Document No.	P2006R0
Date	2020-01-13
Reply To	Lewis Baker <lbaker@fb.com> Eric Niebler <eniebler@fb.com> Kirk Shoop <kirkshoop@fb.com> Lee Howes <lwh@fb.com>
Audience	SG1, LEWG
Targeting	C++23

Eliminating heap-allocations in sender/receiver with connect()/start() as basis operations

Abstract

The "Unified executors" paper, P0443R11, was recently updated to incorporate the sender/receiver concepts as the basis for representing composable asynchronous operations in the standard library.

The basis operation for a sender as specified in P0443R11 is `execution::submit()`, which accepts a sender and a receiver, binds the receiver to the sender and launches the operation. Once the operation is launched, the sender is responsible for sending the result of the operation to the receiver by calling one of the completion-signalling operations (`set_value()`, `set_error()` or `set_done()`) when the operation eventually completes.

In order to satisfy this contract the `submit()` function needs to ensure that the receiver, or a move-constructed copy of the receiver, remains alive until the operation completes so that the result can be delivered to it. This generally means that a sender that completes asynchronously will need to heap-allocate some storage to hold a copy of the receiver, along with any other state needed from the sender, so that it will remain valid until the operation completes.

While many composed operations can avoid additional allocations by bundling their state into a new receiver passed to a child operation and delegating the responsibility for keeping it alive to the child operation, there will still generally be a need for a heap-allocation for each leaf operation.

However, the same is not true with the design of coroutines and awaitables. An awaitable type is able to inline the storage for its operation-state into the coroutine-frame of the awaiting coroutine by returning a temporary object from its `operator co_await()`, avoiding the need to heap-allocate this object internally.

We found that, by taking a similar approach with sender/receiver and defining a basis operation that lets the sender return its operation-state as an object to the caller, the sender is able to delegate the responsibility for deciding where the operation-state object should be allocated to the caller instead of having to heap-allocate it itself internally.

This allows the caller to choose the most appropriate location for the operation-state of an operation it's invoking. For example, an algorithm like `sync_wait()` might choose to store it on the stack, an `operator co_await()` algorithm might choose to store it as a local variable within the coroutine frame, while a sender algorithm like `via()` might choose to store it inline in the parent operation-state as a data-member.

The core change that this paper proposes is refining the sender concept to be defined in terms of two new basis operations:

- `connect(sender auto&&, receiver auto&&) -> operation_state`
  Connects a sender to a receiver and returns the operation-state object that stores the state of that operation.
- `start(operation_state auto&) noexcept -> void`
  Starts the operation (if not already started). An operation is not allowed to signal completion until it has been started.

There are several other related changes in support of this:

- Retain and redefine the `submit()` operation as a customisable algorithm that has a default implementation in terms of `connect()` and `start()`.
- Add an `operation_state` concept.
- Add two new type-traits queries:
  `connect_result_t<S, R>`
  `is_nothrow_receiver_of_v<R, An...>`

In addition to these changes, this paper also incorporates a number of bugfixes to wording in P0443R11 discovered while drafting these changes.

Motivation

This paper proposes a refinement of the sender/receiver design to split out the `submit()` operation into two more fundamental basis operations; `connect()`, which takes a sender and a receiver and returns an object that contains the state of that async operation, and `start()`, which is used to launch the operation.

There are a number of motivations for doing this, each of which will be explored in more detail below:

- It eliminates the need for additional heap-allocations when awaiting senders within a coroutine, allowing the operation-state to be allocated as a local variable in the coroutine frame.
- It allows composed operations to be defined that do not require any heap allocations. This should allow usage of a reasonable subset of async algorithms in contexts that do not normally allow heap-allocations, such as embedded or real-time systems.
- It allows separating the preparation of a sender for execution from the actual invocation of that operation, satisfying one of the desires expressed in P1658R0.
- It makes it easier and more efficient to satisfy the sender/receiver contract in the presence of exceptions during operation launch.

Lifetime impedence mismatch with coroutines

The paper "Unifying asynchronous APIs in the C++ standard library" P1341R0 looked at the interoperability of sender/receiver with coroutines and showed how senders could be adapted to become awaitables and how awaitables could be adapted to become senders.

However, as P1341R0 identified, adapting between sender/awaitable (in either direction) typically incurs an additional heap-allocation. This is due to senders and awaitables generally having inverted ownership models.

The existing sender/receiver ownership model

With the `submit()`-based asynchronous model of sender/receiver, the `submit()` implementation cannot typically assume that either the sender or the receiver passed to it will live beyond the call to `submit()`. This means for senders that complete asynchronously the implementation of `submit()` will typically need to allocate storage to hold the receiver (so it can deliver the result) as well as any additional state needed by the sender for the duration of the operation. This state is often refered to as the "operation state".

See Example 2 in Appendix A.

Note that some senders may be able to delegate the allocation of the operation-state to a child operation's `submit()` implementation by wrapping up the the receiver and other state into a new

receiver wrapper and passing this wrapper to the `submit()` call of the child operation.
See [Example 1 in Appendix A].

This delegation can be recursively composed, potentially allowing the state of an entire chain of operations to be aggregated into a single receiver object passed to the leaf operation. However, leaf-operations will typically still need to allocate as, by definition of being a leaf operation, they won't have any other senders they can delegate to.

In this model, the leaf operation allocates and owns storage required to store the operation state and the leaf operation is responsible for ensuring that this storage remains alive until the operation completes.

So in the sender/receiver model we can coalesce allocations for a chain of operations and have the the allocation performed only by the leaf-operation. Note that for an operation that is composed of multiple leaf operations, however, it will still typically require multiple heap-allocations over the lifetime of the operation.

The coroutine ownership model

With coroutines the ownership model is reversed.

An asynchronous operation is represented using an awaitable object when using coroutines instead of a sender. The user passes the awaitable object to a `co_await` expression which the compiler translates into a sequence of calls to various customisation points.

The compiler translates the expression '`co_await` *`expr`*' expression into something roughly equivalent to the following (some casts omitted for brevity):

```
// 'co_await expr' becomes (roughly)
decltype(auto) __value = expr;
decltype(auto) __awaitable = promise.await_transform(__value);
decltype(auto) __awaiter = __awaitable.operator co_await();
if (!__awaiter.await_ready()) {
  // <suspend-coroutine>
  __awaiter.await_suspend(coroutine_handle<promise_type>::from_promise(promise));
  // <return-to-caller-or-resumer>
}
// <resume-point>
__awaiter.await_resume(); // This produces the result of the co_await expression
```

When a coroutine is suspended at a suspension point, the compiler is required to maintain the lifetime of any objects currently in-scope - execution returns to the caller/resumer without exiting any scopes of the coroutine). The compiler achieves this by placing any objects whose lifetime spans a suspension point into the coroutine-frame, which is typically allocated on the heap instead of on the stack, and thus can persist beyond the coroutine suspending and returning execution to its caller/resumer.

