`std::execution::sequence`

Document Number: P4320R0
Date: 2026-07-15
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes an algorithm, `std::execution::sequence`, which allows asynchronous operations to be composed serially.

Background

As adopted into the working draft in St. Louis [1] `std::execution` includes an algorithm which allows for parallel composition of asynchronous operations (i.e. where the algorithms are allowed to make forward progress together): `std::execution::when_all`. It includes no analogue thereto for serial composition (i.e. where each algorithm only makes forward progress upon the completion of the preceding operation).

Such a primitive is provided by libunifex [2] and stdexec [3].

Discussion

The ability to serially compose work is facially useful. Without `std::execution::sequence` users who wish to serially compose senders must do so using `std::execution::let_value` which is not only less convenient and readable than a purpose-made algorithm, but also raises the concern of so-called “dynamic asynchrony” ([4] at §6.1).

Proposal

[execution.syn]

```
namespace std::execution {
  [...]
  // [exec.adapt], sender adaptors
  template<class-type D>
    struct sender_adaptor_closure { };
```

```
struct starts_on_t { unspecified };
struct continues_on_t { unspecified };
struct on_t { unspecified };
struct schedule_from_t { unspecified };
struct then_t { unspecified };
struct upon_error_t { unspecified };
struct upon_stopped_t { unspecified };
struct let_value_t { unspecified };
struct let_error_t { unspecified };
struct let_stopped_t { unspecified };
struct bulk_t { unspecified };
struct bulk_chunked_t { unspecified };
struct bulk_unchunked_t { unspecified };
struct when_all_t { unspecified };
struct when_all_with_variant_t { unspecified };
struct into_variant_t { unspecified };
struct stopped_as_optional_t { unspecified };
struct stopped_as_error_t { unspecified };
struct associate_t { unspecified };
struct spawn_future_t { unspecified };
struct sequence_t { unspecified };

inline constexpr unspecified write_env{};
inline constexpr unspecified unstoppable{};
inline constexpr starts_on_t starts_on{};
inline constexpr continues_on_t continues_on{};
inline constexpr on_t on{};
inline constexpr schedule_from_t schedule_from{};
inline constexpr then_t then{};
inline constexpr upon_error_t upon_error{};
inline constexpr upon_stopped_t upon_stopped{};
inline constexpr let_value_t let_value{};
inline constexpr let_error_t let_error{};
inline constexpr let_stopped_t let_stopped{};
inline constexpr bulk_t bulk{};
inline constexpr bulk_chunked_t bulk_chunked{};
inline constexpr bulk_unchunked_t bulk_unchunked{};
inline constexpr when_all_t when_all{};
inline constexpr when_all_with_variant_t when_all_with_variant{};
inline constexpr into_variant_t into_variant{};
inline constexpr stopped_as_optional_t stopped_as_optional{};
inline constexpr stopped_as_error_t stopped_as_error{};
inline constexpr associate_t associate{};
```

```
  inline constexpr spawn_future_t spawn_future{};
  inline constexpr sequence_t sequence{};
}
```

[exec.sequence]

Note: This is a new section.

`sequence` adapts any number of input senders into a sender which completes when all input senders have completed, with each input sender's asynchronous operation being started only after the completion of the preceding input sender's asynchronous operation.

The name `sequence` denotes a customization point object. Let `sndrs` be a pack of subexpressions and let `Sndrs` be a pack of the types `decltype((sndrs))...`. The expression `sequence(sndrs...)` is ill-formed if `(sender<Sndrs> && ...)` is `false`.

Otherwise, the expression `sequence(sndrs...)` is expression-equivalent to:

- `just()` if `sizeof...(sndrs)` is `0`,
- `sndrs...` if `sizeof...(sndrs)` is `1`, otherwise
- *`make-sender`*`(sequence, {}, sndrs...)`

Let *`sequence-tag`* denote a unique, empty class type.

Let the expression `sequence.transform_sender(s, es...)` be expression-equivalent to:

```
make-sender(
  sequence-tag{}, tuple(std::forward_like<decltype((s))>(sndrs)...))
```

Where `sndrs` denotes the pack which would be declared by:

```
auto&& [_, _, ...sndrs] = s;
```

The exposition-only class template *`impls-for`* ([exec.snd.expos]) is specialized for *`sequence-tag`* as follows:

```
namespace std::execution {
  template<>
    struct impls-for<sequence-tag> : default-impls {
      static constexpr auto get-state = see below;
      static constexpr auto start = see below;

      template<class Sndr, class... Env>
        static consteval void check-types();
    };
```

```
}

template<class Sndr, class... Env>
  static consteval void check-types();
```

Let `Is` be the pack of integral template arguments of the `integer_sequence` specialization denoted by *indices-for*<Sndr>.

