A Compromise Executor Design Sketch

Document #:	D1660R0
Date:	2019-06-03
Project:	Programming Language C++ SG1 LEWG

Reply-to:	Jared Hoberock <jhoberock@nvidia.com> Michael Garland <mgarland@nvidia.com> Bryce Adelstein Lelbach <blelbach@nvidia.com> Michał Dominiak <mdominak@nvidia.com> Eric Niebler <eniebler@fb.com> Kirk Shoop <kirkshoop@fb.com> Lewis Baker <lbaker@fb.com> Lee Howes <lwh@fb.com> David S. Hollman <dshollm@sandia.gov> Gordon Brown <gordon@codeplay.com>

1 Introduction

We offer a possible design direction for Executors to address Nvidia's and Facebook's concerns regarding certain elements of the design of [P0443R10]. The ultimate goal of this paper is to find consensus for a design direction that is acceptable to all parties in both SG1 and LEWG.

2 Required Background

This paper is best understood in relation to the two visions for Executors as represented by [P0443R10] "A Unified Executors Proposal for C++" and [P1341R0] "Unifying Asynchronous APIs in the Standard Library." In addition, the following papers published concurrently should be read prior to this paper:

— [P1658R0] "Suggestions for Consensus on Executors"
— [P1777R0] "One-Way `execute` is a Poor Basis Operation"
— [P1738R0] "The `Executor` Concept Hierarchy Needs a Single Root"

Given the extent of the Executors back-story, this paper makes no attempt to be entirely self-contained.

3 Proposal Summary

We address the concerns raised in [P1777R0] "One-Way `execute` is a Poor Basis Operation," and [P1738R0] "The `Executor` Concept Hierarchy Needs a Single Root," by adopting Nvidia's [P1658R0] "Suggestions for Consensus on Executors" with an augmented eager one-way `execute` function that makes it workable as a low-level primitive for `Sender`/`Receiver`.

We reaffirm Nvidia's position (slightly modified):

> We propose a compromise between competing requirements on executors by eliminating interface-changing properties, establishing a single `Executor` concept, and re-specifying oneway `execute` to permit `Senders` and `Receivers` to be layered on top. Proposed changes to [P0443R10]:
>
> — Eliminate interface-changing properties `oneway` and `bulk_oneway`.
> — Introduce an `Executor` concept based on a member function named `execute` that eagerly submits a function for execution on an execution agent created for it by the executor.
> — Eliminate `OneWayExecutor` and `BulkOneWayExecutor` concepts.
> — Introduce a customizable bulk execution API whose specific shape is left as future work.
> — Introduce a `Scheduler` concept based on a function named `schedule` that lazily creates a `Sender` of a scheduler.
> — Introduce `Senders`, `Receivers`, and related operations with additions described below.

4 Before and After

Before we get into the meat of the proposal, we present some before-and-after tables for comparison of how executor definition and use would look in some common scenarios.

4.1 One-way execution

Before: P0443R10

```
// compiles only if
//   can_require_concept_v<E, oneway_t>
std::require_concept(
    ex, std::execution::oneway)
  .execute(f);
```

After: Proposed

```
// always compiles if ex is an Executor:
std::tbd::execute(ex, f);
```

4.2 One-Way Bulk Execution

Before: P0443R10

```
// compiles only if
//   can_require_concept_v<E, bulk_oneway_t>
std::require_concept(
    ex, std::execution::bulk_oneway)
  .bulk_execute(f, shape, sf);
```

After: Proposed (hypothesized)

```
// always compiles if ex is an Executor
std::tbd::bulk_execute(ex, f, shape, sf);
```

4.3 Authoring bulk executors

Before: P0443R10

```
struct simplest_bulk_executor {
  auto operator<=>(...) const = default;

  template<Invocable SF,
    class S = invoke_result_t<SF&>,
    Invocable<size_t, decay_t<S>&> Op>
  void bulk_execute(
      Op op, size_t n, SF sf) const {
    auto shared = sf();
    for(size_t i = 0; i < n; ++i)
      op(i, shared);
  }
};
```

