remove ensure_started and start_detached from P2300

Draft Proposal

Document #: D3187R0
Date: 2024-03-19
Project: Programming Language C++
Audience: LEWG Library Evolution

Reply-to: Kirk Shoop
<kirk.shoop@gmail.com>
Lewis Baker
<lewissbaker@gmail.com>

1 Introduction

The current version of P2300 includes two algorithms that operate on senders, `start_detached()` and `ensure_started()` which provide functionality that enables sender-based operations to be launched eagerly, allowing them to run in the background.

However, the current design of these facilities would introduce major footguns in the standard library as they are deceptively enticing yet difficult to use correctly, making it much harder to guarantee code safely cleans up resources used by the eagerly launched operations.

The paper [P3149R1] proposes an alternative async-scope facility that provides the ability to launch work eagerly while still retaining the ability to later join the completion of that work, allowing cleanup to still be performed safely.

This paper proposes that the `start_detached()` and `ensure_started()` facilities be removed from P2300 before it is merged into the working draft.

2 `ensure_started`

Senders are generally assumed to be safe to destroy at any point. It is common to have algorithms that compose senders but that do not guarantee to connect/start their child senders, whether due to exceptions or some short-circuiting behaviour of the algorithm.

However, `ensure_started()` returns a sender that owns work that is potentially already executing asynchronously in the background.

If this `ensure_started()` sender is destroyed without being connected/started then we need to specify the semantics of what will happen to that already-started asynchronous work. There are four common strategies for what to do in this case:

1. Block in the destructor until the asynchronous operation completes - can easily lead to deadlock.
2. Detaching from the async operation, letting it run to completion in the background - makes it hard to implement clean shut-down as you don't necessarily know when resources used by the async operation can be safely destroyed.
3. Treat it as undefined-behavior.
4. Terminate the program - the strategy that `std::thread` takes.

The current `ensure_started()` wording chooses option 2 as the least worst option, but all of the options are generally bad options.

The `ensure_started()` facility is especially dangerous as the returned sender is much more likely to be composed with sender adaptors into other operations and work most of the time, but introduce non-obvious data races that may only occur on failure-codepaths.

The functionality previously provided by `std::execution::ensure_started()`, which allows eagerly starting an operation and then later observing the result, can be obtained by using facilities proposed by [P3149].

Instead of writing:

```
std::execution::ensure_started(sender)
```

you would write:

```
std::execution::spawn_future(sender, scope)
```

where `scope` is some `std::execution::counting_scope` object.

If the sender returned by `spawn_future()` is destroyed before it is connected/started then a stop request is sent and the sender detaches from the operation.

The operation can still then complete asynchronously in the background and can be later joined using the `scope.join()` facility on the `counting_scope` object.

This allows callers to wait until any eagerly launched operation completes and stops accessing resources and thus can safely sequence destruction of those resources to occur after all uses.

3 `start_detached`

Like `ensure_started()`, the `start_detached()` facility allows you to eagerly launch an async operation in the background, only without returning a sender that can be used to later join the work.

This has the same challenges as `ensure_started()` with regards to support for safe cleanup of resources used by the launched operations.

However, it's slightly less dangerous than `ensure_started()` because it has a scarier name and does not return a value that users might mistakenly think they can safely compose with other algorithms.

It still requires other out-of-band mechanisms for joining the detached work before you can safely destroy resources used by the detached operations are destroyed. For example, by using ad-hoc GC such as shared_ptr or other synchronization primitives.

The functionality provided by `start_detached(s)` can be provided instead using an async scope by calling `std::execution::spawn(s, scope)` for some async-scope, `scope`.

4 Proposal

Remove `ensure_started` and `start_detached` from [P2300R7] by removing the following sections.

