std::move(rec)) {}

  void complete(char ch) override final {
    if (ch == CTRL_C)
      unifex::set_done(std::move(rec_));
    else
      unifex::set_value(std::move(rec_), ch);
  }

  void start() noexcept {
    // Enqueue the operation
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

KEYCLICK SENDER

```
// Operation state for a single key click:
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;

  explicit keyclick_operation(Rec rec) : rec_(std::move(rec)) {}

  void complete(char ch) override final {
    if (ch == CTRL_C)
      unifex::set_done(std::move(rec_));
    else
      unifex::set_value(std::move(rec_), ch);
  }

  void start() noexcept {
    // Enqueue the operation
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

…OR USE A HELPER TO CREATE THE SENDER

```
auto read_keyclick() {
  return unifex::create_simple<char>(
    [](unifex::receiver_of<char> auto& r) {
      return keyclick_state{r};
    });
}
```

Returns a sender whose operation state:

calls the lambda in its `start()` function

stores the returned object in the operation state

```
// State for a single key click:
template <unifex::receiver_of<char> Rec>
struct keyclick_state : pending_completion {
  Rec& rec_;

  explicit keyclick_state(Rec& rec) : rec_(rec) {
    // Enqueue the operation
    auto* previous =
      pending_completion_.exchange(this);
    assert(previous == nullptr);
  }

  void complete(char ch) override final {
    if (ch == CTRL_C)
      unifex::set_done(std::move(rec_));
    else
      unifex::set_value(std::move(rec_), ch);
  }

  keyclick_state(keyclick_state &&) = delete;
};
```

“SYSTEM” API FOR REGISTERING KEYBOARD CALLBACK

```
// Fake system API for registering keyboard input callback:
void register_keyboard_callback(void (*callback)(char)) {
  static std::jthread th([=](std::stop_token token) {
      while (!token.stop_requested()) {
        int ch = _getch(); // Totally not portable! ☹
        callback((char)ch);
        if (ch == CTRL_C)
          break;
      }
    });
}
```

PUTTING THE PIECES TOGETHER

```
int main() {
  register_keyboard_callback(on_keyclick);

  // Build an operation that reads a keyclick and
  // executes a continuation:
  auto read_next_char =
      read_keyclick()
    | unifex::then([](char ch) {
        printf("In then with char: %c\n", ch);
      });

  (void) unifex::sync_wait(read_next_char);
}
```

(demo 1)

PUTTING THE PIECES TOGETHER

```
int main() {
  register_keyboard_callback(on_keyclick);

  // Build ar
  // executes
  auto read_r
      read_ke
    | unifex:
        print
      });

  (void) unifex::sync_wait(read_next_char);
}
```

The mission:

Write a program that monitors the entire system for keyboard events and plays **Model M** clicky sounds.

~~HACKING~~ HOOKING WINDOWS

SetWindowsHookExA function (winuser.h)

10/13/2021 • 7 minutes to read

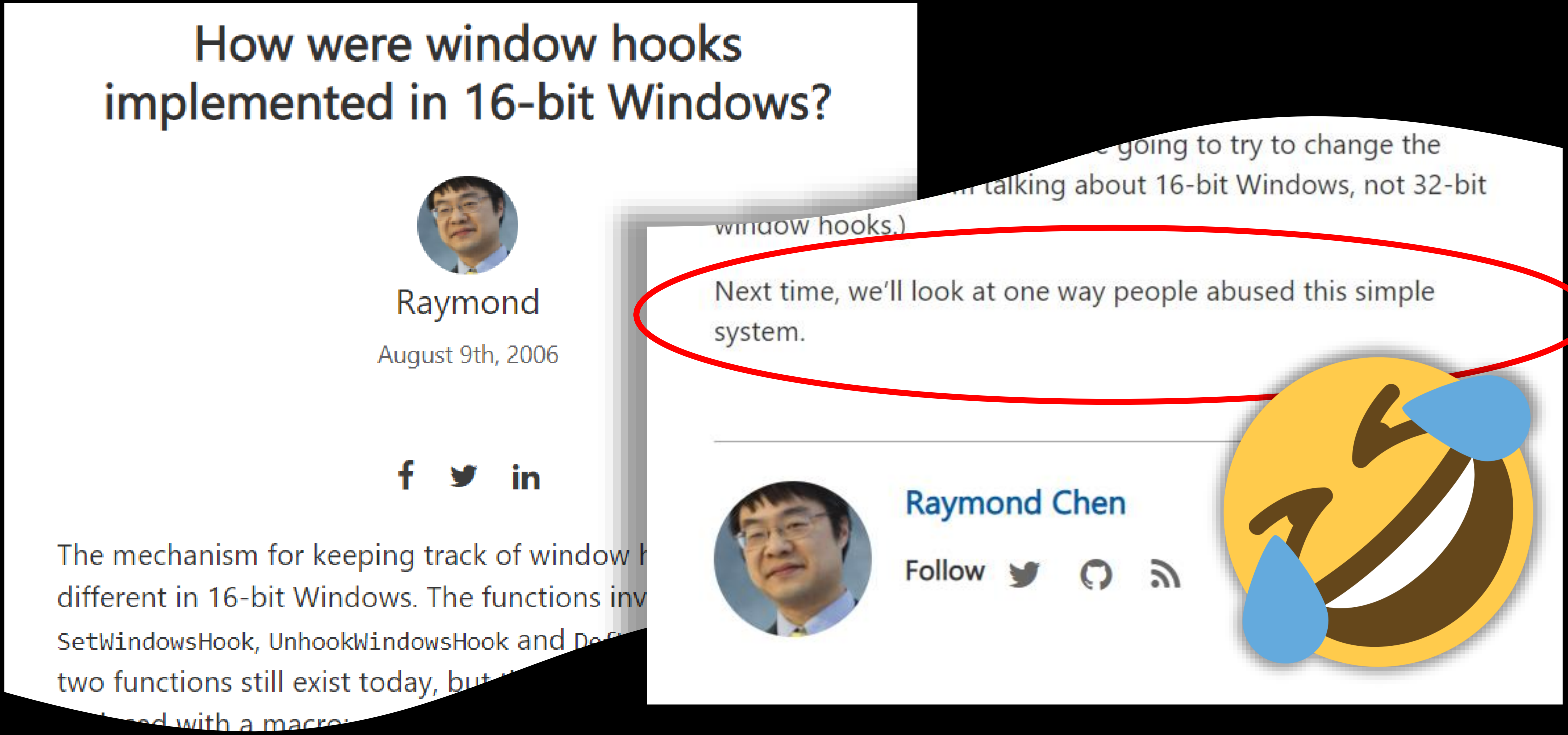
Installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.

Syntax

C++ Copy