After: Proposed

```
struct simplest_bulk_executor {
  auto operator<=>(...) const = default;

  template<Invocable F>
  void execute(F f) const {
    std::tbd::invoke_callback(f);
  }
  // All executors are "bulk" by definition.
};
```

5 Suggestions for Executors

5.1 One Executor concept

Pursuant to the arguments in [P1738R0] and [P1658R0] ("Suggestions for Executors") in support of a single `Executor` concept to serve as the root of a hierarchy, we propose that an `Executor` should be a `CopyConstructible` and `EqualityComparable` type that provides a function named `execute` that eagerly submits work on a single execution agent created for it by the executor.

```
// An Executor is something with an .execute()
// member function that takes a nullary callable.
template<class Ex, class C = void(*)()>
concept Executor =
  Invocable<C> &&
  CopyConstructible<remove_cvref_t<Ex>> &&
  EqualityComparable<remove_cvref_t<Ex>> &&
  requires (Ex&& ex, C&& c) {
    ((Ex&&) ex).execute((C&&) c);
  };
```

5.2 Generalize one-way `execute`

[P1777R0] points out particular shortcomings of one-way `execute` that prevent a truly generic `Sender`/`Receiver` design from being layered on top; especially,

— There is no well-defined in-band (i.e., not unspecified) channel for errors that happen either during task submission or after submission but before task execution (say, a time-out for a time-bounded executor), and

— There is no channel, apart from task destruction, whereby an executor can notify a submitted task that it will not be executed (say, if someone requested cancellation).

To address these shortcomings, we propose a refinement of the `Invocable` concept, tentatively named `Callback`, that in addition to being invocable also has `.error(E)` and `.done()` members. `Callback` takes the place of [P1341R0]'s `Receiver`. Subsumption of `Invocable` makes it possible to continue defining one-way `execute` in terms of nullary `Invocable`s as in P0443, with additional guarantees if the `Invocable` models `Callback`.

The `Callback` concept is as follows:

```
// A CallbackSignal is something with .done() and
// .error() member functions.
template<class T, class E = exception_ptr>
concept CallbackSignal =
  requires (T&& t, E&& e) {
    { ((T&&) t).done() } noexcept;
    { ((T&&) t).error(((E&&) e)) } noexcept;
  };

// A Callback is an Invocable that is also a
// CallbackSignal.
template<class T, class... An>
concept Callback =
  Invocable<T, An...> &&
  CallbackSignal<T>;
  // axiom: if invoking T exits by throwing an exception, it is well-defined to
  // call .error(E)
```

The `CallbackSignal` concept permits an algorithm author to write constraints when error types other than `std::exception_ptr` are used. For example:

```
template <Callback<int> C>
  requires CallbackSignal<C, std::error_code> // Note extra constraint!
void frobozzle(C&& c, std::error_code ec) try {
  if (!ec)
    c(42);
  else
    c.error(ec); // OK, we required this expression to be valid.
} catch (...) {
  c.error(std::current_exception());
}
```

Given `Callback`, the semantic constraints on an executor's `.execute()` member are as follows:

— `.execute(fun)` is required to eagerly submit `fun` for execution on an EA that it creates for it. Let `FUN` be `fun` or an object created by copy or move construction from `fun`. If the type of `fun` models `Callback` then the executor guarantees that exactly one of the following is called at some point prior to the destruction of `fun` and all objects copy or move constructed from `fun`: `FUN()`, `FUN.done()`, or `FUN.error(E)`, where `E` is as described below. [*Note:* An executor has wide discretion when deciding which function to call. — *end note*]

  — If `FUN`'s `error` member function is called in response to a submission error, scheduling error, or other internal executor error, let `E` be an expression that refers to that error if `FUN.error(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 executor calls `FUN.error(E)` on an unspecified weakly-parallel EA ([*Note:* An invocation of `Callback`'s `error` member is required to be `noexcept` — *end note*]), and

