One-Way `execute` is a Poor Basis Operation

Document #:	P1525R1
Date:	2020-10-15
Project:	Programming Language C++
Audience:	Library Evolution

Reply-to:	Gašper Ažman <gasper.azman@gmail.com> Lewis Baker <lbaker@fb.com> Lee Howes <lwh@fb.com> Corentin Jabot <corentin.jabot@gmail.com> Tomasz Kamiński <tomaszkam@gmail.com> Zach Laine <whatwasthataddress@gmail.com> Eric Niebler <eniebler@fb.com> Kirk Shoop <kirkshoop@fb.com> Ville Voutilainen <ville.voutilainen@gmail.com>

1 Abstract

The `executor` concept of [P0443R14] has a single basis operation: a `void`-returning `execute(ex, fun)` function, where `ex` is an `executor` and `fun` is a nullary `invocable`. Any errors that happen, whether during task submission, after submission and prior to execution, or during task execution, are handled in an implementation-defined manner, which can vary from executor to executor. The implication is that no *generic* code can respond to asynchronous errors in a portable way. That prevents higher-level asynchronous algorithms that require flexible error handling from being built on top of the one-way `execute()` function.

In addition, if an `executor` chooses for some reason to not execute a callback that has been submitted for execution, at present there is no mechanism – apart from destruction – for the `executor` to notify the callback that it will never be executed. For reasons described in this paper, destruction is an unsatisfactory way to communicate cancellation.

Finally, in the general case an eager one-way `execute()` operation requires the state for any asynchronous task to be dynamically allocated. In contrast, the sender/receiver design (also in [P0443R14]) permits zero-allocation scheduling, as will be demonstrated.

Since one-way `execute()` is the basis operation of the `executor` concept from [P0443R14], another way to frame the thesis of this paper is as follows:

The `scheduler` concept is a more foundational abstraction than the `executor` concept.

1.1 Revision History

1.1.1 Revision 1

This version updates the discussion points and the example code to accommodate the recent change to sender/receiver to split `execution::submit`, which used to be a basis operation, into `execution::connect` and `execution::start`.

1.2 Terms and Definitions

1.2.1 One-way execute

For the purpose of this document, by "one-way `execute`," we mean a `void`-returning function that accepts a nullary `invocable` and eagerly submits it for execution on an execution agent that the executor creates for it.

We contrast one-way `execute` with a `connect` operation that takes a `sender` and a `receiver`, paired with a `start` operation on the resulting `operation_state`.

1.2.2 Basis Operation

In generic programming, the *basis operations* of a concept are those expressions that are required to be valid for types satisfying that concept, in addition to the semantic and complexity requirements for those expressions. For example, the basis operations for C++20's `input_iterator` concept are unary `*`; prefix and postfix `++`; and `==` and `!=` with a sentinel.

The basis operations of a well-designed concept or concept hierarchy are the minimal set of operations that are both sufficient and necessary for efficiently implementing all algorithms of interest within a particular domain.

1.2.3 Sender

A sender is a general representation of a (possibly deferred, possibly async) computation. Its single basis operation, `connect`, takes a receiver and returns an instance of a non-movable type satisfying `operation_state`. The caller of `connect` assumes responsibility to keep the operation state object alive until the async operation has completed. The operation has not logically started until `start` is called on it (see below). The operation completes when one of the receiver's basis operations is called.

The receiver's functions are called from whatever execution context the sender completed in. (Senders obtained by calling a `scheduler`'s `schedule` operation place extra requirements on that execution context; see below.)

1.2.4 Operation state

The operation state is literally the state associated with an async operation. It typically keeps the receiver alive as a data member. The `operation_state` concept has a single basis operation called `start()` that takes the operation state as an argument and ensures the operation is started, possibly by enqueuing it for execution on some execution context. The caller of `start()` is required to keep the execution state alive for the duration of the async operation. Once one of the receiver functions has been called, the operation state can be assumed to be destroyed.