The important thing to note in the expansion of a `co_await` expression above is that the awaitable object has the opportunity to return an object from its `operator co_await()` method and this return-value becomes a temporary object whose lifetime extends until the end of the full-expression (ie. at the next semicolon). By construction this object will span the suspend-point (`await_ready()` is called before the suspend-point and `await_resume()` is called after the suspend-point) and so the compiler will ensure that storage for the awaiter object is reserved in the coroutine frame of the awaiting coroutine.

Implementations of awaitable types that represent async operations can use this behaviour to their advantage to externalise the allocation of the operation-state by storing the operation-state inline in the awaiting coroutine's coroutine-frame, thus avoiding the need for an additional heap-allocation to store it.

See Example 4 in Appendix A which shows an implementation of a simple allocation-free executor that uses this technique.

This same strategy of inlining storage of child operation's state into the storage for parent operation also occurs when the compiler applies the coroutine heap-allocation elision optimisation[1]. This optimisation works by allowing the compiler to elide heap-allocations for child coroutine-frames whose lifetimes are strictly nested within the lifetime of the caller by inlining the allocation into storage space reserved for it in the parent coroutine-frame.

Taken to its limit, this strategy tends towards a single allocation per high-level operation that contains enough storage for the entire tree of child operations (assuming the storage requirements of the child operations can be statically calculated by the compiler).

Comparing Sender/Receiver and Coroutine Lifetime Models

Taking a step-back we can make some comparisons of the differences of ownership/lifetime models in submit()-based sender/receiver and coroutines/awaitables:

Sender/Receiver	**Coroutines/Awaitables**
Coalesces allocations/state into child operations by wrapping receivers.	Coalesces allocations into parent operations by returning state from `operator co_await()` and by HALO inlining child coroutine-frames.
Tends towards a single allocation for each leaf-level operation.	Tends towards a single allocation per top-level operation.
Type of operation-state is hidden from consumer - an internal implementation detail.	Type of operation-state is exposed to caller allowing its storage to be composed/inlined into parent operation-state.
Producer is responsible for keeping operation-state alive until the operation completes	Consumer is responsible for keeping the operation-state alive until the operation completes

[1] P0981R0 - Halo: coroutine Heap Allocation eLision Optimization: the joint response (R.Smith, G.Nishanov)

and destroying the operation-state after it completes.	and destroying the operation-state after it completes.
Often requires moving state of higher-level operations between operation-states of different leaf operations many times as different leaf operations come and go.	Allows storing state of higher-level operations in a stable location (the higher-level operation-state) and passing references to that operation-state into child operations (eg. via the coroutine_handle)
Higher-level operations will often need a number of separate heap-allocations over its lifetime as different leaf operations come and go. Allows dynamically adjusting memory usage over time, potentially reducing overall memory pressure.	Higher-level operations tend to allocate a single larger allocation, reducing the overall number of allocations, but some of this storage may go unused during some parts of the operation, potentially leading to higher memory pressure in some cases.

Adapting between sender/receiver and coroutines

One of the goals for the sender/receiver design has been to integrate well with coroutines, allowing applications to write asynchronous code in a synchronous style, using the co_await keyword to suspend the coroutine until the asynchronous operation completes.

The paper P1341R0 showed that it is possible to adapt typed-senders to be awaitable and that it's possible to adapt awaitables to become senders. It also discussed how the inverted ownership model resulted in the overhead of an extra heap-allocation whenever we do this.

When we adapt an awaitable to become a sender we need to heap-allocate a new coroutine-frame that can co_await the awaitable, get the result and then pass the result to a receiver. This coroutine-frame is not generally eligible for the heap-allocation elision optimisation (HALO) as the lifetime of the coroutine is not nested within the lifetime of the caller.

When we adapt a sender to become an awaitable, the sender will generally need to heap-allocate the operation-state at the leaf-operation as the sender does not know that the coroutine will implicitly keep the sender and receiver passed to `submit()` alive beyond the call to `submit()`.

The paper P1341R0 thus proposed to make the core concept for representing asynchronous operations a Task, which required implementations to provide both the sender and awaitable interfaces so that tasks could be used either in code that used senders or in code that used coroutines interchangeably. Implementations could provide one of the implementations and the other would have a default implementation provided, albeit with some overhead, or it could provide native implementations of both sender and awaitable interfaces to achieve better performance.

There were a few downsides to this approach, however.

- It forced a dependency of the core concepts on coroutines (`operator co_await()` and `coroutine_handle` type) and this meant that implementers that may not be able to initially implement coroutines for their platforms would be unable to implement the core asynchronous concepts.
- To achieve the best performance for both sender/receiver and coroutines would require implementing every algorithm twice - once under sender/receiver using its ownership model and once under coroutines for its ownership model.
  This would not only be required for your algorithm but for the entire closure of algorithms that your algorithm is built on.
  Having to implement two versions of each algorithm places a high burden on implementers of these algorithms.

Thus, we no longer recommend pursuing the Task concept that requires both coroutines and sender/receiver interfaces to be implemented.

The changes proposed by this paper change the ownership model of sender/receiver to be the same as that of coroutines. This allows us to instead build a generic implementation of `operator co_await()` that can work with any `typed_sender` and that does not require any additional heap-allocations.

This eliminates the need to implement async algorithms twice to be able to get efficient usage with both coroutines and senders. An async algorithm can just implement the sender-interface and can rely on the default `operator co_await()` implementation for senders to allow it to be efficiently used in `co_await` expressions.

Note that a particular type that implements the sender concept can still choose to provide a custom implementation of `operator co_await()` if desired.

Simplifying exception-safe implementations of sender algorithms

The semantics of the `submit()` method as described in P0443R11 required that the implementation of `submit()` would eventually call one of the receiver methods that indicates completion of the operation if `submit()` returns normally.

While the specification was silent on the semantics if `submit()` were to exit with an exception, the intent was that `submit()` would not subsequently invoke (or have successfully invoked) any of the completion-signalling functions on the receiver.

This allows the caller to catch the exception thrown out of `submit()` if desired and either handle the error or pass the error onto the caller's receiver by calling `set_error()`.

However, implementations of algorithms that are themselves senders must be careful when implementing this logic to ensure that they are able to correctly handle an exception propagating from the call to `submit()`. If it naively moves its receiver into the receiver wrapper it passes to a child operation's `submit()` function then if that `submit()` function invocation throws then the caller may be left with its receiver now being in a moved-from state and thus not being able to deliver a result to its receiver.