— If `FUN`'s `error` member function is called in response to an exception that propagates out of the invocation of `FUN`, let `E` be `make_exception_ptr(callback_invocation_error{})` invoked from within a `catch` clause that has caught the exception. The executor calls `FUN.error(E)` on an unspecified weakly-parallel EA, and

— A call to `FUN.done()` is made on an unspecified weakly-parallel EA ([*Note:* An invocation of a `Callback`'s `done` member is required to be `noexcept` — *end note*]).

— If the type of `fun` does *not* model `Callback`, errors are handled in an implementation-defined way.

Enforcing these semantics is aided by defining a `std::tbd::invoke_callback` algorithm (defined in terms of a simpler `std::tbd::try_invoke_callback` algorithm) that takes an `Invocable` and invokes it, funneling errors appropriately:

```
// This definition lives in the stdlib somewhere:
//   runtime_error const __invocation_error {"callback invocation error"};
extern runtime_error const __invocation_error; // exposition only

struct callback_invocation_error : runtime_error, nested_exception {
  callback_invocation_error() noexcept
    : runtime_error(__invocation_error), nested_exception() {}
};

template<CallbackSignal S, class... Args, Invocable<Args...> C>
void std::tbd::try_invoke_callback(S&& s, C&& c, Args&&... args) noexcept
```

Effects: Equivalent to:

```
try {
  invoke((C&&) c, (Args&&) args...);
} catch (...) {
  ((S&&) s).error(make_exception_ptr(callback_invocation_error{}));
}
```

```
template<class... Args, Invocable<Args...> C>
void std::tbd::invoke_callback(C&& fun, Args&&... args) noexcept(see below);
```

Remarks: The expression in the `noexcept` clause is equivalent to

```
is_nothrow_invocable_v<C> || CallbackSignal<C>
```

Effects: Equivalent to

```
if constexpr (CallbackSignal<C>)
  try_invoke_callback((C&&) c, (C&&) c, (Args&&) args...);
else
  invoke((C&&) c, (Args&&) args...);
```

With the above definition of `Executor` and `std::tbd::invoke_callback`, the `inline_executor` becomes the trivial:

```
struct inline_executor {
  auto operator<=>(inline_executor) const = default;
```

```
  template<Invocable C>
  void execute(C&& c) {
    std::tbd::invoke_callback(forward<C>(c));
  }
};
```

5.3 Reexpress `bulk_execute` as an algorithm

[Editor's note: This section is highly speculative.]

In this scheme, all `Executor`s would be required to provide the `execute` member function. As proposed by P1658:

> [...] in order to retain the functionality offered by the `bulk_oneway` property, we propose to introduce a customization point object named `bulk_execute` callable with any `Executor` to create bulk execution.

At present, we can imagine several possible shapes for such an API for bulk execution, and we may decide to standardize one or more of them. In particular, we could ship customizable versions of the following algorithms:

- A bulk execution algorithm with the same shape and semantics as `bulk_execute` in P0443; that is, it takes an executor, an iteration function, a loop induction variable, and a state factory, and eagerly launches execution agents in bulk.
- A eager bulk execution algorithm like the previous, but that additionally takes a `Callback` that can receive the final value of the state variable and that can receive an error or done signal.
- A *lazy* bulk execution algorithm that does not take a `Callback` but that returns a `Sender` of either `void` or of the final value of the state variable.
- A lazy bulk execution algorithm that returns an asynchronous range of `Sender`s, each `Sender` representing a chunk of work to which additional work may be chained.

Whichever bulk execution algorithm(s) we choose to standardize, they will all have the property that they accept any type that models the `Executor` concept; that is, they will have a fallback implementation that uses the one-way `execute` API.