1.2.5 Receiver

A receiver is a general representation of a callback. It has three basis operations:

— *`set_value`*, which is invoked with the result(s) of the sender's computation, if any, when that operation completes.
— *`set_error`*, which is invoked with any errors from enqueueing the work, or if the task itself completes with an error.
— *`set_done`*, which is invoked on a receiver by a sender when the sender's computation has been cancelled.

The execution context in which these operations are invoked is specific to each sender. In general, they will execute inline; that is, in whatever context the sender completed on.

1.2.6 Scheduler

A *scheduler* is a factory for senders that complete in the scheduler's execution context. For a `scheduler` *sched*, a receiver *rec*, and an operation state *op* initialized with `connect(schedule(`*sched*`), `*rec*`)`, then `start(`*op*`)` will enqueue for execution a work unit that is guaranteed to call `set_value(move(`*rec*`))` in the execution context associated with *sched* – if that is at all possible. If not, it will call `set_error` on *rec* with the reason for the failure in an unspecified execution context.

2 No reliable error propagation

2.1 Errors cannot be intercepted

Consider the following strategies that a tasking system might employ to respond to scheduling or execution errors:

1. On error, ignore the error and propagate a default value instead.
2. On error, cancel some dependent execution.
3. On error, send the error information to a particular error log before propagating the error.
4. On error, re-schedule the task on a fallback execution context.

All of these are reasonable responses to scheduling and execution errors in a tasking system, and all can be built using generic asynchronous algorithms, but only if the executor passes scheduling and invocation errors to those callbacks that desire it.

For instance, Appendix B shows how a failure to execute work on one scheduler can reschedule the work on a fallback scheduler. Such a generic fallback scheduler cannot be built if `execute()` is taken as the basis operation for asynchrony.

In that example, note that the separation of the value and error channels gives us the ability to place independent constraints on the execution contexts of the two. The sender returned from `schedule` *requires* that the value channel completes in the scheduler's execution context, which is how we achieve predictable scheduling. But if there is an error, the context on which the receiver's `set_error()` channel is run is *unspecified*. We have an easy way to guarantee that there is always an execution context available to process errors – inline, for instance – while also guaranteeing that task submission is non-blocking when we need that guarantee.

2.2 Errors that happen after submission but before invocation have no place in-band to go

Although an executor may choose to report submission errors to the caller with an exception, that is not an option for errors that happen *after* submission but before execution. Consider that there is no requirement that an executor create an execution agent eagerly when `execute` is called; it may defer the creation until a later time, at which point it may fail. Also consider the case of deadline executor that un-stages work that hasn't been started before a certain time-out. If one-way `execute` is the basis operation, then once work has been submitted there is no way to communicate to the work that an error happened because there is no defined channel for errors.

With `schedule` and sender/receiver, such a deadline executor can pass a `error_timeout_exceeded` error to the receiver's error channel.

2.2.1 Corollary: One-way `execute` can be implemented in terms of `schedule` but not vice versa

As a corollary of the preceding points, we cannot implement `schedule` with perfect fidelity in terms of one-way `execute`. For executors, it is unspecified what happens when `execute` fails to enqueue the function for execution. As a consequence, there is no way to pass those errors to a receiver's error channel in a generic algorithm.

In contrast, `execute` *can* be implemented in terms of `schedule`/`connect`/`start`. See Appendix A for an example implementation.

[P0443R14] permits executors to be transparently treated as schedulers via an imperfect adaptation baked into the design of the customization points. Reviewers have rightly flagged this as suspect. In addition to the problem noted above where an executor's scheduling errors are not reported in the error channel, another thorny issue is discussed in executors#463, which details how the `execution::connect` customization point tries and fails to accommodate users who pass an executor instead of a scheduler. Consider the specification of `connect`'s behavior when passed an executor *e* and a receiver *r*. It is equivalent to *as-operation*`{e, r}` where *as-operation* is:

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

and *as-invocable* is:

```
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(std::move(*r_));
  }
  void operator()() & noexcept try {
    execution::set_value(std::move(*r_));
    r_ = nullptr;
  } catch(...) {
    execution::set_error(std::move(*r_), current_exception());
    r_ = nullptr;
  }
};
```

The problem happens when the call to `execute` in *as-operation*`::start()` throws. In that case, the instance of *as-invocable* will be destroyed, presumably without having been invoked, and its destructor calls `set_done`. However, the `catch(...)` in *as-operation*`::start()` will catch the exception and try to route it to the receiver's `set_error`. Both `set_done` and `set_error` are terminal. Calling them both violates the receiver contract.

