One-Way `execute` is a Poor Basis Operation

Document #:	D1525R0
Date:	2019-06-17
Project:	Programming Language C++ SG1 LEWG

Reply-to:	Eric Niebler <eniebler@fb.com> Kirk Shoop <kirkshoop@fb.com> Lewis Baker <lbaker@fb.com> Lee Howes <lwh@fb.com>

1 Abstract

The `OneWayExecutor` concept of [P0443R10] has a single basis operation: a `void`-returning `ex.execute(fun)` member function, where `ex` is a `OneWayExecutor` 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 control structures 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.

Paper [P1660R0] will discuss how most of the below issues can be addressed with a modified design for one-way `execute`.

1.1 Terms and Definitions

1.1.1 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 `submit` operation that takes a Sender and a Receiver, as described in [P1341R0].

1.1.2 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 required semantics for those expressions. For example, the basis operations for C++20's `InputIterator` are unary `*`; prefix and postfix `++`; and `==` and `!=` with a sentinel (and associated semantic requirements for each operation).

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

1.1.3 Sender

A Sender is a general representation of a (possibly deferred, possibly async) computation. Its single basis operation, `submit`, takes a Receiver and returns `void`. Like `execute`, `submit` eagerly submits a task for execution. The Receiver's functions are called from whatever execution context the Sender completed in. (Senders returned from a `Scheduler`'s `schedule` operation place extra requirements on that execution context; see below.)

1.1.4 Receiver

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

— *`value`*, which is invoked with the result of the Sender's computation, if any, when that operation completes.
— *`error`*, which is invoked with any errors from enqueueing the work, or if the task itself completes with an error.
— *`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.

1.1.5 Scheduler

A *`Scheduler`* (formerly an "*`Executor`*" in [P1341R0]) is a factory for Senders that complete in the execution context of the `Scheduler`. For a `Scheduler sched` and a Receiver `rec`, `submit(schedule(sched), rec)` is guaranteed to call `value` on `rec` in the execution context of `sched` if that is at all possible. If not, it will call `error` on `rec` with the reason for the failure in an unspecified execution context.

Note: In the current design, the Senders returned by `schedule` will pass a sub-executor to the Receiver's `value` member to facilitate the construction of nested work within the same 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 higher-level control structures, 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 executor can reschedule the work on a fallback executor.

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 `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 One-way `execute` cannot be `noexcept` in general

In [P0443R10]'s one-way `execute` function, it is specified that the submitted function is required to be decayed in the caller's thread of execution; presumably any exceptions from that operation are propagated back to the caller on their thread. If some other kind of error happens – say, the Executor fails to create an execution agent – it is unspecified how that error is reported. It is reasonable then to assume that for some Executors, these errors are also reported to the caller on their thread via an exception. The implication is that generic code using an Executor must be ready for a call to `execute` to throw.

With `schedule` and Sender/Receiver, any errors in `submit` are sent to the Receiver's error channel. `submit` can be a guaranteed no-throw operation.

2.3 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. Firstly, 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.

2.3.1 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` generally in terms of one-way `execute`. In P0443, for one-way 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 contrast, `execute` can be implemented in terms of `schedule/submit`. See Appendix A for an example implementation.

2.4 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 requires users to pass an Invocable that accepts a `std::error_code` (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, `value` and `error` are separate channels, and they can be compiled differently. `value` can be compiled for and execute on the GPU, whereas `error` can be compiled for and execute on the host. This also has the advantage that a bulk algorithm can have a scalar `error` handler, something that is much harder to craft in the `bulk_execute` design of [P0443R10].

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 `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: `value`, `error`, and `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 `done()`.

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

The `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 [P1677R0] "Cancellation is not an Error" 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 the destructors of a continuation might get called:

1. It is being destroyed after `.value()` has been called on it.
2. It is being destroyed after `.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 four cases, a continuation would need to keep state.

Also, it's 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, `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, `value()` has not been called), and that scheduling has not failed nor has invocation failed (that is, `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 `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 `done()` without tracking any additional state.

Any executor that does *not* support work cancellation can safely ignore the `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 `done()` channel in ASIO would be as simple as inserting a call to `done()` on line 165 of `<asio/detail/impl/scheduler.ipp>` before the call to `o->destroy()`.

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

4 Additional problems with one-way `execute`

4.1 When using coroutines, one-way execute cannot take advantage of no-allocation scheduling

One-way `execute` cannot take advantage of the ability to do no-allocation scheduling for coroutines (and potentially in the future for `connect`/`start`-based `submit`).

By having the `schedule()` method return a Sender that can be used to lazily start the operation, we can have this Sender define a custom `operator co_await()` method that returns an `Awaiter` object that contains storage for the queue item needed to track the schedule operation.

When a coroutine executes `co_await ex.schedule()`, this allows the Sender to allocate storage required for the queue entry as a local variable on the coroutine frame rather than the executor having to allocate the storage for the continuation/Receiver internally to `execute()`, in the case of one-way `execute`, or `submit()` in the case of Sender/Receiver.

For an example of an executor that does not require heap-allocating any storage when `co_await`ed within a coroutine see Appendix C.

It is not possible to build this kind of non-allocating executor-schedule operation if one-way `execute()` is the basis operation.

A similar approach can be achieved with Sender/Receiver if we split the `submit()` operation up into two separate operations: a `connect()` operation that accepts the Sender and Receiver and returns a state-machine object, and a `start()` method on the state-machine object that launches the async operation. Once the operation is started, the caller of `connect()` is required to keep the state-machine object alive until the operation completes.

This allows the caller to place the state-machine for the async operation inline within the coroutine frame or as a member of some other object without forcing the state-machine to be heap-allocated (although the caller can still heap-allocate the state-machine if they choose to).