General, efficient, and customizable interfaces for bulk execution in both lazy and eager forms are prerequisites for merging P0443 into the working paper. Their exact specifications are left as future work pending the approval of this design direction.

6 Suggestions for Senders and Receivers

As summarized in Nvidia's P1658, an `Executor` as understood above is a factory for execution agents. However, an `Executor` from P1341R0 is a factory for execution agent *factories*. This difference in kind is why we recommend having separate concepts. In this proposed design, we demonstrate a way in which both P0443's `Executor` and P1341's `Executor`, renamed `Scheduler`, can coexist and interoperate seamlessly.

6.1 One-way `Executors` are a kind of `Sender`

Once we have augmented the one-way `execute` algorithm as described to give clients a way to specify a channel for error and cancellation signals, it becomes possible to define `Sender`'s `submit` operation such that one-way `Executors` satisfy the `Sender` concept. We do that by defining `std::tbd::submit` as a customization point as defined below:

```
template<class T, class C>
concept _SenderTo = // exposition only
```

```
  requires (T&& t, C&& c) { {((T&&) t).submit((C&&) c)} noexcept; } ||
  requires (T&& t, C& c) { ((T&&) t).submit(c); };
```

```
template<class S, CallbackSignal C>
void std::tbd::submit(S&& s, C&& c) noexcept requires see below;
```

Submit a `Sender` of unknown type and a `Callback` of unknown type.

Remarks: The expression in the `requires` clause is equivalent to: `_SenderTo<S, C>`.

Effects: If `noexcept(((S&&) s).submit((C&&) c))` is `true` or if the current EA doesn't support exceptions, equivalent to: `((S&&) s).submit((C&&) c)`; otherwise,

```
try_invoke_callback((C&&) c, [&]{((S&&) s).submit(c);});
```

```
template<Callback C, Executor<C> E>
void std::tbd::submit(E&& e, C&& c) noexcept requires see below;
```

Submit a nullary `Callback` with a one-way `Executor`.

Remarks: The expression in the `requires` clause is equivalent to: `(!_SenderTo<E, C>)`.

Effects: Equivalent to: `((E&&) e).execute((C&&) c);`.

```
template<class E, Callback<E> C>
void std::tbd::submit(E&& e, C&& c) noexcept requires see below;
```

Submit a `Callback` that expects an executor with a one-way `Executor`.

Remarks: The expression in the `requires` clause is equivalent to: `(!_SenderTo<E, C>) && Executor<E, with-subex>`, where `with-subex` is the type of the variable `c2` in the *Effects:* clause below.

Effects: Equivalent to

```
auto c2 = compose_callback((C&&) c, [=]{ return e; });
((E&&) e).execute(move(c2));
```

With the above definition of `submit`, one-way `Executors` become full-fledged `Senders` that can participate in any `Sender` combinator library.

6.2 One-way `Executors` are a kind of `Scheduler`

Once we have defined `submit` such that `Executor` satisfy the `Sender` concept, we can take the additional step of defining `schedule` such that `Executors` also satisfy the `Scheduler` concept. Recall that an `Executor` is an EA factory, and that a `Scheduler` is an EA factory factory. We can make an `Executor` an EA factory factory by defining an overload of `schedule` that takes an `Executor` and returns the `Executor` unmodified. (`schedule` is required to return a `Sender`, but we just defined `submit` such that `Executors` *are* `Senders`.)

The `schedule` customization point is defined as follows:

```
template<class S>
auto std::tbd::schedule(S&& s) noexcept requires(see below);
```

Remarks: Returns a `Sender` of a sub-scheduler with the following properties:

— It may be `submit`'ed at most once with a `Callback` that expects a `Scheduler`.
— On success, the `Callback` shall be called on an EA created for it by `S`.

— On success, the `Callback` will be called with a `Scheduler` denoting the EA on which the `Callback` is executing.
— If an error happens while enqueuing the `Callback`, or if an exception propagates out of the `Callback`, the `Callback`'s `.error()` member will be called with the error on an unspecified weakly-parallel EA, or `std::terminate()` will be called.
— If the `Callback` is unstaged without execution, the `Callback`'s `.done()` member will be called on an unspecified weakly-parallel EA.