The problem is fundamental. There are two possible resolutions to the problem. The resolution described in executors#463 is to simply swallow the exception an accept that the error has been erroneously mapped into a cancellation notification. The other is to introduce an extra allocation and do the necessary synchronization so that the thread calling `execute` can know before it returns whether a call to `set_error` is appropriate. Neither of these solutions is appealing, and neither should just happen silently behind the user's back.

2.3 There is no way to compile the normal code differently than the exceptional code

Should we decide to address the above issues by extending one-way `execute` to require users to pass an invocable that accepts a `std::error_code` in addition to a value (for example), we run into a different problem: the same function is now used for both normal and exceptional execution. If the execution context is an NVIDIA GPU,

that means that both the normal function execution as well as error handling must be compiled for the GPU.

With `schedule` and sender/receiver, `set_value` and `set_error` are separate channels, and they can be compiled differently. `set_value` can be compiled for and execute on the GPU, whereas `set_error` can be compiled for and execute on the host. This also has the advantage that a bulk algorithm can have a scalar `set_error` handler, something that is much harder to craft in the `bulk_execute` design of [P0443R14].

3 No reliable propagation of a cancellation signal

This section describes the problems with one-way `execute` as a basis operation that stem from its lack of support for a "done" signal to propagate cancellation information. First we discuss what "done" means for async computations, and why it is separate from destruction.

3.1 What does `set_done()` mean?

The reason for a receiver's "error" channel is pretty straightforward; it is the same reason C++ has exceptions: it is greatly advantageous to isolate the exceptional control flow from the normal control flow.

The reasons for a receiver's "done" channel are less obvious, but it comes down to cancellation. In the presence of cancellation, all async operations look like functions that return `std::optional`: they either complete successfully with a result, they exit via an exception, or else they return with neither a result nor an exception. These options correspond to the receiver's three channels: `set_value`, `set_error`, and `set_done`.

In functional programming circles, `optional` is represented as the Maybe monad, which has two constructors: `Just` and `None`, which correspond to an optional with a value and `nullopt`. Composing operations in the Maybe monad uses short-circuiting: if the preceding computation results in None, the subsequent computations are not even tried; the result is simply None.

The same is true of composing asynchronous computations. If a preceding computation is cancelled, dependent computations should likewise be canceled, bypassing the normal control flow. Think of it as exception unwind, but without the exception. That is the meaning of `set_done`.

3.2 Why is the `set_error()` channel a bad way to report cancellation?

The `set_error()` channel of a receiver, like C++ exceptions, is for exceptional circumstances: things like dropped network connections, resource allocation failure, or inability to create an execution agents. Cancellation is not exceptional; it is the normal operating mode for many interesting async algorithms. For instance, a `when_any()` algorithm would take many tasks, enqueue them all for execution, and then cancel the rest when the first completes. The exceptional code path should not have to deal with normal control flow. Cancellation requests is something distinct from value propagation or error propagation. That is why we believe they deserve their own distinct channel.

See [P1677R2] "Cancellation is serendipitous-success" for a full discussion of cancellation in relation to asynchronous errors.

3.3 Why is callback destruction-without-execution insufficient for communicating cancellation?

Even if one accepts that async cancellation is fundamental, and that it is still not an error, it is not obvious that we need a dedicated channel to communicate cancellation. After all, isn't it sufficient to simply destroy a continuation without executing it?

There are lots of reasons why the destructors of a continuation might get called:

1. It is being destroyed after `set_value()` has been called on it.
2. It is being destroyed after `set_error()` has been called on it.
3. It is a moved-from object that is being cleaned up.