```
HHOOK SetWindowsHookExA(
  [in] int       idHook,
  [in] HOOKPROC  lpfn,
  [in] HINSTANCE hmod,
  [in] DWORD     dwThreadId
);
```

~~HACKING~~ HOOKING WINDOWS


How were window hooks implemented in 16-bit Windows?

Raymond

August 9th, 2006

The mechanism for keeping track of window h
different in 16-bit Windows. The functions inv
SetWindowsHook, UnhookWindowsHook and De
two functions still exist today, but

going to try to change the
talking about 16-bit Windows, not 32-bit
window hooks.)

Next time, we'll look at one way people abused this simple system.

Raymond Chen

Follow

Raymond Chen, "Old New Thing", https://devblogs.microsoft.com/oldnewthing/20060809-18/?p=30183

Step 2:

Model keyboard input as a range of senders

A RANGE OF KEYCLICK SENDERS

```cpp
auto keyclicks() {
  // Return an infinite range of async
  // keyclicks
  return std::views::iota(0u)
    | std::views::transform([](auto) {
        return read_keyclick();
      });
}
```

```cpp
unifex::task<void> echo_keyclicks() {
  for (auto keyclick : keyclicks()) {
    char ch = co_await std::move(keyclick);
    printf("Read a character! %c\n", ch);
  }
}

int main() {
  register_keyboard_callback(on_keyclick);

  (void)unifex::sync_wait(echo_keyclicks());
}
```

A RANGE OF KEYCLICK SENDERS

```
unifex::task<void> echo_keyclicks() {
  for (auto keyclick : keyclicks()) {
    char ch = co_await std::move(keyclick);
    printf("Read a character! %c\n", ch);
  }
}

int main() {
  register_keyboard_callback(on_keyclick);

  (void)unifex::sync_wait(echo_keyclicks());
}
```

If the keyclick is `Ctrl-C`, the sender completes by calling `set_done()`.

In a coroutine, if an awaited sender completes with `set_done()`, the result is an uncatchable “exception”.

`sync_wait()` handles this “exception” and returns an empty optional.

A RANGE OF KEYCLICK SENDERS

```
unifex::task<void> echo_keyclicks() {
  for (auto keyclick : keyclicks()) {
    std::optional<char> ch =
      co_await unifex::done_as_optional(std::move(keyclick));

    if (ch) {
      printf("Read a character! %c\n", *ch);
    } else {
      printf("Interrupt!\n");
      break;
    }
  }
}
```

Use done_as_optional() to “catch” the cancellation and map it into an empty optional.

(demo 2)

A RANGE OF KEYCLICK SENDERS

```
unifex::task<void> echo_keyclicks() {
  for (auto keyclick : keyclicks() | std::views::transform(unifex::done_as_optional)) {
    std::optional<char> ch = co_await std::move(keyclick);

    if (ch) {
      printf("Read a character! %c\n", ch);
    } else {
      printf("Interrupt!\n");
      break;
    }
  }
}
```

Sender transformation could also be applied with a range transform.

A RANGE OF KEYCLICK SENDERS