> 4.21.13. `execution::ensure_started`
>
> ```
> execution::sender auto ensure_started(
>     execution::sender auto sender
> );
> ```
>
> Once ensure_started returns, it is known that the provided sender has been connected and `start` has been called on the resulting operation state (see § 5.2 Operation states represent work); in other words, the work described by the provided sender has been submitted for execution on the appropriate execution resources. Returns a sender which completes when the provided sender completes and sends values equivalent

to those of the provided sender.

If the returned sender is destroyed before `execution::connect()` is called, or if `execution::connect()` is called but the returned operation-state is destroyed before `execution::start()` is called, then a stop-request is sent to the eagerly launched operation and the operation is detached and will run to completion in the background. Its result will be discarded when it eventually completes.

Note that the application will need to make sure that resources are kept alive in the case that the operation detaches. e.g. by holding a `std::shared_ptr` to those resources or otherwise having some out-of-band way to signal completion of the operation so that resource release can be sequenced after the completion.

4.22.1. `execution::start_detached`

void start_detached(
    execution::sender auto sender
);

Like ensure_started, but does not return a value; if the provided sender sends an error instead of a value, `std::terminate` is called.

11.9.6.17. execution::ensure_started [exec.ensure.started]
ensure_started eagerly starts the execution of a sender, returning a sender that is usable as intput to additional sender algorithms.

Let ensure-started-env be the type of an execution environment such that, given an instance e, the expression get_stop_token(e) is well-formed and has type stop_token.

The name ensure_started denotes a customization point object. For some subexpression s, let S be decltype((s)). If sender_in<S, ensure-started-env> or constructible_from<decay_t<env_of_t>, env_of_t> is false, ensure_started(s) is ill-formed. Otherwise, the expression ensure_started(s) is expression-equivalent to:

tag_invoke(ensure_started, get_completion_scheduler(get_env(s)), s),
if that expression is valid.

Mandates: The type of the tag_invoke expression above satisfies sender.

Otherwise, tag_invoke(ensure_started, s), if that expression is valid.

Mandates: The type of the tag_invoke expression above satisfies sender.

Otherwise, constructs a sender s2, which:

Creates an object sh_state that contains a stop_source, an initially null pointer to an operation state awaitaing completion, and that also reserves space for storing:

the operation state that results from connecting s with r described below, and

the sets of values and errors with which s can complete, with the addition of exception_ptr.

the result of decay-copying get_env(s).

s2 shares ownership of sh_state with r described below.

Constructs a receiver r such that:

When set_value(r, args...) is called, decay-copies the expressions args... into sh_state. It then checks sh_state to see if there is an operation state awaiting completion; if so, it notifies the operation state that the results are ready. If any exceptions are thrown, the exception is caught and set_error(r, current_exception()) is called instead.

When set_error(r, e) is called, decay-copies e into sh_state. If there is an operation state awaiting completion, it then notifies the operation state that the results are ready.

When set_stopped(r) is called, it then notifies any awaiting operation state that the results are ready.

get_env(r) is an expression e of type ensure-started-env such that get_stop_token(e) is well-formed and returns the results of calling get_token() on sh_state's stop source.

r shares ownership of sh_state with s2. After r has been completed, it releases its ownership of sh_state.

Calls get_env(s) and decay-copies the result into sh_state.

Calls connect(s, r), resulting in an operation state op_state2. op_state2 is saved in sh_state. It then calls start(op_state2).

When s2 is connected with a receiver out_r of type OutR, it returns an operation state object op_state that contains:

An object out_r' of type OutR decay-copied from out_r,

A reference to sh_state,

A stop callback of type optional<stop_token_of_t<env_of_t>::callback_type>, where stop-callback-fn is the unspecified class type:

```
struct stop-callback-fn {
  stop_source& stop_src_;
  void operator()() noexcept {
    stop_src_.request_stop();
  }
};
```

s2 transfers its ownership of sh_state to op_state.

When start(op_state) is called:

If r has already been completed, then let CF be whichever completion function was used to complete r. Calls CF(out_r', args2...), where args2... is a pack of xvalues referencing the subobjects of sh_state that have been saved by the original call to CF(r, args...) and returns.