A good demonstration of the problem is in the implementation of a `sequence()` algorithm that takes two senders and launches the two operations in sequence - only calling `submit()` on the second sender once the first sender has completed with `set_value()`.

Example 1 in Appendix B highlights the problem with a naive implementation of this algorithm.

One strategy for implementing a correct, exception-safe implementation is for the caller to store its receiver in a stable location and then only pass a pointer or reference to that receiver to the receiver-wrapper passed to the child operation's `submit()` function.

However, under the sender/receiver design described in P0443R11, getting access to a stable location for the receiver would typically require a heap-allocation.

Example 2 in Appendix B shows a solution that makes use of a shared_ptr to to allow correctly handling exceptions that might be thrown from the second sender's submit().

The changes to the sender/receiver design proposed by this paper provides a solution to this that does not require a heap-allocation to store the receiver. The receiver can be stored in the operation-state object returned from `connect()`, which the caller is required to store in a stable location until the operation completes. Then we can pass a receiver-wrapper into the child operation that just holds a pointer to this operation-state and can get access to the receiver via that pointer.

Example 3 in Appendix B shows the alternative `connect()`/`start()`-based implementation of the `sequence()` algorithm for comparison.

This allows some algorithms to further reduce the number of heap-allocations required to implement them compared to the `submit()`-based implementation.

Ability to separate resource allocation for operation from launch

The paper P1658R0 "Suggestions for Consensus on Executors" suggested factoring submit() into more basic operations - a finalize() and a start().

P1658R0 makes the observation that the submit() operation signals that the sender is 1. ready for execution and 2. may be executed immediately, and suggests that it would be valuable to be able to decouple the cost of readying a sender from its launch.

Examples of expensive finalization mentioned in P1658R0 include:

- Memory allocation of temporary objects required during execution
- Just-in-time compilation of heterogeneous compute kernels
- Instantiation of task graphs
- Serialization of descriptions of work to be executed remotely

Being able to control where the expensive parts of launching an operation occurs is important for performance-conscious code.

Splitting the `submit()` operation up into a `connect()` and `start()` operations should make this possible.

Wording

This wording change is described as a delta to P0443R11.

Update subsection “Header `<execution>` synopsis” as follows:

```
// Customization points:

inline namespace unspecified {
  inline constexpr unspecified set_value = unspecified;

  inline constexpr unspecified set_done = unspecified;

  inline constexpr unspecified set_error = unspecified;

  inline constexpr unspecified execute = unspecified;

  inline constexpr unspecified connect = unspecified;

  inline constexpr unspecified start = unspecified;

  inline constexpr unspecified submit = unspecified;

  inline constexpr unspecified schedule = unspecified;

  inline constexpr unspecified bulk_execute = unspecified;
}

template<class S, class R>
  using connect_result_t = invoke_result_t<decltype(connect), S, R>;

template<class, class> struct as-receiver; // exposition only
```

```
template<class, class> struct as-invocable; // exposition only

// Concepts:
template<class T, class E = exception_ptr>
  concept receiver = see-below;

template<class T, class... An>
  concept receiver_of = see-below;

template<class R, class... An>
  inline constexpr bool is_nothrow_receiver_of_v =
    receiver_of<R, An...> &&
    is_nothrow_invocable_v<decltype(set_value), R, An...>;

template<class O>
  concept operation_state = see-below;

template<class S>
  concept sender = see-below;

template<class S>
  concept typed_sender = see-below;

... as before

// Sender and receiver utilities type
class sink_receiver;

namespace unspecified { struct sender_base {}; }
using unspecified::sender_base;

template<class S> struct sender_traits;
```

Change 1.2.2 “Invocable archetype” as follows:

The name `execution::invocable_archetype` is an implementation-defined type ~~that, along with any argument pack, models `invocable`~~such that `invocable<execution::invocable_archetype&>` is `true`.

A program that creates an instance of execution::invocable_archetype is ill-formed.

Change 1.2.3.4 `execution::execute`, bullet 3 as follows:

Otherwise, if `F` is not an instance of *`as-invocable<R, E>`* for some type *`R`*, and `invocable<remove_cvref_t<F>&> && sender_to<E,` *`as-receiver`*`<remove_cvref_t<F>, E>>` is `true`, `execution::submit(e,` *`as-receiver`*`<remove_cvref_t<F>, E>`~~`(`~~`{forward<F>(f)`~~`)`~~`})` ~~if `E` and *`as-receiver<F>`* model `sender_to`~~, where *`as-receiver`* is some implementation-defined class template equivalent to:

```
template<~~invocable~~class F, class>
struct as-receiver {
~~private:~~
  ~~using invocable_type = std::remove_cvref_t<F>;~~
  ~~invocable_type~~F f_;
~~public:~~
  ~~explicit as-receiver(invocable_type&& f)~~
    ~~: f_(move_if_noexcept(f)) {}~~
  ~~explicit as-receiver(const invocable_type& f) : f_(f) {}~~
  ~~as-receiver(as-receiver&& other) = default;~~
  void set_value() noexcept(is_nothrow_invocable_v<F&>) {
    invoke(f_);
  }
  [[noreturn]] void set_error(std::exception_ptr) noexcept {
    terminate();
  }
  void set_done() noexcept {}
};
```

Before subsection 1.2.3.5 `execution::submit`, add the following two subsections, and renumber the subsequent subsections.

1.2.3.x `execution::connect`

The name `execution::connect` denotes a customization point object. The expression `execution::connect(S, R)` for some subexpressions `S` and `R` is expression-equivalent to:

- `S.connect(R)`, if that expression is valid, if its type satisfies `operation_state`, and if the type of `S` satisfies `sender`.
- Otherwise, `connect(S, R)`, if that expression is valid, if its type satisfies `operation_state`, and if the type of `S` satisfies `sender`, with overload resolution performed in a context that includes the declaration

  ```
  void connect();
  ```

  and that does not include a declaration of `execution::connect`.

- Otherwise, *`as-operation`*`{S, R}`, if `R` is not an instance of *`as-receiver`*`<`*`F`*`, S>` for some type *`F`*, and if `receiver_of<T> && executor_of<U,` *`as-invocable`*`<T, S>>` is `true` where `T` is the type of `R` without *cv*-qualification and `U` is the type of `S` without *cv*-qualification, and where *`as-operation`* is an implementation-defined class equivalent to

```
struct as-operation {
  U e_;
  T r_;
  void start() noexcept try {
    execution::execute(
       std::move(e_), as-invocable<T, S>{r_});
  } catch(...) {
    execution::set_error(
      std::move(r_), current_exception());
  }
};
```

  and *`as-invocable`* is a class template equivalent to the following:

```
template<class R, class>
struct as-invocable {
  R* r_ ;
  explicit as-invocable(R& r) noexcept
    : r_(std::addressof(r)) {}
  as-invocable(as-invocable&& other) noexcept
    : r_(std::exchange(other.r_, nullptr)) {}
  ~as-invocable() {
    if(r_)
      execution::set_done((R&&) *r_);
  }
  void operator()() & noexcept {
    try {
      execution::set_value((R&&) *r_);
    } catch(...) {
      execution::set_error(
        (R&&) *r_, current_exception());
    }
    r_ = nullptr;
  }
};
```

- Otherwise, `execution::connect(S, R)` is ill-formed.