4. It has been cancelled.

Only for reason (4) should a destructor call be interpreted as the "done" signal. In order to distinguish (4) from the other three cases, a continuation would need to keep state.

Also, it is not clear what it would mean to destroy a continuation due to stack unwinding because of an active exception. Presumably, that would be an error situation and not a cancellation, but clearly if the destructor is being called, `set_error()` never will be. Would that be a logic error? Or should it be simply ignored, which would require the continuation to keep additional state and two calls to `std::unhandled_exceptions()` (see the design of `scope_success` [P0052R10])?

In contrast, we hypothesize that most executors will know when a particular work item is being cancelled and can propagate the "done" signal without tracking extra state. Executors generally un-stage work items to execute them and then either immediately destroy them or move them to a separate queue for lazy reclamation. The executor knows that any work items that are currently staged for execution have not yet been run (that is, `set_value()` has not been called), and that scheduling has not failed nor has invocation failed (that is, `set_error()` has not been called). So, if the executor supports work cancellation, any request to cancel one or all of the currently staged work items can trivially call `set_done()` on the them before un-staging and destroying them. Such executors – which we imagine to be the vast majority – can trivially insert a call to `set_done()` without tracking any additional state.

Any executor that does *not* support work cancellation can safely ignore the `set_done()` channel. No cancellation means no need to ever send a cancellation signal.

3.4 Example: Adding a "done" channel to ASIO's `scheduler`

The `scheduler` class in [ASIO] permits the scheduler to be shut down while there are still outstanding work items in its queue. These are simply destroyed at present. We believe that supporting the `set_done()` channel in ASIO would be as simple as inserting a call to `set_done()` on line 165 of `<asio/detail/impl/scheduler.ipp>` before the call to `o->destroy()`.

This demonstrates why we believe that adding support for the "done" channel to an executor is not an onerous requirement.

4 Additional considerations

4.1 One-way execute cannot guarantee no-allocation scheduling

A scheduler can be implemented such that it provides a guaranteed no-allocation scheduling operation. This is possible because the `connect` operation that joins a sender to a receiver returns the operation state to the caller. The caller can then place that state anywhere, even on the stack if it knows that the async operation will complete before the function returns.

One-way `execute` cannot easily take advantage of the ability to do no-allocation scheduling because it launches work eagerly and does not return the operation state to the caller.

An executor could get part way there if it reserved a certain amount of space for a fixed number of tasks, and if the caller were careful not to try to execute any functions that didn't fit in the space reserved. In the general case, however, if the user enqueues too many tasks, or the tasks themselves are too large, `execute()` must fall back to dynamically allocating space or else fail to schedule the work.

The fundamentally eager, fire-and-forget semantics of `execute()` mean that it is impossible in the general case to write an executor that can guarantee no allocations. This has unfortunate consequences when used together with coroutines. When using coroutines, there is typically an allocation for the coroutine frame (although some allocations are elided by compiler optimizations). If `execute()` is selected as the basis operation, then posting work to an executor from a coroutine requires *another* allocation. With schedulers, however, the problem is avoided. We can design our coroutine types such that the operation state returned by `connect()` is *guaranteed*

to be stored inline in the coroutine frame. No extra allocations are necessary from coroutines to transition to a separate execution context.

Please see Appendix C for an example of a scheduler that permits allocation-free scheduling.

4.2 Concerns about scheduler complexity are likely misplaced

The rest of this document argues that there are strong technical arguments to prefer `scheduler` over `executor`. This section explains why the extra syntactic and semantic requirements of the `scheduler` concept are not overly burdensome.

There can be no doubt that the `scheduler` concept places a higher burden on execution context providers than `executor`. After all, it's very simple to write a single `execute()` function, and many execution contexts provide an interface much like that already. So `executor` is better than `scheduler`, right?

Not really. In truth, execution contexts like thread pools are often complicated beasts. The extra syntactic burden of providing a scheduler for them is small in comparison, and are outweighed by the benefits to generic code of the additional functionality. And as standard concepts go, `scheduler` is middling in complexity. Far more complex are concepts like `bidirectional_iterator` and `random_access_iterator` in terms of sheer number of syntactic requirements.