> [*Note:* Although the other problems with `execute` described in this paper can be addressed with the designed sketched by [P1660R0], the inefficiency when used with coroutines is *not* addressed by that paper. — *end note*]

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

Implementation of one-way `execute` in terms of `schedule()`, `submit()` and `pinvoke()`

```
inline constexpr struct execute_cpo {
private:
  template<Invocable Func>
  struct receiver {
    Func func_;

    template<Executor SubEx>
    void value(SubEx&&) noexcept(std::is_nothrow_invocable_v<Func>) {
      static_cast<Func&&>(func_)();
    }

    template<typename E>
    [[noreturn]] void error(E&&) noexcept { std::terminate(); }

    [[noreturn]] void done() noexcept { std::terminate(); }
  };

  template<Executor Ex, Invocable F>
  friend void pinvoke(execute_cpo, Ex&& ex, F&& func) {
    submit(
      schedule((E&&)ex),
      receiver<std::remove_cvref_t<F>>{(F&&)func});
  }

public:
  template<Executor Ex, Invocable Func>
    requires std::is_void_v<pinvoke_result_t<execute_cpo, Ex, Func>>
  void operator()(Ex&& ex, Func&& func) const
    noexcept(is_nothrow_pinvocable_v<execute_cpo, Ex, Func>) {
    pinvoke(*this, (Ex&&)ex, (Func&&)func);
  }
} execute{};
```

6 Appendix B: Composing Executors based on Sender/Receiver

With a sender/receiver-based schedule operation as a basis operation we can more easily build composed executors.

```
template<Sender S1, Sender S2>
struct fallback_sender {
  S1 primary;
  S2 fallback;

  template<Receiver R>
  struct wrapped_receiver {
    S2 fallback;
    R receiver;

    template<typename... Values>
    void value(Values&&.. values) {
      set_value(receiver, (Values&&)values...);
    }

    void done() {
      set_done(receiver);
    }

    template<typename Error>
    void error(Error&&) {
      submit(std::move(fallback), std::move(receiver));
    }
  };

  template<Receiver R>
  void submit(R receiver) && {
    submit(std::move(primary), wrapped_receiver<R>{std::move(receiver)});
  }
};

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

template<Executor Ex1, Executor Ex2>
struct fallback_executor {
  Ex1 primary;
  Ex2 fallback;

  auto schedule() {
    return fallback(primary.schedule(), fallback.schedule());
  }
};

void example() {
  thread_pool tp;
  manual_executor fallback;

  fallback_executor ex{tp.get_executor(), fallback};
  std::for_each(std::par.on(ex), range, [](auto& x) {
    process(x);
  });
}
```

7 Appendix C: Allocation-free scheduling from a coroutine

Example executor that does not require heap-allocation of the queue items when awaited from a coroutine. https://wandbox.org/permlink/nxjpIdlqyz7DUXbm https://godbolt.org/z/NJb60u

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

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

  class executor {
    thread_dispatcher& dispatcher_;
    class schedule_sender {
      thread_dispatcher& dispatcher_;

      class awaiter final : private queue_item {
        thread_dispatcher& dispatcher_;
        std::experimental::coroutine_handle<> continuation_;

        void execute() noexcept final {
          continuation_.resume();
        }

      public:
        awaiter(thread_dispatcher& dispatcher) noexcept
        : dispatcher_(dispatcher)
        {}

        bool await_ready() noexcept { return false; }
        void await_suspend(
            std::experimental::coroutine_handle<> continuation) noexcept {
          continuation_ = continuation;
          dispatcher_.enqueue(this);
        }
        void await_resume() noexcept {}
      };

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

      awaiter operator co_await() noexcept {
        return awaiter{dispatcher_};
      }

      template<typename Receiver>
      void submit(Receiver r) noexcept {
        struct receiver_queue_item final : queue_item {
          thread_dispatcher& dispatcher_;
          Receiver receiver_;

          explicit receiver_queue_item(thread_dispatcher& d, Receiver&& r)
          : dispatcher_(d)
          , receiver_((Receiver&&)r)
          {}

          void execute() noexcept final {
            set_value(receiver_, executor{dispatcher_});
            set_done(receiver_);
            delete this;
          }
        };
        try {
          dispatcher_.enqueue(
            new receiver_queue_item{dispatcher_, (Receiver&&)r});
        } catch (...) {
          set_error(r, std::current_exception());
        }
      }
    };
  public:
    executor(thread_dispatcher& dispatcher) noexcept
    : dispatcher_(dispatcher)
    {}

    schedule_sender schedule() noexcept {
      return schedule_sender{dispatcher_};
    }
  };

public:
  thread_dispatcher()
  : thread_([this] { this->run(); }) {}

  ~thread_dispatcher() {
    request_stop();
    thread_.join();
  }
  executor get_executor() { return executor{*this}; }

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

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

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

8 References

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

[P0443R10] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, H. Carter Edwards, Gordon Brown, David Hollman. 2019. A Unified Executors Proposal for C++.
https://wg21.link/p0443r10

[P1341R0] Lewis Baker. 2018. Unifying Asynchronous APIs in the Standard Library.
https://wg21.link/p1341r0

[P1660R0] Jared Hoberock, Michael Garland, Bryce Adelstein Lelbach, Michał Dominiak, Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes, David S. Hollman, and Gordon Brown. 2019. P1660R0: A Compromise Executor Design Sketch.
http://wg21.link/P1660R0

[P1677R0] Kirk Shoop. 2019. P1677R0: Cancellation is not an Error.
http://wg21.link/P1677R0