`when_all` Oughtn't Hallucinate `set_stopped`

Document Number: P4269R1
Date: 2026-08-30
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes that `std::execution::when_all` not use a `std::inplace_stop_source` when it can be statically determined such a stop source would never be used.

Background

`std::execution::when_all` accepts one (soon to be zero [1]) or more senders and returns a sender which can be used to form asynchronous operations which completes when all child asynchronous operations complete, and with a value completion which is the concatenation of the value completions with which each child completed.

The above describes the happy case. In the unhappy case, wherein any child operation ends with `std::execution::set_error` or `::set_stopped`, `std::execution::when_all` requests that all other children stop, waits for them to end, and then passes on an appropriate unhappy completion.

To support the above the operation state of `std::execution::when_all` includes a `std::inplace_stop_source` (so that `std::inplace_stop_source::request_stop` can be called when any operation completes unhappily) stop tokens from which are injected into each child operation.

Discussion

Zero Child Senders

This scenario is not currently supported. When/if it is supported [1] there is no observable difference between an implementation of `std::execution::when_all` which does and does not create/use a `std::inplace_stop_source`, and therefore whether or not such creation and/or use happens is covered under quality of implementation ("QoI").

One Child Sender

When `std::execution::when_all` is provided with a single sender it generates a sender which behaves almost identically to the provided sender. The “almost” is necessary because despite the fact `std::execution::when_all` will never need to generate a stop request (because it only generates stop requests when a child completes unhappily, to induce the other children to stop, but since there’s only one child there are no other children) it still creates a `std::inplace_stop_source`, creates a stop callback which repeats stop requests from the stop token provided to `std::execution::when_all` (by requesting stop against the `std::inplace_stop_source` managed by `std::execution::when_all`), and injects the stop token from that `std::inplace_stop_source` into the sole child operation.

At least one senders/receivers implementation in the wild [2] works around/avoids the above by defining `when_all(s)` as being equivalent to `s`:

“[...] we do implement `when_all(s)` (the unary case) by coalescing to `s`.” —Ben Deane

Multiple Child Senders

When `std::execution::when_all` is provided with multiple senders it becomes possible that:

- A child operation will end unhappily, and
- Other operations will still be outstanding

In which scenario `std::execution::when_all` reacts by calling `std::inplace_stop_source::request_stop`. This means that a `std::inplace_stop_source` must be available for its use.

However one notes that the above scenario requires that a child operation end unhappily. If this is not possible then `std::execution::when_all` will never need to request that the other operations stop, and will therefore never need to call `std::inplace_stop_source::request_stop`, and will therefore not need access to a `std::inplace_stop_source`.

On first consideration it might seem that whether or not a `std::inplace_stop_source` is created in the above scenario is QoI, but this is not the case. Creating a `std::inplace_stop_source` and passing its stop tokens into child operations is an observable side effect (since child operations observe this through their receiver’s environment). This means that the above scenario not only leads to a `std::inplace_stop_source` being created and used for no reason, but also that it can lead to `std::execution::set_stopped_t()` completion signatures being generated where they’re never used, and where the user might expect otherwise.

Examples

Consider an lvalue-connectable sender `s`. `s` has the following completion signatures when connected in an environment with an unstoppable token:

```
std::execution::completion_signatures<
  std::execution::set_value_t()>
```

And the following completion signatures when connected in an environment with a stoppable token:

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_stopped_t()>
```

Under the status quo both `std::execution::when_all(s)` and `std::execution::when_all(s, s)` produce an lvalue-connectable sender with the following completion signatures when connected in an environment with an unstoppable token:

```
std::execution::completion_signatures<
  std::execution::set_value_t(),
  std::execution::set_stopped_t()>
```

A bizarre outcome since the second completion signature will never be used.

Uncertainty of Sender Response

Just because a sender emits `std::execution::set_stopped_t()` when connected in an environment with a stop token does not mean that it responds to stop requests delivered through said token. For example given:

- A sender `s`, and
- A stoppable token `t`

Consider (note [3] at 26:34):

```
std::execution::unstoppable(s | also_stopped_by(t))
```

The resulting sender advertises `std::execution::set_stopped_t()` and emits said completion in response to stop requests flowing via `t` but does not respond to stop requests from the stop token provided in the receiver's environment.

This is not the only example of a sender which belies naïve expectations vis-à-vis stop tokens. Consider:

```
s | std::execution::let_stopped([]() noexcept {
  return std::execution::just();
})
```

The resulting sender does not advertise `std::execution::set_stopped_t()`, but nonetheless it is responsive to stop requests.

Consider two properties of senders:

- Responds to stop requests, and
- Advertises `std::execution::set_stopped_t()`

The above examples show that the above properties are independent. That is: One cannot use the presence or absence of one of the above properties to infer the presence or absence of the other.

As pointed out to me by Lewis Baker the above is relevant vis-à-vis `std::execution::when_all`. Imagine a set of child senders provided to std::execution::when_all. Imagine that one of them is fallible. Now imagine that none of the others are responsive to stop requests. A `std::inplace_stop_source` is unnecessary since despite the fact one of the children can fail, this can't cause the others to stop.

Unfortunately, due to the above analysis, we cannot determine whether a sender responds to stop requests, because senders do not currently publish generically-consumable information about this property.

Synchronous Senders

It is perfectly acceptable for `std::execution`-based asynchronous operations to complete synchronously (i.e. inline with the call to `std::execution::start`). Lewis Baker points out that given a fallible sender s_0 which completes synchronously, and any other sender s_1, it is not necessary for `std::execution::when_all(`s_0`, `s_1`)` to use a stop source. This is because `std::execution::when_all` is specified to call `std::execution::start` on operation states in the order in which the associated sender was provided. As such the asynchronous operation associated with s_0 completes before the asynchronous operation associated with s_1 starts. Therefore if the asynchronous operation associated with s_0 fails the implementation of `std::execution::when_all` can simply neglect to start the asynchronous operation associated with s_1 (note that it is perfectly allowed to create an operation state, never start it, and allow its destructor to run [4]).

Note that despite the sound logic of the above it is not (at least currently) implementable. We don't have the ability to determine whether or not a sender's associated asynchronous operation is synchronous [5] (beyond running the operation, by which time it's too late to decide whether to compile the code to use a stop source).

Proposal

[exec.when.all]

The following wording assumes the application of P4217R1.

[...]

The expression `when_all(sndrs...)` is expression-equivalent to:

- *make-sender*`(when_all, {}, sndrs...)` if `sizeof...(sndrs)` is ~~not 0~~greater than `1`,
- `auto(sndrs...[0])` if `sizeof...(sndrs)` is `1`, or
- `just()` otherwise.

[...]

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