Some executors, however, are very simple, and for those the extra syntax needed to satisfy the `scheduler` concept can be a very large fraction of the total. One such often- cited example is the `inline_executor`, which runs tasks immediately in the context of the caller, blocking the calling thread until the task completes. Its definition is given roughly as:

```
struct inline_executor {
  template <class Fun>
  void execute(Fun fun) const {
    fun();
  }
  bool operator==(const inline_executor&) const = default;
};
```

(Here we gloss over details such as properly responding to property queries such as for blocking behavior.)

In contrast, here is the implementation of the `inline_scheduler`:

```
struct inline_scheduler {
  auto schedule() const {
    return execution::just();
  }
  bool operator==(const inline_scheduler&) const = default;
};
```

This makes use of the `just` generic algorithm from [P1897R3]. Think of `just()` as a way to make a "ready" sender: `just(args...)` is a sender that, when `connect`-ed to a receiver and `start`-ed, passes `args...` to the receiver's `set_value` channel immediately on the same thread on which `start` was called.

The `inline_scheduler` could be used as follows:

```
auto op = execution::connect(inline_scheduler{}, some_receiver_of_void);
execution::start(op); // guaranteed to complete inline on the caller's thread
```

or, given the implementation of the `execute()` algorithm from Appendix A, as:

```
// A complicated way to print "hello world" on the current thread.
execution::execute(inline_sheduler{}, []{ printf("hello world\n"); });
```

The implementation of `just()` is not terribly complex but admittedly more complex than the `inline_executor`. It could be argued that this definition of `inline_scheduler` is merely moving the complexity around. Is it cheating to implement `inline_scheduler` in terms of `just()`? Hardly. The whole point of the sender/receiver abstraction is to enable a suite of composable, generic algorithms so that more complicated and interesting async patterns can be composed out of smaller, simpler pieces like `just()`. This implementation of the `inline_scheduler` is an illustration of how sender/receiver makes it possible to cleanly decompose asynchrony into small orthogonal components that combine in interesting, and sometimes pleasantly surprising, ways.

Users of the executor abstraction frequently have need to write simple, shim executor adaptors in order to customize the behavior of an executor. That is necessary because the design of the executor concept doesn't lend itself well to a suite of generic algorithms that capture interesting async patterns. With schedulers, there is less need to write scheduler adaptors because the algorithms are generally expressed in terms of senders and adaptors over senders.

This paper does not try to argue that all schedulers will be as simple at their executor analogues. It merely argues that the extra complexity is not onerous and is well- compensated by the additional functionality.

5 Appendix A: One-way `execute` as a generic algorithm

Below is the implementation of customizable one-way `execute` algorithm in terms of schedulers, senders, and receivers (using `tag_invoke` from [P1895R0] to find program-defined specializations).

Working code for this example can be found here: https://godbolt.org/z/TYa1dh.

```
inline constexpr struct __execute_cpo {
private:
  template<std::invocable Func>
  struct as_receiver {
    Func func_;
    as_receiver(Func func) : func_((Func&&) func) {}

    void set_value() && noexcept(std::is_nothrow_invocable_v<Func>) {
      static_cast<Func&&>(func_)();
    }
    [[noreturn]] void set_error(auto&&) && noexcept { std::terminate(); }
    void set_done() && noexcept { }
  };

  template<execution::scheduler Sched, std::invocable Func>
  friend void tag_invoke(__execute_cpo, Sched sched, Func func) {
    execution::submit(
      execution::schedule((Sched&&) sched), as_receiver{(Func&&)func});
  }

public:
  template<execution::scheduler Sched, std::invocable Func>
  void operator()(Sched sched, Func func) const
      noexcept(is_nothrow_tag_invocable_v<__execute_cpo, Sched, Func>) {
    static_assert(
      std::is_void_v<tag_invoke_result_t<__execute_cpo, Sched, Func>>);
    tag_invoke(*this, (Sched&&)sched, (Func&&)func);
  }
} execute{};
```

[*Note:* The above implementation is expressed in terms of the generic `submit` algorithm from [P0443R14]. You can find a fuller example that expresses this algorithm strictly in terms of fundamental basis operations

here: https://godbolt.org/z/f71TKq — *end note*]

A real `execute()` implementation would take care of a few extra details. First, if the scheduler reports that it is "blocking" (by way of the blocking property query), then the dynamic allocation in `submit` is unnecessary. The operation state can live on the stack because we know the operation will complete before the operation state goes out of scope.