1.2.3.x `execution::start`

The name `execution::start` denotes a customization point object. The expression `execution::start(O)` for some lvalue subexpression `O` is expression-equivalent to:

- `O.start()`, if that expression is valid.
- Otherwise, `start(O)`, if that expression is valid, with overload resolution performed in a context that includes the declaration

```
void start();
```

  and that does not include a declaration of `execution::start`.

- Otherwise, `execution::start(O)` is ill-formed.

Change 1.2.3.5 "`execution::submit`" in recognition of the fact that `submit` is a customizable algorithm that has a default implementation in terms of `connect`/`start` as follows:

The name `execution::submit` denotes a customization point object.

~~A receiver object is *submitted for execution via a sender* by scheduling the eventual evaluation of one of the receiver's value, error, or done channels.~~

For some subexpressions `s` and `r`, let `S` be a type such that `decltype((s))` is `S` and let `R` be a type such that `decltype((r))` is `R`. The expression `execution::submit(s, r)` is ill-formed if ~~R does not model receiver, or if S does not model either sender or executor~~`sender_to<S, R>` is not `true`. Otherwise, it is expression-equivalent to:

- `s.submit(r)`, if that expression is valid and `S` models `sender`. If the function selected does not submit the receiver object `r` via the sender `s`, the program is ill-formed with no diagnostic required.
- Otherwise, `submit(s, r)`, if that expression is valid and `S` models `sender`, with overload resolution performed in a context that includes the declaration

```
void submit();
```

  and that does not include a declaration of `execution::submit`. If the function selected by overload resolution does not submit the receiver object `r` via the sender `s`, the program is ill-formed with no diagnostic required.

- Otherwise, execution::execute(s,*as-invocable*<R>(forward<R>(r))) if S and *as-invocable*<R> model executor, where *as-invocable* is some implementation-defined class template equivalent `to:`

```
template<receiver R>
struct as-invocable {
```

```
private:
  using receiver_type = std::remove_cvref_t<R>;
  std::optional<receiver_type> r_ {};
  void try_init_(auto&& r) {
    try {
      r_.emplace((decltype(r)&&) r);
    } catch(...) {
      execution::set_error(r, current_exception());
    }
  }
public:
  explicit as-invocable(receiver_type&& r) {
    try_init_(move_if_noexcept(r));
  }
  explicit as-invocable(const receiver_type& r) {
    try_init_(r);
  }
  as-invocable(as-invocable&& other) {
    if(other.r_) {
      try_init_(move_if_noexcept(*other.r_));
      other.r_.reset();
    }
  }
  ~as-invocable() {
    if(r_)
      execution::set_done(*r_);
  }
  void operator()() {
    try {
      execution::set_value(*r_);
    } catch(...) {
      execution::set_error(*r_, current_exception());
    }
    r_.reset();
  }
};
```

- Otherwise, `execution::start((new submit-receiver<S, R>{s, r})->state_)`, where *`submit-receiver`* is an implementation-defined class template equivalent to

```
template<class S, class R>
struct submit-receiver {
```

```
    struct wrap {
      submit-receiver* p_;
      template<class...As>
        requires receiver_of<R, As...>
      void set_value(As&&... as) && {
        execution::set_value(
          std::move(p_->r_), (As&&) as...);
        delete p_;
      }
      template<class E>
        requires receiver<R, E>
      void set_error(E&& e) && noexcept {
        execution::set_error(
          std::move(p_->r_), (E&&) e);
        delete p_;
      }
      void set_done() && noexcept {
        execution::set_done(std::move(p_->r_));
        delete p_;
      }
    };
    remove_cvref_t<R> r_;
    connect_result_t<S, wrap> state_;
    submit-receiver(S&& s, R&& r)
      : r_((R&&) r)
      , state_(execution::connect((S&&) s, wrap{this}))
    {}
  };
```

Change 1.2.3.6 `execution::schedule` as follows:

The name `execution::schedule` denotes a customization point object. For some subexpression `s`, let `S` be a type such that `decltype((s))` is `S`. The expression `execution::schedule(`~~`S`~~`s)` ~~for some subexpression `S`~~ is expression-equivalent to:

- ~~`S`~~`s.schedule()`, if that expression is valid and its type ~~N~~ models `sender`.
- Otherwise, `schedule(`~~`S`~~`s)`, if that expression is valid and its type ~~N~~ models `sender` with overload resolution performed in a context that includes the declaration

  ```
  void schedule();
  ```

  and that does not include a declaration of `execution::schedule`.
- ~~Otherwise, *decay-copy*(S) if the type S models sender.~~

- Otherwise, *as-sender*<remove_cvref_t<S>>{s} if S satisfies executor, where *as-sender* is an implementation-defined class template equivalent to

```
template<class E>
struct as-sender {
private:
  E ex_;
public:
  template<template<class...> class Tuple,
           template<class...> class Variant>
    using value_types = Variant<Tuple<>>;
  template<template<class...> class Variant>
    using error_types = Variant<std::exception_ptr>;
  static constexpr bool sends_done = true;

  explicit as-sender(E e)
    : ex_((E&&) e) {}
  template<class R>
    requires receiver_of<R>
  connect_result_t<E, R> connect(R&& r) && {
    return execution::connect((E&&) ex_, (R&&) r);
  }
  template<class R>
    requires receiver_of<R>
  connect_result_t<const E &, R> connect(R&& r) const & {
    return execution::connect(ex_, (R&&) r);
  }
};
```

- Otherwise, execution::schedule(~~S~~s) is ill-formed.