```
[]<class State, class Rcvr>(auto&&, State& state, const Rcvr& rcvr) noexcept
{
  if constexpr (State::uses-stop-source) {
    return make-when-all-env(state.stop-src, get_env(rcvr));
  } else {
    return get_env(rcvr);
  }
}
```

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

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(noexcept(e)) ->
  decltype(e)
{
  return e;
}
```

where *e* is the expression

`std::forward<Sndr>(sndr).apply(`*make-state*`<Rcvr>())`

and where *make-state* is the following exposition-only class template:

```
enum class disposition { started, error, stopped };        // exposition only

template<class Rcvr>
struct make-state {
  template<class... Sndrs>
  auto operator()(auto, auto, Sndrs&&... sndrs) const {
    using values_tuple = see below;
    using errors_variant = see below;
    using stop_callback = stop_callback_for_t<
      stop_token_of_t<env_of_t<Rcvr>>, on-stop-request>;
    static constexpr bool has-stop-source = see below;     // exposition only

    struct state-type {
      void arrive(Rcvr& rcvr) noexcept {                   // exposition only
        if (0 == --count) {
          complete(rcvr);
        }
      }

      void complete(Rcvr& rcvr) noexcept;                  // exposition only

      atomic<size_t> count{sizeof...(sndrs)};              // exposition only
      inplace_stop_source stop_src{};                      // exposition only
      atomic<disposition> disp{disposition::started};      // exposition only
      errors_variant errors{};                             // exposition only
      values_tuple values{};                               // exposition only
      optional<stop_callback> on_stop{nullopt};            // exposition only
    };

    return state-type{};
  }
};
```

Let *copy-fail* be `exception_ptr` if decay-copying any of the child senders' result datums can potentially throw; otherwise, *none-such*, where *none-such* is an unspecified empty class type.

The alias `values_tuple` denotes the type

```
tuple<value_types_of_t<Sndrs, FWD-ENV-T(env_of_t<Rcvr>), decayed-tuple,
  optional>...>
```

if that type is well-formed; otherwise, `tuple<>`.

The alias `errors_variant` denotes the type `variant<`*`none-such`*`, `*`copy-fail`*`, Es...>` with duplicate types removed, where `Es` is the pack of the decayed types of all the child senders' possible error result datums.

The static data member *`uses-stop-source`* is `true` if:

- `errors_variant` does not denote the type `variant<`*`none-such`*`>`, or
- There exists an element `S` of `Sndrs` such that `completion_signatures_of_t<S, env_of_t<Rcvr>>` contains `set_stopped_t()`

`false` otherwise.

The member `void` *`state-type`*`::`*`complete`*`(Rcvr& rcvr) noexcept` behaves as follows:

[...]

- Otherwise, evaluates:
  ```
  if constexpr (uses-stop-source && sends-stopped) {
    on_stop.reset();
    set_stopped(std::move(rcvr));
  }
  ```
  where *`sends-stopped`* equals `true` if and only if there exists an element `S` of `Sndrs` such that `completion_signatures_of_t<S, `*`when-all-env`*`<Env>>` contains `set_stopped_t()`.

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

```
[]<class State, class Rcvr, class... Ops>(
    State& state, Rcvr& rcvr, Ops&... ops) noexcept -> void {
  if constexpr (state.uses-stop-source) {
    state.on_stop.emplace(
      get_stop_token(get_env(rcvr)),
      on-stop-request{state.stop_src});
  }
  (start(ops), ...);
}
```

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

[...]

Implementation Experience

This paper has been implemented against nVidia's reference implementation of `std::execution` [6].

Revision History

R1

- Updated the wording to reflect the fact the unary version of `std::execution::when_all` decay-copies its argument (i.e. it doesn't return the argument by reference) [7]

Acknowledgments

The author would like to thank Lewis Baker for productive discussion regarding this problem.

The author would like to thank Maarten Arnst for noticing a wording issue in R0 of this paper [7].

References

[1] R. Leahy. when_all() is just just() P4217R1
[2] https://github.com/intel/cpp-baremetal-senders-and-receivers
[3] R. Leahy. Evolving C++ Networking With Senders & Receivers (Part 2) Core C++ 2024
[4] L. Baker et al. Eliminating heap-allocations in sender/receiver with connect()/start() as basis operations P2006R1
[5] M. Nadolski. A sender query for completion behavior P3206R0
[6] https://github.com/NVIDIA/stdexec/pull/2124
[7] https://github.com/NVIDIA/stdexec/issues/2241