Also, it should be possible to specify an allocator to use when the scheduler is non-blocking. That would be handled by associating an allocator with the invocable via a `get_allocator` property query.

In P0443's `executor` concept, `execute()` generally allows exceptions thrown from the function object to propagate back out to the executor's event loop, where some generic exception-handling code can log the error, handle it and resume the event loop or maybe terminate. This default implementation of `execute()` does not have this behavior. Having said that, as the behavior of an exception thrown from the function-callback is implementation-defined then a default implementation that terminates, as this one does, is valid.

Custom implementations of `execute()` are still free to do what they previously did.

Also, a particular implementation of a scheduler is free to wrap all calls to `set_value` in a `try/catch` and route any exceptions back to the execution context. Then the scheduler would call `set_done` on the receiver to satisfy the receiver contract. That would make the default implementation of `execute()` above behave as `execute()` functions typically do today in [ASIO].

6 Appendix B: Composing Schedulers based on Sender/Receiver

With a sender/receiver-based `schedule` operation as a basis operation we can build composed schedulers with interesting behaviors.

Below is an example `fallback_scheduler` that composes two schedulers. When scheduling work on a `fallback_scheculer`, it first tries to schedule work on the primary scheduler. If that fails, it tries to schedule the work on the fallback scheduler. It is impossible to build such a generic `fallback_executor` using the one-way `execute()` function as a basis operation because there is no generic way to detect scheduling failures, and thus no way to take a fallback action.

[*Note:* Note that because the `connect` operation returns the operation state to the caller, it is possible here for the primary state and the fallback state to share space because their lifetimes never overlap. They are stored below in a `union`. — *end note*]

A working example of this code can be found here: https://godbolt.org/z/P46nhW.

```
enum class which { none, primary, fallback };

template<typename T>
void _destroy(T& t) noexcept {
  t.~T();
}

template<execution::sender S1, execution::sender S2>
struct fallback_sender : execution::sender_base {
  S1 primary_;
  S2 fallback_;

  template<execution::receiver R>
  struct operation_state {
    struct primary_receiver {
      operation_state* state_;

      template<typename... Values>
```

```
      requires execution::receiver_of<R, Values...>
    void set_value(Values&&... values) && {
      execution::set_value((R&&) state_->receiver_, (Values&&)values...);
    }
    void set_done() && noexcept {
      execution::set_done((R&&) state_->receiver_);
    }
    void set_error(auto&&) && noexcept try {
      _destroy(state_->primary_state_);
      state_->which_ = which::none;
      ::new(&state_->fallback_state_)
        auto(execution::connect((S2&&) state_->fallback_, secondary_receiver{state_}));
      state_->which_ = which::fallback;
      execution::start(state_->fallback_state_);
    } catch(...) {
      execution::set_error((R&&) state_->receiver_, std::current_exception());
    }
  };
  struct secondary_receiver {
    operation_state* state_;

    template<typename... Values>
      requires execution::receiver_of<R, Values...>
    void set_value(Values&&... values) && {
      execution::set_value((R&&) state_->receiver_, (Values&&)values...);
    }
    void set_done() && noexcept {
      execution::set_done((R&&) state_->receiver_);
    }
    template<typename Error>
      requires execution::receiver<R, Error>
    void set_error(Error&& error) && noexcept {
      execution::set_error((R&&)state_->receiver_, (Error&&)error);
    }
  };

  S2 fallback_;
  R receiver_;
  union {
    execution::connect_result_t<S1, primary_receiver> primary_state_;
    execution::connect_result_t<S2, secondary_receiver> fallback_state_;
  };
  which which_;

  operation_state(S1&& primary, S2&& fallback, R&& receiver)
    : fallback_((S2&&) fallback)
    , receiver_((R&&) receiver)
    , primary_state_(
        execution::connect((S1&&)primary, primary_receiver{this}))
    , which_(which::primary)
  {}
  ~operation_state() {
    switch(which_) {
    case which::primary:
```

```
        _destroy(primary_state_);
        break;
      case which::fallback:
        _destroy(fallback_state_);
        break;
      default:;
      }
    }

    void start() & noexcept {
      execution::start(primary_state_);
    }
  };

  template<execution::receiver R>
  operation_state<R> connect(R receiver) && {
    return {(S1&&) primary_, (S2&&) fallback_, (R&&) receiver};
  }
};

template<execution::sender S1, execution::sender S2>
fallback_sender<S1, S2> fallback(S1 primary, S2 fallback) {
  return {{}, std::move(primary), std::move(fallback)};
}

template<execution::scheduler Sched1, execution::scheduler Sched2>
struct fallback_scheduler {
  Sched1 primary_;
  Sched2 fallback_;

  auto schedule() {
    return fallback(execution::schedule(primary_), execution::schedule(fallback_));
  }
};
template<class Sched1, class Sched2>
fallback_scheduler(Sched1, Sched2) -> fallback_scheduler<Sched1, Sched2>;
```