Effects: Equivalent to:

```
auto fn = []<class Child, size_t I>() {
  auto fn = []<class QualifiedChild>() {
    auto cs = get_completion_signatures<QualifiedChild, Env...>();
    constexpr bool last = I != sizeof...(Is) - 1;
    using value_completions = gather-signatures<
      set_value_t, decltype(cs), tuple, variant-or-empty>>;
    if constexpr (!(last || is_same_v<value_completions, variant<tuple<>>>))
    {
      throw unspecified-exception{};
    }
  };
  if constexpr (I) {
    fn.template operator()<Child>();
  } else {
    using type = remove_cvref_t<Child>;
    if constexpr (!is_constructible_v<Child, type>) {
      throw unspecified-exception{};
    }
    fn.template operator()<type>();
  }
};
(fn.template operator()<child-type<Sndr, Is>, Is>(), ...);
```

Let *sequence-state* denote the following exposition-only class template:

```
template<class Rcvr, class Sndr, class... Sndrs>
struct sequence-state {
  template<size_t I>
  struct receiver {                    // exposition only
    using receiver_concept = receiver_tag;

    sequence-state& state;             // exposition only
    Rcvr& rcvr;                        // exposition only

    template<class... Args>
```

```
    constexpr void set_value(Args&&... args) noexcept {
      state.impl<I>(rcvr, execution::set_value, std::forward<Args>(args)...);
    }
    template<class... Args>
    constexpr void set_error(Args&&... args) noexcept {
      state.impl<I>(rcvr, execution::set_error, std::forward<Args>(args)...);
    }
    template<class... Args>
    constexpr void set_stopped(Args&&... args) noexcept {
      state.impl<I>(
        rcvr, execution::set_stopped, std::forward<Args>(args)...);
    }

    constexpr env_of_t<const Rcvr&> get_env() const noexcept {
      return execution::get_env(rcvr);
    }
  };

  variant<
    connect_result_t<Sndr, receiver<0>>,
    see below> ops;              // exposition only
  tuple<Sndrs...> sndrs;         // exposition only

  template<size_t I, class Tag, class... Args>
  constexpr void impl(Rcvr& rcvr, Tag tag, Args&&... args) noexcept {
    constexpr bool last = I == sizeof...(Sndrs);
    constexpr bool success = is_same_v<Tag, set_value_t>;
    if constexpr (last || !success) {
      tag(std::move(rcvr), std::forward<Args>(args)...);
    } else {
      auto&& next = std::get<I>(sndrs);
      receiver<I + 1> r{rcvr, *this};
      constexpr bool nothrow = noexcept(
        connect(std::move(next), std::move(r)));
      auto mkop = [&] {
        return connect(std::move(next), std::move(r));
      };
      try {
        auto& op = ops.template emplace<I + 1>(emplace-from{mkop});
        start(op);
      } catch (...) {
        if constexpr (nothrow) {
          set_error(std::move(rcvr), current_exception());
```

```
        }
      }
    }
  }

  template<class... Ts>
  constexpr sequence-state(Rcvr& rcvr, Sndr&& sndr, Ts&&... ts)
    noexcept(
      (is_nothrow_constructible_v<Sndrs, Ts> && ...) &&
      noexcept(connect(declval<Sndr>(), declval<receiver<0>>())))
    : ops(in_place_index<0>,
          emplace-from{[&] {
            return connect(
              std::forward<Sndr>(sndr), receiver<0>{rcvr, *this});
          }),
      sndrs(std::forward<Ts>(ts)...) {}
};
```

Let `Is` be a pack of the template arguments of the type denoted by `index_sequence_for<Sndrs...>`. The unspecified template arguments of the type of the *ops* exposition-only member of template specializations of *sequence-state* are the types contained by the pack `connect_result_t<Sndrs, receiver<Is + 1>>...`.

The member *impls-for*`<sequence_t>::`*get-state* is initialized with a callable object equivalent to the following lambda expression:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(see below) {
  auto&& [_, tuple] = std::forward<Sndr>(sndr);
  auto&& [sndr, ...sndrs] = std::forward<decltype(tuple)>(tuple);
  return sequence-state<
    Rcvr,
    decltype(sndr),
    remove_reference_t<decltype(sndrs)>>(
      rcvr,
      std::forward<decltype(sndr)>(sndr),
      std::forward<decltype(sndrs)>(sndrs)...);
}
```

The expression in the `noexcept` clause is equivalent to `noexcept(`*e*`)` where *e* is the expression evaluated by the `return` statement.

The member *impls-for*`<`*sequence-tag*`>::`*start* is initialized with a callable object equivalent to the following lambda expression:

```
[](auto& state, auto&) noexcept -> void {
  execution::start(get<0>(state.ops));
}
```

Open Design Questions

- Should `std::execution::sequence` lazily connect successive operations (status quo of this proposal) or eagerly connect operations when it is connected?
- Should `std::execution::sequence` be specified through recursive composition of a binary version thereof?

Implementation Experience

This algorithm is provided by Nvidia's stdexec [3].

References

[1] M. Dominiak et al. std::execution P2300R10
[2] https://github.com/facebookexperimental/libunifex/blob/effb7527401b32b5a2d82fdf6d1a8e8810cbdb07/doc/api_reference.md#sequencesender-predecessors-sender-last---sender
[3] https://github.com/NVIDIA/stdexec/blob/711da5971a8e8e940763c11bf6bbeb1c1bb22c3a/include/stdexec/__detail/__sequence.hpp
[4] B. Lelbach. C++ Asynchronous Parallel Algorithms P3300R0