```
keyclicks() | std::views::transform(unifex::done_as_optional)
```

This is a synchronous range adaptor, which applies to the senders themselves.

It’s not hard to imagine one might want to apply a range adaptor to the *results* of the senders.

Asynchronous ranges beg asynchronous range adaptors.

Step 3:
Model Ctrl-C as a sender

CTRL-C SENDER (WINDOWS-SPECIFIC)

`consoleHandler()` checks if there is a pending completion. If so, it completes it.

The constructor registers the control handler, and the destructor unregisters it.

```
struct ctrl_c_handler {
  struct pending {
    virtual void complete() = 0;
    virtual ~pending() {}
  };
  static inline std::atomic<pending*> pending_{nullptr};

  static BOOL WINAPI consoleHandler(DWORD signal) {
    if (signal == CTRL_C_EVENT) {
      if (auto* pending = pending_.exchange(nullptr))
        pending->complete();
    }
    return TRUE;
  }

  ctrl_c_handler() {
    BOOL result = ::SetConsoleCtrlHandler(&consoleHandler,TRUE);
    assert(result);
  }
  ~ctrl_c_handler() {
```

CTRL-C SENDER (WINDOWS-SPECIFIC)

```
};

auto ctrl_c_handler::event() const {
  return unifex::create_simple<>(
    []<unifex::receiver_of R>(R& rec) {
      struct state : pending {
        R& rec_;
        state(R& rec) : rec_(rec) {
          auto* previous = pending_.exchange(this);
          assert(previous == nullptr);
        }
        void complete() override final { unifex::set_value(std::move(rec_)); }
        state(state&&) = delete;
      };
      return state{rec};
    }
  );
}
```

`ctrl_c_handler::event()` returns a sender…

CTRL-C SENDER (WINDOWS-SPECIFIC)

```
};

auto ctrl_c_handler::event() const {
  return unifex::create_simple<>(
    []<unifex::receiver_of R>(R& rec) {
      struct state : pending {
        R& rec_;
        state(R& rec) : rec_(rec) {
          auto* previous = pending_.exchange(this);
          assert(previous == nullptr);
        }
        void complete() override final { unifex::set_value(std::move(rec_)); }
        state(state&&) = delete;
      };
      return state{rec};
    }
  );
}
```

`ctrl_c_handler::event()` returns a sender…

whose operation state calls this lambda in its `start()` function …

CTRL-C SENDER (WINDOWS-SPECIFIC)

```
};

auto ctrl_c_handler::event() const {
  return unifex::create_simple<>(
    []<unifex::receiver_of R>(R& rec) {
      struct state : pending {
        R& rec_;
        state(R& rec) : rec_(rec) {
          auto* previous = pending_.exchange(this);
          assert(previous == nullptr);
        }
        void complete() override final { unifex::set_value(std::move(rec_)); }
        state(state&&) = delete;
      };
      return state{rec};
    }
  );
}
```

`ctrl_c_handler::event()` returns a sender…

whose operation state calls this lambda in its `start()` function …

that returns this object to be stored in the operation state.

CTRL-C SENDER (WINDOWS-SPECIFIC)

```
};

auto ctrl_c_handler::event() const {
  return unifex::create_simple<>(
    []<unifex::receiver_of R>(R& rec) {
      struct state : pending {
        R& rec_;
        state(R& rec) : rec_(rec) {
          auto* previous = pending_.exchange(this);
          assert(previous == nullptr);
        }
        void complete() override final { unifex::set_value(std::move(rec_)); }
        state(state&&) = delete;
      };
      return state{rec};
    }
  );
}
```

Step 4:
Asynchronously transform range of senders into clicky noises until interrupt sender completes.

… UNTIL INTERRUPT SENDER COMPLETES

1. Use **`unifex::stop_when()`** sender adaptor to send stop request when Ctrl-C sender completes.
2. Remove no-longer-necessary special handling for Ctrl-C from the **`keyclick_operation`**.

… UNTIL INTERRUPT SENDER COMPLETES

```
int main() {
  register_keyboard_callback(on_keyclick);
  ctrl_c_handler ctrl_c;

  (void) unifex::sync_wait(
      echo_keyclicks()
    | unifex::stop_when(ctrl_c.event()));
}
```

`stop_when()` sends a stop request to its child operation (`echo_keyclicks`) when its trigger (`ctrl_c.event`) completes…

… UNTIL INTERRUPT SENDER COMPLETES

```
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;

  explicit keyclick_operation(Rec rec)
    : rec_(std::move(rec)) {}

  void complete(char ch) override final {
    if (CTRL_C == ch)
      unifex::set_done(std::move(rec_));
    else
      unifex::set_value(std::move(rec_), ch);
  }

  void start() noexcept {
    // Enqueue the operation
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