7 Appendix C: Allocation-free scheduling

Below is an example scheduler that does not require heap-allocation of the queue items. Full working code can be found at https://godbolt.org/z/hcxhG9.

```
class thread_dispatcher {
  struct queue_item {
    queue_item* next_ = nullptr;
    virtual void execute(int id) noexcept = 0;
  };

  queue_item* head_ = nullptr;
  bool stopRequested_ = false;
  std::mutex mut_;
  std::condition_variable cv_;
  std::thread thread1_;
```

```
  std::thread thread2_;

  class scheduler {
    thread_dispatcher& dispatcher_;
    class schedule_sender : public execution::sender_base {
      thread_dispatcher& dispatcher_;

    public:
      explicit schedule_sender(thread_dispatcher& dispatcher) noexcept
        : dispatcher_(dispatcher)
      {}

      template<typename Receiver>
      auto connect(Receiver r) {
        class operation_state final : public queue_item {
          thread_dispatcher& dispatcher_;
          Receiver receiver_;

          void execute(int id) noexcept override try {
            execution::set_value((Receiver&&) receiver_, id);
          } catch(...) {
            execution::set_error((Receiver&&) receiver_, std::current_exception());
          }
        public:
          explicit operation_state(thread_dispatcher& d, Receiver&& r)
            : dispatcher_(d), receiver_((Receiver&&) r)
          {}

          void start() & noexcept {
            dispatcher_.enqueue(this);
          }
        };
        return operation_state{dispatcher_, (Receiver&&) r};
      }
    };
  public:
    explicit scheduler(thread_dispatcher& dispatcher) noexcept
      : dispatcher_(dispatcher)
    {}

    schedule_sender schedule() noexcept {
      return schedule_sender{dispatcher_};
    }
    bool operator==(const scheduler&) const = default;
  };

public:
  thread_dispatcher()
    : thread1_([this] { this->run(1); })
    , thread2_([this] { this->run(2); })
  {}

  ~thread_dispatcher() {
    request_stop();
```

```
    thread1_.join();
    thread2_.join();
  }

  scheduler get_scheduler() { return scheduler{*this}; }

private:
  void request_stop() noexcept {
    std::lock_guard lock{mut_};
    stopRequested_ = true;
    cv_.notify_all();
  }

  void run(int id) {
    std::unique_lock lock{mut_};
    while (!stopRequested_) {
      if (head_ == nullptr) {
        cv_.wait(lock);
      }
      while (head_ != nullptr) {
        auto* item = head_;
        head_ = item->next_;
        lock.unlock();
        item->execute(id);
        lock.lock();
      }
    }
  }

  void enqueue(queue_item* item) noexcept {
    std::lock_guard lock{mut_};
    item->next_ = head_;
    head_ = item;
    cv_.notify_one();
  }
};
```

8 Appendix D: Implementation of a two-way `to_future` algorithm

This example demonstrates how sender/receiver can be used to implement a two-way `to_future()` algorithm that eagerly submits a sender for execution and returns a `std::future` that receives the result of the computation.