Remarks: The expression in the `requires` clause is equivalent to: `((S&&)).schedule()`.

Effects: Equivalent to: `return ((S&&) s).schedule();`.

```
template<Executor E>
auto std::tbd::schedule(E&& e) noexcept(see below);
```

Remarks: The expression in the `noexcept` is equivalent to

```
is_nothrow_constructible_v<remove_cvref_t<E>, E>
```

Effects: Equivalent to: `return e;`

With the above definition, generic code can treat `Executor`s as if they were `Scheduler`s. This is easy to see: for a given object of type `T`, one can construct the corresponding factory-of-`T` that always returns the object you started with. In other words, there is a trivial isomorphism mapping `T`'s into factories-of-`T`.

6.3 Factor submit into a composition of more basic operations

Nvidia's P1658 observes that `Sender`'s `submit` operation conflates two separate responsibilities: (1) connecting a `Sender` to a `Receiver` (aka, `Callback`), and (2) submitting the resulting async operation for execution. The former could be an expensive operation that some users may want to decouple from the operation's launch.

Also, every asynchronous operation invariably has some *state* associated with it. Separating `submit` into two operations gives us a chance to externalize this state. That is of critical importance when awaiting a `Sender` in a coroutine: the ideal place for the operation's state is on the coroutine frame, where the state would not incur an extra allocation.

A future paper will propose refactoring the `Sender`'s `submit` operation into a `connect` operation that joins a `Sender` with a `Callback` and returns a nullary `Invocable` which, when invoked, launches the operation. The `connect` primitive looks roughly as follows:

```
template<class S, class C>
concept _SenderTo =
  CallbackSignal<C> &&
  requires (S&& s, C&& c) {
    { ((S&&) s).connect((C&&) c) } -> Invocable;
  };

// Connect a Sender to a Callback and return the async operation's state:
template<class S, class C>
auto std::tbd::connect(S&& s, C&& c) requires _SenderTo<S, C>;
```

6.4 Other considerations

6.4.1 Polymorphic Executor Wrapper

We note that removing interface-changing properties necessitates changes to the polymorphic executor wrapper. We do not propose any suggested replacement at this time. The re-specification of the polymorphic wrapper is left as future work.

6.4.2 Scheduler and Sender in P0443

We observe that the `Scheduler` and `Sender` parts of this design sketch are separable from the `Executor` and `Callback` parts. We may choose to amend P0443 with `Executor` and `Callback`, and propose `Scheduler` and `Sender` in a separate paper. Alternatively, we could move it all forward together.

We look for guidance from SG1 and LEWG as to how to proceed.