… UNTIL INTERRUPT SENDER COMPLETES

```
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;

  explicit keyclick_operation(Rec rec)
    : rec_(std::move(rec)) {}

  void complete(char ch) override final {



    unifex::set_value(std::move(rec_), ch);
  }

  void start() noexcept {
    // Enqueue the operation
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

Since we are externalizing the stop condition, we can remove the special handling for Ctrl-C from keyclick_operation.

(demo 3)

Q: Why doesn't this work?

A: Although `stop_when()` sends a stop request, the keyclick operation isn't listening for one!

It should register a stop callback.

MAKE KEYCLICK OPERATIONS INTERRUPTABLE

```
struct pending_completion {
  virtual void complete(char) = 0;
  virtual void cancel() = 0;
  virtual ~pending_completion() {}
};

// Function to execute (synchronously) when the
// stop_when() algorithm requests stop of its stop
// source.
struct cancel_keyclick {
  void operator()() const noexcept {
    auto* current = pending_completion_.exchange(nullptr);
    if (current != nullptr) {
      current->cancel();
    }
  }
};
```

Add a way to cancel a pending completion.

Write a function that cancels the pending completion.

… UNTIL INTERRUPT SENDER COMPLETES

```
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;
  std::optional<stop_callback_for_t<Rec, cancel_keyclick>> on_stop_{};

  explicit keyclick_operation(Rec rec) : rec_(std::move(rec)) {}

  void complete(char ch) override final {
    unifex::set_value(std::move(rec_), ch);
  }

  void cancel() override final {
    unifex::set_done(std::move(rec_));
  }

  void start() noexcept {
    on_stop_.emplace(unifex::get_stop_token(rec_), cancel_keyclick{});
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

Register a stop callback with the receiver's associated stop token that will cancel the outstanding completion.

This stop token receives stop requests from the stop source in the stop_when() algorithm.

… UNTIL INTERRUPT SENDER COMPLETES

```
template <unifex::receiver_of<char> Rec>
struct keyclick_operation : pending_completion {
  Rec rec_;
  std::optional<stop_callback_for_t<Rec, cancel_keyclick>> on_stop_{};

  explicit keyclick_operation(Rec rec) : rec_(std::move(rec)) {}

  void complete(char ch) override final {
    unifex::set_value(std::move(rec_), ch);
  }

  void cancel() override final {
    unifex::set_done(std::move(rec_));
  }

  void start() noexcept {
    on_stop_.emplace(unifex::get_stop_token(rec_), cancel_keyclick{});
    auto* previous = pending_completion_.exchange(this);
    assert(previous == nullptr);
  }
};
```

Here be dragons.

… AND NOW THIS WORKS

```
int main() {
  register_keyboard_callback(on_keyclick);
  ctrl_c_handler ctrl_c;

  (void) unifex::sync_wait(
      echo_keyclicks()
    | unifex::stop_when(ctrl_c.event()));
}
```

`stop_when()` sends a stop request to its child operation (`echo_keyclicks()`) when its trigger (`ctrl_c.event()`) completes…

(demo 4)

All that remains is a boat-load of nasty platform-specific hackery
(hook Windows events, play clicky sounds).

You can find all the demo code including the full-fat example from Kirk Shoop at:
https://github.com/ericniebler/executors_demo_code_cppcon_2021.git

DEMO TIME

(with huge shoutout to Kirk Shoop)

Where to now?

WHAT'S TO COME

P2300 "std::execution", currently on track for C++23, brings:

- the concepts,
- the customization points,
- a handful of fundamental async algorithms,
- coroutine integration, and
- integration with the C++17 parallel algorithms.

Future additions will include:

- more standard async algorithms (see libunifex)
- a `timed_scheduler` concept and time-based async algorithms; e.g., `timeout()`.
- portable access to a "system" scheduler, backed by e.g., Windows Thread Pool or GCD
- a manual event loop scheduler
- a nursery in which async work can be spawned (fire-and-forget) and stopped and/or joined.

Expect coroutine types that are deeply integrated with sender/receiver and ranges:

- `std::task`
- `std::generator`
- `std::async_generator`

WHAT’S TO COME, CONTINUED

Fully async flavors of the C++17 parallel algorithms are currently in the planning stages.

Full support for async ranges (aka, reactive streams), together with a suite of reactive algorithms and adaptors is in the early prototyping phase.

IO schedulers?

Simple async socked-based networking?

ADDITIONAL RESOURCES

P2300R2: “std::execution”:

https://wg21.link/P2300R2

Libunifex:

https://github.com/facebookexperimental/libunifex

Demo code:

https://github.com/ericniebler/executors_demo_code_cppcon_2021.git