Merge subsections 1.2.4 and 1.2.5 into a new subsection "Concepts receiver and receiver_of" and change them as follows:

~~XXX TODO The receiver concept...~~A receiver represents the continuation of an asynchronous operation. An asynchronous operation may complete with a (possibly empty) set of values, an error, or it may be cancelled. A receiver has three principal operations corresponding to the three ways an asynchronous operation may complete: set_value, set_error, and set_done. These are collectively known as a receiver's *completion-signal operations*.

```
// exposition only:
template<class T>
inline constexpr bool is-nothrow-move-or-copy-constructible =
  is_nothrow_move_constructible<T> ||
   copy_constructible<T>;

template<class T, class E = exception_ptr>
```

```
concept receiver =
  move_constructible<remove_cvref_t<T>> &&
  constructible_from<remove_cvref_t<T>, T> &&
  (is-nothrow-move-or-copy-constructible<remove_cvref_t<T>>) &&
  requires(remove_cvref_t<T>&& t, E&& e) {
    { execution::set_done((T&&) tstd::move(t)) } noexcept;
    { execution::set_error((T&&) tstd::move(t), (E&&) e) }
noexcept;
  };

template<class T, class... An>
concept receiver_of =
  receiver<T> &&
  requires(remove_cvref_t<T>&& t, An&&... an) {
    execution::set_value((T&&) tstd::move(t), (An&&) an...);
  };
```

The receiver's completion-signal operations have semantic requirements that are collectively known as the *receiver contract*, described below:

- None of a receiver's completion-signal operations shall be invoked before `execution::start` has been called on the operation state object that was returned by `execution::connect` to connect that receiver to a sender.
- Once `execution::start` has been called on the operation state object, exactly one of the receiver's completion-signal operations shall complete non-exceptionally before the receiver is destroyed.
- If `execution::set_value` exits with an exception, it is still valid to call `execution::set_error` or `execution::set_done` on the receiver.

Once one of a receiver's completion-signal operations has completed non-exceptionally, the receiver contract has been satisfied.

Before 1.2.6 "Concepts `sender` and `sender_to`," insert a new section 1.2.x "Concept `operation_state`" as follows:

1.2.x Concept `operation_state`

```
template<class O>
concept operation_state =
  destructible<O> &&
  is_object_v<O> &&
  requires (O& o) {
```

```
    {execution::start(o)} noexcept;
  };
```

An object whose type satisfies `operation_state` represents the state of an asynchronous operation. It is the result of calling `execution::connect` with a `sender` and a `receiver`.

`execution::start` may be called on an `operation_state` object at most once. Once `execution::start` has been called on it, the `operation_state` must not be destroyed until one of the receiver's completion-signal operations has begun executing, provided that invocation will not exit with an exception.

The start of the invocation of `execution::start` shall strongly happen before [intro.multithread] the invocation of one of the three receiver operations.

`execution::start` may or may not block pending the successful transfer of execution to one of the three receiver operations.

Change 1.2.6 "Concepts `sender` and `sender_to`" as follows

XXX TODO The sender and sender_to concepts...

~~Let *sender-to-impl* be the exposition-only concept~~

```
template<class S, class R>
concept sender-to-impl =
  requires(S&& s, R&& r) {
    execution::submit((S&&) s, (R&&) r);
  };
```

~~Then,~~

```
template<class S>
concept sender =
  move_constructible<remove_cvref_t<S>> &&
  sender-to-impl<S, sink_receiver>;
  !requires {
    typename sender_traits<remove_cvref_t<S>>::
      __unspecialized; // exposition only
  };

template<class S, class R>
concept sender_to =
  sender<S> &&
  receiver<R> &&
  sender-to-impl<S, R>;
```

```
requires (S&& s, R&& r) {
  execution::connect((S&&) s, (R&&) r);
};
```

None of these operations shall introduce data races as a result of concurrent invocations of those functions from different threads.

An sender type's destructor shall not block pending completion of the submitted function objects. [*Note:* The ability to wait for completion of submitted function objects may be provided by the associated execution context. *--end note*]

~~In addition to the above requirements, types `S` and `R` model `sender_to` only if they satisfy the requirements from the Table below.~~

~~In the Table below,~~

- ~~`s` denotes a (possibly const) sender object of type `S`,~~
- ~~`r` denotes a (possibly const) receiver object of type `R`.~~

~~**Expression**~~	~~**Return Type**~~	~~**Operational semantics**~~
~~`execution::submit(s, r)`~~	~~`void`~~	~~If `execution::submit(s, r)` exits without throwing an exception, then the implementation shall invoke exactly one of `execution::set_value(rc, values...)`, `execution::set_error(rc, error)` or `execution::set_done(rc)` where `rc` is either `r` or an object moved from `r`. If any of the invocations of `set_value` or `set_error` exits via an exception then it is valid to call to either `set_done(rc)` or `set_error(rc, E)`, where `E` is an `exception_ptr` pointing to an unspecified exception object. `submit` may or may not block pending the successful transfer of execution to one of the three receiver operations.~~

		~~The start of the invocation of submit strongly happens before [intro.multithread] the invocation of one of the three receiver operations.~~

In subsection 1.2.7 “Concept `typed_sender`”, change the definition of the `typed_sender` concept as follows:

```
template<class S>
  concept typed_sender =
    sender<S> &&
    has-sender-types<sender_traits<remove_cvref_t<S>>>;
```

Change 1.2.8 “Concept `scheduler`” as follows:

XXX TODO The scheduler concept...

```
template<class S>
  concept scheduler =
    copy_constructible<remove_cvref_t<S>> &&
    equality_comparable<remove_cvref_t<S>> &&
    requires(E&& e) {
      execution::schedule((S&&)s);
    }; // && sender<invoke_result_t<execution::schedule, S>>
```

None of a scheduler's copy constructor, destructor *[... as before]*

[...]

~~`execution::submit(N, r)`,~~ `execution::start(o)`, where `o` is the result of a call to `execution::connect(N, r)` for some receiver object `r`, is required to eagerly submit `r` for execution on an execution agent that `s` creates for it. Let `rc` be `r` or an object created by copy or move construction from `r`. The semantic constraints on the sender `N` returned from a scheduler s's `schedule` function are as follows:

- If `rc`'s `set_error` function is called in response to a submission error, scheduling error, or other internal error, let `E` be an expression that refers to that error if `set_error(rc, E)` is well-formed; otherwise, let `E` be an `exception_ptr` that refers to that error. [*Note*: `E` could be the result of calling `current_exception` or `make_exception_ptr` — *end note*] The scheduler calls `set_error(rc, E)` on an unspecified weakly-parallel execution agent ([*Note*: An invocation of set_error on a receiver is required to be noexcept — *end note*]), and

- If `rc`'s `set_error` function is called in response to an exception that propagates out of the invocation of `set_value` on `rc`, let `E` be `make_exception_ptr(receiver_invocation_error{})` invoked from within a catch clause that has caught the exception. The executor calls `set_error(rc, E)` on an unspecified weakly-parallel execution agent, and
- A call to `set_done(rc)` is made on an unspecified weakly-parallel execution agent ([*Note*: An invocation of a receiver's set_done function is required to be noexcept — *end note*]).