A working example of this code can be found here: https://godbolt.org/z/Ms1Ysd.

```
template <class...> struct _one_or_none;
template <> struct _one_or_none<> {
  using type = void;
};
template <class T> struct _one_or_none<T> {
  using type = T;
};

template <class...> struct _just_one;
template <class T> struct _just_one<T> {
```

```
  using type = T;
};

template <execution::typed_sender Sender>
using _sender_value_t =
  typename execution::sender_traits<remove_cvref_t<Sender>>::
    template value_types<_just_one, _one_or_none>::type::type;

template <execution::typed_sender Sender>
using _sender_error_t =
  typename execution::sender_traits<remove_cvref_t<Sender>>::
    template error_types<_just_one>::type;

template <class S>
concept _single_typed_sender =
  execution::typed_sender<S> &&
  requires {
    typename _sender_value_t<S>;
    typename _sender_error_t<S>;
  };

struct operation_cancelled {
  static char const* what() noexcept { return "operation cancelled"; }
};

template <class Sender>
struct _op {
  struct _rec {
    void set_value(auto&&... vals) {
      op_->promise_.set_value((decltype(vals)) vals...);
      delete op_;
    }
    void set_error(exception_ptr err) noexcept {
      op_->promise_.set_exception(err);
      delete op_;
    }
    void set_done() noexcept {
      op_->promise_.set_exception(
        make_exception_ptr(operation_cancelled{}));
      delete op_;
    }
    _op* op_;
  };
  explicit _op(Sender&& sender)
    : state_(execution::connect((Sender&&) sender, _rec{this}))
    , promise_{}
  {}
  void start() & noexcept {
    execution::start(state_);
  }
  execution::connect_result_t<Sender, _rec> state_;
  promise<_sender_value_t<Sender>> promise_;
};
```

```
template< _single_typed_sender Sender >
  requires std::same_as<_sender_error_t<Sender>, exception_ptr>
future<_sender_value_t<Sender>> as_future( Sender&& sender ) {
  auto* op = new _op<Sender>{(Sender&&) sender};
  auto fut = op->promise_.get_future();
  op->start();
  return fut;
}
```

There are some things to note about the above implementation of `to_future()`. First and foremost, it does an extra allocation – one in `to_future()` itself and the other in the constructor of `std::promise` to construct the shared state – so this particular implementation is not zero-overhead. The extra allocation can be eliminated by passing a custom allocator to the `std::promise` constructor that over-allocates and puts the operation state (`_op<Sender>::state_` above) into the unused portion of the allocation. Though not difficult, it would obscure the algorithm, so it's not presented here.

The other notable thing about this example is the handling of cancellation. This implementation of `to_future()` causes an instance of `operation_cancelled` to be thrown from `future::get()` when the sender completes by calling `set_done()`. This conflates cancellation with error handling. Arguably a better approach would be to report cancellation by having `future::get()` return an `optional`, using `nullopt` to indicate that the computation was cancelled.

9 References

[ASIO] Asio C++ Library.
https://github.com/chriskohlhoff/asio

[P0443R14] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, H. Carter Edwards, Gordon Brown, D. S. Hollman. 2020. A Unified Executors Proposal for C++.
https://wg21.link/p0443r14

[P1677R2] Kirk Shoop, Lisa Lippincott, Lewis Baker. 2019. Cancellation is not an Error.
https://wg21.link/p1677r2

[P1895R0] Lewis Baker, Eric Niebler, Kirk Shoop. 2019. tag_invoke: A general pattern for supporting customisable functions.
https://wg21.link/p1895r0

[P1897R3] Lee Howes. 2020. Towards C++23 executors: A proposal for an initial set of algorithms.
https://wg21.link/p1897r3