7 Appendix 1: Prototype Implementation

This prototype implementation can be found online here: https://godbolt.org/z/RwYbYS.

```
/**********************************************************
 *
 * As in P0443, oneway execute is a function that takes
 * a nullary callable, creates an execution agent to execute
 * it, and returns void. What is new is the ability to pass
 * an enriched callable to execute that, in addition to being
 * a nullary callable, also has .error() and .done() members.
 * The concept that enriched callables model is named "Callback"
 * and it subsumes Invocable.
 *
 * concept Invocable<C, An...> { // (already standard)
 *   void operator()(An...);
 * }
 *
 * concept CallbackSignal<C, E = std::exception_ptr> {
 *   void done() noexcept;
 *   void error(E) noexcept;
 * }
 *
 * concept Callback<C, An...> {
 *   requires Invocable<C, An...> && CallbackSignal<C>;
 * }
 *
 * concept Executor< E, C [=void(*)()] > {
 *   requires Invocable< C >;
 *   void execute(C);
 * }
 *
 * Authors of oneway executors can accept and execute simple
 * callables and use an implementation-defined scheme to deal
 * with errors. If passed a Callback, an executor should send
 * errors to the Callback's .error() channel. This is made
 * simple with a invoke_callback helper that invokes the
 * callable inline, routing errors to the error channel if the
 * callable is in fact a Callback.
 *
 * Here, for example, is the inline_executor:
 *
 *   struct inline_executor {
 *     template<Invocable C>
 *     void execute(C c) {
 *       std::tbd::invoke_callback(std::move(c));
 *     }
 *   };
 *
 * With Callbacks instead of raw Invocables, we can implement
 * Sender/Receiver in a way that is compatible with oneway
 * executors. It is possible to define the submit() and
```

```
 * schedule() customization points such that a oneway executor,
 * when used with a Callback, satisfies the requirements of
 * Sender and even Scheduler. I show the code below the break.
 *
 * The final section demonstrates how Sender/Callback/Scheduler
 * would be defined in the presence of Executor/Callback
 *
 ************************************************************/

#include <string>
#include <utility>
#include <type_traits>
#include <functional>
#include <exception>
#include <stdexcept>

namespace std {
template<class A, class B>
concept Same = __is_same(A, B) && __is_same(B, A);

template<class C, class... An>
concept Invocable = requires (C&& c, An&&... an) {
  std::invoke((C&&) c, (An&&) an...);
};

namespace tbd {
// A CallbackSignal is something with .done() and
// .error() member functions.
template<class T, class E = exception_ptr>
concept CallbackSignal =
  requires (T&& t, E&& e) {
    {((T&&) t).done()} noexcept;
    {((T&&) t).error(((E&&) e))} noexcept;
  };

// A Callback is an Invocable that is also a
// CallbackSignal.
template<class T, class... An>
concept Callback =
  Invocable<T, An...> &&
  CallbackSignal<T>;

// A Executor is something with an .execute()
// member function that takes a nullary callable.
template<class T, class C = void(*)()>
concept Executor =
  Invocable<C> &&
  requires (T&& t, C&& c) {
    ((T&&) t).execute((C&&) c);
  };
```

```
// Definition in the stdlib somewhere:
extern runtime_error const invocation_error_; // = "invocation error";

// All exceptions from invocations of Callbacks are
// wrapped in callback_invocation_error:
struct callback_invocation_error : runtime_error, nested_exception {
  callback_invocation_error() noexcept
    : runtime_error(invocation_error_)
    , nested_exception()
  {}
};

// execute the callable immediately in the current context
// and send any errors to the handler.
template<CallbackSignal F, class... Args, Invocable<Args...> C>
constexpr void try_invoke_callback(F&& f, C&& c, Args&&... args) noexcept
try {
  invoke((C&&) c, (Args&&) args...);
} catch (...) {
  ((F&&) f).error(make_exception_ptr(callback_invocation_error{}));
}

// execute the callable immediately in the current context