[*Note*: The senders returned from a scheduler's `schedule` function have wide discretion when deciding which of the three receiver functions to call upon submission. — *end note*]

Change subsection 1.2.9 Concepts "`executor` and `executor_of`" as follows to reflect the fact that the operational semantics of `execute` require a copy to be made of the invocable:

XXX TODO The `executor` and `executor_of` concepts...

Let *`executor-of-impl`* be the exposition-only concept

```
template<class E, class F>
  concept executor-of-impl =
    invocable<remove_cvref_t<F>&> &&
    constructible_from<remove_cvref_t<F>, F> &&
    move_constructible<remove_cvref_t<F>> &&
    copy_constructible<E> &&
    is_nothrow_copy_constructible_v<E> &&
    is_nothrow_destructible_v<E> &&
    equality_comparable<E> &&
    requires(const E& e, F&& f) {
      execution::execute(e, (F&&)f);
    };
```

Then,

```
template<class E>
  concept executor =
    executor-of-impl<E, execution::invocable_archetype>;

template<class E, class F>
  concept executor_of =
    executor<E> &&
    executor-of-impl<E, F>;
```

Remove subsection 1.2.10.1 “Class `sink_receiver`”.

Change subsection 1.2.10.2 “Class template `sender_traits`” as follows:

The class template `sender_traits` can be used to query information about a sender; in particular, what values and errors it sends through a receiver's value and error channel, and whether or not it ever calls `set_done` on a receiver.

```
template<class S>
  struct sender-traits-base {}; // exposition-only

template<class S>
    requires (!same_as<S, remove_cvref_t<S>>)
  struct sender-traits-base
    : sender_traits<remove_cvref_t<S>> {};

template<class S>
    requires same_as<S, remove_cvref_t<S>> &&
    sender<S> && has-sender-traits<S>
  struct sender-traits-base<S> {
    template<template<class...> class Tuple,
             template<class...> class Variant>
      using value_types =
        typename S::template value_types<Tuple, Variant>;
    template<template<class...> class Variant>
      using error_types =
        typename S::template error_types<Variant>;
    static constexpr bool sends_done = S::sends_done;
  };

template<class S>
  struct sender_traits : sender-traits-base<S> {};
```

The primary `sender_traits<S>` class template is defined as if inheriting from an implementation-defined class template *`sender-traits-base<S>`* defined as follows:

- Let *`has-sender-types`* be an implementation-defined concept equivalent to:

```
template<template<template<class...> class,
                  template<class...> class> class>
  struct has-value-types; // exposition only

template<template<template<class...> class> class>
```

```
    struct has-error-types; // exposition only

  template<class S>
    concept has-sender-types =
      requires {
        typename
          has-value-types<S::template value_types>;
        typename
          has-error-types<S::template error_types>;
        typename bool_constant<S::sends_done>;
      };
```

If `has-sender-types<S>` is `true`, then `sender-traits-base` is equivalent to:

```
  template<class S>
    struct sender-traits-base {
      template<template<class...> class Tuple,
               template<class...> class Variant>
        using value_types =
          typename S::template value_types<Tuple,
                                           Variant>;
      template<template<class...> class Variant>
        using error_types =
          typename S::template error_types<Variant>;
      static constexpr bool sends_done = S::sends_done;
    };
```

- Otherwise, let `void-receiver` be an implementation-defined class type equivalent to

```
  struct void-receiver { // exposition only
    void set_value() noexcept;
    void set_error(exception_ptr) noexcept;
    void set_done() noexcept;
  };
```

If `executor_of<S, as-invocable<void-receiver, S>>` is `true`, then `sender-traits-base` is equivalent to

```
  template<class S>
    struct sender-traits-base {
      template<template<class...> class Tuple,
               template<class...> class Variant>
        using value_types = Variant<Tuple<>>;
      template<template<class...> class Variant>
        using error_types = Variant<exception_ptr>;
      static constexpr bool sends_done = true;
    };
```

- Otherwise, if `derived_from<S, sender_base>` is `true`, then *`sender-traits-base`* is equivalent to

```
template<class S>
  struct sender-traits-base {};
```

- Otherwise, *`sender-traits-base`* is equivalent to

```
template<class S>
  struct sender-traits-base {
    using __unspecialized = void; // exposition only
  };
```

Change 1.5.4.5 “`static_thread_pool` sender execution functions” as follows:

In addition to conforming to the above specification, `static_thread_pool` ~~executors~~ schedulers’ senders shall conform to the following specification.

```
class C
{
  public:
    template<template<class...> class Tuple,
             template<class...> class Variant>
      using value_types = Variant<Tuple<>>;
    template<template<class...> class Variant>
      using error_types = Variant<>;
    static constexpr bool sends_done = true;
```

    `template<`~~`class Receiver`~~`receiver_of R>`
      ~~`void`~~*`see-below`* ~~`submit`~~`connect(`~~`Receiver`~~`R&& r) const;`
`};`

`C` is a type satisfying the typed_`sender` requirements.

    `template<`~~`class Receiver`~~`receiver_of R>`
      ~~`void`~~*`see-below`* ~~`submit`~~`connect(`~~`Receiver`~~`R&& r) const;`

Returns: An object whose type satisfies the `operation_state` concept.

Effects: ~~Submits~~ When `execution::start` is called on the returned operation state, the receiver `r` is submitted for execution on the `static_thread_pool` according to the the properties established for `*this`. ~~l~~Let `e` be an object of type `exception_ptr`~~;~~, then static_thread_pool will evaluate one of set_value(r), set_error(r, e), or set_done(r).

Appendix

Appendix A - Examples of status quo lifetime/ownership

Example 1: Delegating responsibility for allocating storage to a child sender

```
template<typename Func, typename Inner>
struct transform_sender {
  Inner inner_;
  Func func_;

  template<typename Receiver>
  struct transform_receiver {
    Func func_;
    Receiver receiver_;

    template<typename... Values>
    void set_value(Values&&... values) {
      receiver_.set_value(std::invoke(func_, (Values&&)values...));
    }
    template<typename Error>
    void set_error(Error&& error) {
      receiver_.set_error((Error&&)error);
    }
    void set_done() {
      receiver_.set_done();
    }
  };

  template<typename Receiver>
  void submit(Receiver r) {
    // Here we delegate responsibility for storing the receiver, 'r'
    // and a copy of 'func_' to the implementation of inner_.submit() which
    // is required to store the transform_receiver we pass to it.
    inner_.submit(transform_receiver<Receiver>{func_, std::move(r)});
  }
};
```