Otherwise, it emplace constructs the stop callback optional with the arguments get_stop_token(get_env(out_r')) and stop-callback-fn{stop-src}, where stop-src refers to the stop source of sh_state.

Then, it checks to see if stop-src.stop_requested() is true. If so, it calls set_stopped(out_r').

Otherwise, it sets sh_state operation state pointer to the address of op_state, registering itself as awaiting the result of the completion of r.

When r completes it will notify op_state that the result are ready. Let CF be whichever completion function was used to complete r. op_state's stop callback optional is reset. Then CF(std::move(out_r'), args2...) is called, where args2... is a pack of xvalues referencing the subobjects of sh_state that have been saved by the original call to CF(r, args...).

[Note: If sender s2 is destroyed without being connected to a receiver, or if it is connected but the operation state is destroyed without having been started, then when r completes and it releases its shared ownership of sh_state, sh_state will be destroyed and the results of the operation are discarded.
– end note]

Given a subexpression s, let s2 be the result of ensure_started(s). The result of get_env(s2) shall return an lvalue reference to the object in sh_state that was initialized with the result of get_env(s).

Given subexpressions s2 and e where s2 is a sender returned from ensure_started or a copy of such, let S2 be decltype((s2)) and let E be decltype((e)). The type of tag_invoke(get_completion_signatures, s2, e) shall be equivalent to:

```
make_completion_signatures<
  copy_cvref_t<S2, S>,
  ensure-started-env,
  completion_signatures<set_error_t(exception_ptr&&),
                        set_error_t(Es)...>,
  set-value-signature,
  error-types>
```

where Es is a (possibly empty) template parameter pack, set-value-signature is the alias template:

```
template<class... Ts>
  using set-value-signature =
    completion_signatures<set_value_t(decay_t&&...)>;
```

and error-types is the alias template:

```
template
  using error-types =
    completion_signatures<set_error_t(decay_t&&)>;
```

Let s be a sender expression, r be an instance of the receiver type described above, s2 be a sender returned from ensure_started(s) or a copy of such, r2 is the receiver to which s2 is connected, and args is the pack of subexpressions passed to r's completion function CSO when s completes. s2 shall invoke CSO(r2, args2...) where args2 is a pack of xvalue references to objects decay-copied from args, or by calling set_error(r2, e2) for some subexpression e2. The objects passed to r2's completion operation shall be valid until after the completion of the invocation of r2's completion operation.

11.9.7.1. execution::start_detached [exec.start.detached]
start_detached eagerly starts a sender without the caller needing to manage

the lifetimes of any objects.

The name start_detached denotes a customization point object. For some subexpression s, let S be decltype((s)). If S does not satisfy sender, start_detached is ill-formed. Otherwise, the expression start_detached(s) is expression-equivalent to:

tag_invoke(start_detached, get_completion_scheduler(get_env(s)), s), if that expression is valid.

Mandates: The type of the tag_invoke expression above is void.

Otherwise, tag_invoke(start_detached, s), if that expression is valid.

Mandates: The type of the tag_invoke expression above is void.

Otherwise:

Let R be the type of a receiver, let r be an rvalue of type R, and let cr be a lvalue reference to const R such that:

The expression set_value(r) is not potentially-throwing and has no effect,

For any subexpression e, the expression set_error(r, e) is expression-equivalent to terminate(),

The expression set_stopped(r) is not potentially-throwing and has no effect, and

The expression get_env(cr) is expression-equivalent to empty_env{}.

Calls connect(s, r), resulting in an operation state op_state, then calls start(op_state).

If the function selected above does not eagerly start the sender s after connecting it with a receiver that ignores value and stopped completion operations and calls terminate() on error completions, the behavior of calling start_detached(s) is undefined.

5 References

[P2300R7] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Bryce Adelstein Lelbach. 2023-04-21. ‘std::execution‘.
https://wg21.link/p2300r7