// and send any errors to the handler (if any).
template<class... Args, Invocable<Args...> C>
constexpr void invoke_callback(C&& c, Args&&... args)
noexcept(is_nothrow_invocable_v<C, Args...> || CallbackSignal<C>) {
  if constexpr (CallbackSignal<C>)
    try_invoke_callback((C&&) c, (C&&) c, (Args&&) args...);
  else
    invoke((C&&) c, (Args&&) args...);
}

template<Invocable Fn, Invocable<invoke_result_t<Fn>> C>
auto compose_callback(C&& c, Fn&& fn) {
  struct __local {
    remove_cvref_t<C> c_;
    remove_cvref_t<Fn> fn_;
    void operator()() && {
      invoke((C&&) c_, invoke((Fn&&) fn_));
    }
    void done() && noexcept requires CallbackSignal<C> {
      ((C&&) c_).done();
    }
    void error(exception_ptr ep) && noexcept requires CallbackSignal<C> {
      ((C&&) c_).error(move(ep));
    }
  };
  return __local{(C&&) c, (Fn&&) fn};
}
```

```
inline constexpr struct __execute_fn {
  // Pass the callable and the handler to the executor to create an
  // execution agent for execution.
  template<Invocable C, Executor<C> E>
  constexpr void operator()(E&& e, C&& c) const
      noexcept(noexcept(((E&&) e).execute((C&&) c))) {
    ((E&&) e).execute((C&&) c);
  }
} const execute{};

struct inline_executor {
  template<Invocable C>
  void execute(C c) {
    invoke_callback(move(c));
  }
};

}} // namespace std::tbd

//
// ^^^^^ Executor machinery ends here      ^^^^^
// vvvvv Sender/Scheduler machinery below vvvvv
//

// A helper concept used to constrain part of the implementation
// of submit.
template<class T, class C>
concept _SenderTo =
  std::tbd::CallbackSignal<C> &&
  (requires (T&& t, C&& c) { {((T&&) t).submit(((C&&) c))} noexcept; } ||
   requires (T&& t, C& c) { ((T&&) t).submit(c); });

// The implementation of the submit customization point.
inline constexpr struct __submit_fn {
  inline static constexpr auto __with_subex = [](auto e, auto c) {
    return std::tbd::compose_callback(std::move(c), [=]{return e;});
  };
  using __with_subex_fn = decltype(__with_subex);
  template<class E, class C>
  using __with_subex_t = std::invoke_result_t<__with_subex_fn, E, C>;

  // Handle s.submit(c):
  template<class C, _SenderTo<C> S>
  void operator()(S s, C c) const noexcept {
    if constexpr (noexcept(std::move(s).submit(std::move(c))))
      std::move(s).submit(std::move(c));
    else
      std::tbd::try_invoke_callback(std::move(c), [&]{std::move(s).submit(c);});
  }
  // For use when passed a Executor and a nullary
  // Callback.
```

```
  template<std::tbd::Callback C, std::tbd::Executor<C> E>
  void operator()(E e, C c) const noexcept requires (!_SenderTo<E, C>) {
    std::move(e).execute(std::move(c));
  }
  // For use when passed a Executor and a Callback that
  // is expecting to be passed a sub-executor.
  template<class E, std::tbd::Callback<E> C>
    requires std::tbd::Executor<E, __with_subex_t<E, C>>
  void operator()(E e, C c) const noexcept requires (!_SenderTo<E, C>) {
    auto c2 = __with_subex(e, std::move(c));
    ((E&&) e).execute(std::move(c2));
  }
} const submit {};

// A multi-type concept for constraining compatible Senders
// and Callbacks.
template<class S, class C>
concept SenderTo =
  requires (S&& s, C&& c) {
    submit((S&&) s, (C&&) c);
  };

// Below are some nasty implementation details for the
// Scheduler concept, which is inherently self-recursive.
// Basically a Scheduler is a type with a .schedule()
// member that returns a Sender of a Scheduler. We hackishly
// test that constraint with an archetypical Callback that
// requires its argument to be a either a Scheduler, or the
// type we started with. (That terminates the recursion for
// sane definitions of Schedulers.)
template<class, class>
inline constexpr bool __is_scheduler_or_v = false;
template<class S, class T>
concept _IsSchedulerOr = __is_scheduler_or_v<S, T>;
template<class T>
struct __scheduler_callback {
  template<_IsSchedulerOr<T> S>
  void operator()(S) const {}
  void done() const noexcept {}