Example 2: A simple execution context that shows the allocation necessary for operation-state for the 'schedule()' operation.

```
class simple_execution_context {
  struct task_base {
    virtual void execute() noexcept = 0;
    task_base* next;
  };

  class schedule_sender {
    simple_execution_context& ctx;
  public:
    explicit schedule_sender(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    template<std::receiver_of Receiver>
    void submit(Receiver&& r) {
      class task final : private task_base {
        std::remove_cvref_t<Receiver> r;
      public:
        explicit task(Receiver&& r) : r((Receiver&&)r) {}
```

```
        void execute() noexcept override {
          try {
            std::execution::set_value(std::move(r));
          } catch (...) {
            std::execution::set_error(std::move(r), std::current_exception());
          }
          delete this;
        }
      };

      // Allocate the "operation-state" needed to hold the receiver
      // and other state (like storage of 'next' field of intrusive list,
      // vtable-ptr for dispatching type-erased implementation)
      task* t = new task{static_cast<Receiver&&>(r)};

      // Enqueue this task to the executor's linked-list of tasks to execute.
      ctx.enqueue(t);
    }
  };

  class scheduler {
    simple_execution_context& ctx;
  public:
    explicit scheduler(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    schedule_sender schedule() const noexcept { return schedule_sender{ctx}; }
  };

public:
  scheduler get_scheduler() noexcept { return scheduler{*this}; }

  // Processes all pending tasks until the queue is empty.
  void drain() noexcept {
    while (head != nullptr) {
      task_base* t = std::exchange(head, head->next);
      t->execute();
    }
  }

private:
  void enqueue(task_base* t) noexcept {
    t->next = std::exchange(head, t);
  }

  task_base* head = nullptr;
};
```

Example 3: The same simple_execution_context as above but this time with the schedule() operation implemented using coroutines and awaitables. Note that it does not require any heap allocations.

```
class simple_execution_context {
  class awaiter {
    friend simple_execution_context;

    simple_execution_context& ctx;
    awaiter* next = nullptr;
    std::coroutine_handle<> continuation;

  public:
    explicit awaiter(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    bool await_ready() const noexcept { return false; }
```

```
    void await_suspend(std::continuation_handle<> h) noexcept {
      continuation = h;
      ctx.enqueue(this);
    }
    void await_resume() noexcept {}
  };

  class schedule_awaitable {
    simple_execution_context& ctx;
  public:
    explicit schedule_awaitable(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    // Return an instance of the operation-state from 'operator co_await()'
    // This is will be placed as a local variable within the awaiting coroutine's
    // coroutine-frame and means that we don't need a separate heap-allocation.
    awaiter operator co_await() const noexcept {
      return awaiter{ctx};
    }
  };

  class scheduler {
    simple_execution_context& ctx;
  public:
    explicit scheduler(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    schedule_awaitable schedule() const noexcept { return schedule_awaitable{ctx}; }
  };

public:
  scheduler get_scheduler() noexcept { return scheduler{*this}; }

  // Processes all pending awaiters until the queue is empty.
  void drain() noexcept {
    while (head != nullptr) {
      awaiter* a = std::exchange(head, head->next);
      a->execute();
    }
  }

private:
  void enqueue(awaiter* a) noexcept {
    a->next = std::exchange(head, a);
  }

  awaiter* head = nullptr;
};
```

Example 4: The same simple_execution_context but this time implemented using the connect/start refinements to the sender/receiver proposed by this paper. This uses similar techniques to the coroutine version above. ie. returning the operation-state to the caller and relying on them to keep the operation-state alive until the operation completes.

```
class simple_execution_context {
  struct task_base {
    virtual void execute() noexcept = 0;
    task_base* next;
  };

  class schedule_sender {
    simple_execution_context& ctx;
```

```
  public:
    explicit schedule_sender(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    template<typename Receiver>
    class operation_state final : private task_base {
      simple_execution_context& ctx;
      std::remove_cvref_t<Receiver> receiver;

      void execute() noexcept override {
        try {
          std::execution::set_value(std::move(receiver));
        } catch (...) {
          std::execution::set_error(std::move(receiver), std::current_exception());
        }
      }

    public:

      explicit operation_state(simple_execution_context& ctx, Receiver&& r)
      : ctx(ctx), receiver((Receiver&&)r) {}

      void start() noexcept & {
        ctx.enqueue(this);
      }
    };

    // Returns the operation-state object to the caller which is responsible for
    // ensuring it remains alive until the operation completes once start() is called.
    template<std::receiver_of Receiver>
    operation_state<Receiver> connect(Receiver&& r) {
      return operation_state<Receiver>{*this, (Receiver&&)r};
    }
  };

  class scheduler {
    simple_execution_context& ctx;
  public:
    explicit scheduler(simple_execution_context& ctx) noexcept : ctx(ctx) {}

    schedule_sender schedule() const noexcept { return schedule_sender{ctx}; }
  };

public:
  scheduler get_scheduler() noexcept { return scheduler{*this}; }

  // Processes all pending tasks until the queue is empty.
  void drain() noexcept {
    while (head != nullptr) {
      task_base* t = std::exchange(head, head->next);
      t->execute();
    }
  }

private:
  void enqueue(task_base* t) noexcept {
    t->next = std::exchange(head, t);
  }

  task_base* head = nullptr;
};
```

Appendix B - Exception-safe sender adapters

Example 1: A sender-adapter that executes two other senders sequentially. This is difficult to get right because of the potential for the submit() method to throw.
This code snippet shows the problem with a naive approach.

```
template<typename First, typename Second>
class sequence_sender {
  First first;
  Second second;

  template<typename Receiver>
  class first_receiver {
    Second second;
    Receiver receiever;

  public:
    explicit first_receiver(Second&& second, Receiver&& recevier)
      noexcept(std::is_nothrow_move_constructible_v<Second> &&
               std::is_nothrow_move_constructible_v<Receiver>)
      : second((Second&&)second), receiver((Receiver&&)receiver) {}

    void set_value() && noexcept {
      try {
        execution::submit(std::move(second), std::move(receiver));
      } catch (...) {
        // BUG: What do we do here?
        //
        // We need to signal completion using 'receiver' but now
        // 'receiver' might be in a moved-from state and so we
        // cannot safely invoke set_error(receiver, err) here.
      }
    }

    template<typename Error>
    void set_error(Error&& e) && noexcept {
      execution::set_error(std::move(receiver), (E&&)e);
    }

    void set_done() && noexcept {
      execution::set_done(std::move(receiver));
    }
  };

public:
  explicit sequence_sender(First first, Second second)
    noexcept(std::is_nothrow_move_constructible_v<First> &&
             std::is_nothrow_move_constructible_v<Second>)
  : first((First&&)first), second((Second&&)second)
  {}

  template<typename Receiver>
  void submit(Receiver receiver) && {
    // If this call to submit() on the first sender throws then
    // we let the exception propagate out without calling the
    // 'receiver'.
    execution::submit(
      std::move(first),
      first_receiver<Receiver>{std::move(second), std::move(receiver)});
  }
```

```
};
```

Example 2: This shows a more correct implementation that makes use of shared_ptr to allow recovery in the case that the submit() on the second sender throws. We pass a copy of the shared_ptr into submit() and also retain a copy that we can use in case submit() throws an exception.

```
template<typename Receiver>
class shared_receiver {
  std::shared_ptr<Receiver> receiver_;

public:
  explicit shared_receiver(Receiver&& r)
    : receiver_(std::make_shared<Receiver>((Receiver&&)r))
  {}

  template<typename... Values>
    requires value_receiver<Receiver, Values...>
  void set_value(Values&&... values) && noexcept(
    is_nothrow_invocable_v<decltype(execution::set_value), Receiver, Values...>) {
    execution::set_value(std::move(*receiver_), (Values&&)values...);
  }

  template<typename Error>
    requires error_receiver<Receiver, Error>
  void set_error(Error&& error) && noexcept {
    exection::set_error(std::move(*receiver_), (Error&&)error);
  }

  void set_done() && noexcept requires done_receiver<Receiver> {
    execution::set_done(std::move(*receiver_));
  }
};

template<typename First, typename Second>
class sequence_sender {
  First first;
  Second second;

  template<typename Receiver>
  class first_receiver {
    Second second;
    shared_receiver<Receiver> receiver;

  public:
    explicit first_receiver(Second&& second, Receiver&& recevier)
      noexcept(std::is_nothrow_move_constructible_v<Second> &&
               std::is_nothrow_move_constructible_v<Receiver>)
      : second((Second&&)second), receiver((Receiver&&)receiver) {}

    void set_value() && noexcept {
      try {
        execution::submit(std::move(second), std::as_const(receiver));
      } catch (...) {
        // We only copied the receiver into submit() so we still have access
        // to the original receiver to deliver the error.
        //
        // Note that we must assume that if submit() throws then it will not
        // have already called any of the completion methods on the receiver.
        execution::set_error(std::move(receiver), std::current_exception());
      }
    }
```

```
    template<typename Error>
    void set_error(Error&& e) && noexcept {
      execution::set_error(std::move(receiver), (E&&)e);
    }

    void set_done() && noexcept {
      execution::set_done(std::move(receiver));
    }
  };

public:
  explicit sequence_sender(First first, Second second)
    noexcept(std::is_nothrow_move_constructible_v<First> &&
             std::is_nothrow_move_constructible_v<Second>)
  : first((First&&)first), second((Second&&)second)
  {}

  template<typename Receiver>
    requires std::execution::sender_to<Second, shared_receiver<Receiver>>
  void submit(Receiver receiver) && {
    // If this call to submit() on the first sender throws then
    // we let the exception propagate out without calling the
    // 'receiver'.
    execution::submit(
      std::move(first),
      first_receiver<Receiver>{std::move(second), std::move(receiver)});
  }
};
```

Example 3: Implementation of the sequence() algorithm using connect()/start()-based senders. Notice that this implementation does not require any heap-allocations to implement correctly.

```
// Helper that allows in-place construction of std::variant element
// using the result of a call to a lambda/function. Relies on C++17
// guaranteed copy-elision when returning a prvalue.
template<std::invocable Func>
struct __implicit_convert {
  Func func;
  operator std::invoke_result_t<Func>() && noexcept(std::is_nothrow_invocable_v<Func>) {
    return std::invoke((Func&&)func);
  }
};
template<std::invocable Func>
__implicit_convert(Func) -> __implicit_convert<Func>;

template<typename First, typename Second>
class sequence_sender {
  template<typename Receiver>
  class operation_state {
    class second_receiver {
      operation_state* state_;
    public:
      explicit second_receiver(operation_state* state) noexcept : state_(state) {}

      template<typename... Values>
        requires std::execution::receiver_of<Receiver, Values...>
      void set_value(Values&&... values) noexcept(std::is_nothrow_invocable_v<
          decltype(std::execution::set_value), Receiver, Values...>) {
        std::execution::set_value(std::move(state_->receiver_), (Values&&)values...);
      }

      template<typename Error>
        requires std::execution::receiver<Receiver, Error>
```

```cpp
      void set_error(Error&& error) noexcept {
        std::execution::set_error(std::move(state_->receiver_), (Error&&)error);
      }

      void set_done() noexcept {
        std::execution::set_done(std::move(state_->receiver_));
      }
    };

    class first_receiver {
      operation_state* state_;
    public:
      explicit first_receiver(operation_state* state) noexcept : state_(state) {}

      void set_value() noexcept {
        auto* state = state_;
        try {
          auto& secondState = state->secondOp_.template emplace<1>(
            __implicit_convert{[state] {
              return std::execution::connect(std::move(state->secondSender_),
                                             first_receiver{state});
            }});
          std::execution::start(secondState);
        } catch (...) {
          std::execution::set_error(std::move(state->receiver_), std::current_exception());
        }
      }

      template<typename Error>
        requires std::execution::receiver<Receiver, Error>
      void set_error(Error&& error) noexcept {
        std::execution::set_error(std::move(state_->receiver_), (Error&&)error);
      }

      void set_done() noexcept {
        std::execution::set_done(std::move(state_->receiver_));
      }
    };

    explicit operation_state(First&& first, Second&& second, Receiver receiver)
      : secondSender_((Second&&)second)
      , receiver_((Receiver&&)receiver)
      , state_(std::in_place_index<0>, __implicit_convert{[this, &first] {
                 return std::execution::connect(std::move(first),
                                                first_receiver{this});
               }})
    {}

    void start() & noexcept {
      std::execution::start(std::get<0>(state_));
    }

  private:
    Second secondSender_;
    Receiver receiver_;

    // This operation-state contains storage for the child operation-states of
    // the 'first' and 'second' senders. Only one of these is active at a time
    // so we use a variant to allow the second sender to reuse storage from the
    // first sender's operation-state.
    std::variant<std::execution::connect_result_t<First, first_receiver>,
                 std::execution::connect_result_t<Second, second_receiver>> state_;
  };

public:
```

```
  explicit sequence_sender(First first, Second second)
  : firstSender_((First&&)first)
  , secondSender_((Second&&)second)
  {}

  template<typename Receiver>

  operation_state<std::remove_cvref_t<Receiver>> connect(Receiver&& r) && {
    return operation_state<std::remove_cvref_t<Receiver>>{
      std::move(first_), std::move(second_), (Receiver&&)r};
  }

private:
  First firstSender_;
  Second secondSender_;
};
```