  void error(std::exception_ptr) const noexcept {}
};

// With this definition of schedule(), a type with .schedule()
// is a Scheduler, but so too is a Executor.
struct __schedule_fn {
  template<class S>
    requires requires(S s) { s.schedule(); }
  auto operator()(S s) const noexcept {
    return s.schedule();
  }
  template<std::tbd::Executor E>
```

```
  auto operator()(E e) const noexcept {
    return e;
  }
};
inline constexpr __schedule_fn const schedule {};

template<class T>
concept Scheduler = requires (T t) {
  {schedule(t)} -> SenderTo<__scheduler_callback<T>>;
};

template<class S, class T>
  requires std::Same<S, T> || Scheduler<S>
inline constexpr bool __is_scheduler_or_v<S, T> = true;

//
// ^^^^^ Sender/Scheduler machinery ends here ^^^^^
// vvvvv Test code below            vvvvv
//

template<class Fn>
struct terminate_on_error : Fn {
  constexpr terminate_on_error(Fn fn)
    : Fn(std::move(fn)) {}
  void done() const noexcept
  {}
  template <class E>
  [[noreturn]] void error(E&&) const noexcept {
    std::terminate();
  }
};

// Show how an Executor works with execute with either
// a raw Invocable or a Callback
void foo0() {
  std::tbd::inline_executor ie;

  std::tbd::execute(ie, []() noexcept {});
  std::tbd::execute(ie, terminate_on_error{[]() noexcept {}});
}

struct inline_scheduler {
  struct inline_executor {
    template<std::Invocable C>
    void execute(C c) && noexcept {
      std::tbd::invoke_callback(std::move(c));
    }
  };

  std::tbd::inline_executor schedule() {
    return {};
```

```
  }
};
static_assert(Scheduler<std::tbd::inline_executor>);
static_assert(Scheduler<inline_scheduler>);

// Show how execute works with Schedulers and raw
// Invocables and also Callbacks.
void foo1() {
  inline_scheduler is;

  std::tbd::execute(schedule(is),
                    []() noexcept {});
  std::tbd::execute(schedule(is),
                    terminate_on_error{[]() noexcept {}});
}

template<class... An>
struct inline_callback {
  void done() const noexcept {}
  void operator()(An...) const noexcept {}
  [[noreturn]] void error(std::exception_ptr) const noexcept {
    std::terminate();
  }
};

template<class... An>
struct inline_sender {
  std::tuple<An...> an_;

  template<std::tbd::Callback<An&...> C>
  void submit(C&& c) {
    std::apply([&](An&... an) {
                 std::invoke((C&&) c, an...);
               },
               an_);
  }
};

// Show how submit works with a Sender and a Callback
void foo2() {
  inline_sender<int, float> ie{std::make_tuple(42, 42.0)};

  submit(ie, inline_callback<int, float>{});
}

struct full_inline_scheduler {
  struct inline_sender {
    template<std::Invocable C>
    void execute(C c) && noexcept(
        noexcept(std::tbd::invoke_callback((C&&) c))) {
      std::tbd::invoke_callback((C&&) c);
```

```
    }
    template<std::tbd::Callback<full_inline_scheduler> C>
    void submit(C c) && noexcept(
        noexcept(((C&&) c)(full_inline_scheduler{}))) {
      std::invoke((C&&) c, full_inline_scheduler{});
    }
  };

  inline_sender schedule() {
    return {};
  }
};

// Show how a Scheduler can work with execute, and how an
// Executor can work with submit:
void foo3() {
  full_inline_scheduler is;

  std::tbd::execute(schedule(is), inline_callback{});
  submit(schedule(is), inline_callback<full_inline_scheduler>{});

  submit(std::tbd::inline_executor{},
         terminate_on_error{[]{}});
  submit(std::tbd::inline_executor{},
         terminate_on_error{[](std::tbd::inline_executor){}});
}
```

8 References

[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

[P1658R0] Jared Hoberock. 2019. P1658R0: Suggestions for Consensus on Executors.
http://wg21.link/P1658R0

[P1738R0] Eric Niebler. 2019. P1738R0: The `Executor` Concept Hierarchy Needs a Single Root.
http://wg21.link/P1738R0

[P1777R0] Eric Niebler, Lewis Baker, Lee Howes, and Kirk Shoop. 2019. P1777R0: One-way `execute` is a Poor Basis Operation.
http://wg21.link/P1777R0