Make `when_all` a Ronseal Algorithm

Document Number: P3887R0
Date: 2025-10-21
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes modifying `std::execution::when_all` such that it sends (and reports that it sends) stopped completion signals only when at least one of its child operations sends (and reports that it sends) such a signal.

Background

P2300 [1] adds the `std::execution::when_all` algorithm. Therein it describes the algorithm thusly (id. at §4.20.11):

"`when_all` returns a sender that completes once all of the input senders have completed."

Early Stop

In addition to a high level articulation of purpose (quoted above) P2300 has the following to say regarding the behavior of `std::execution::when_all` ([1] at §4.20.11):

"*[`when_all`] completes inline on the execution resource on which the last input sender completes, unless stop is requested before `when_all` is started, in which case it completes inline within the call to start.*"

libunifex, which is listed as "field experience" for P2300 (id. at §1.10.1), has an implementation of `when_all`. This implementation does not have the eager stop detection described above. It does however have late stop detection (i.e. if, upon the completion of all children, a stop has been requested the value(s) or error which would otherwise have been synthesized is replaced by a stopped completion signal).

Note that the above behavior with respect to early stop detection was not present in the original proposal for `std::execution::when_all` ([2] at §9.6.5.14). The original proposal specified the completion behavior of `std::execution::when_all` in terms only of the completions of the children thereof. The behavior described above was added in revision 3 of P2300 ([3] at §9.6.5.14). Revision 3 does not motivate the change except by saying (id. at §2.1):

"*Specify the cancellation scope of the `when_all` algorithm.*"

Note that a consequence of the modified/current behavior is that when early stop is detected the child operations of `std::execution::when_all` are not started (i.e. `std::execution::start` is not applied thereto). This is not the case for the libunifex implementation (i.e. it always starts all its children).

Also note that no part of the above is predicated on the associated stop token actually being capable of reporting a stop request, a property which P2300 goes out of its way to make detectable (see `std::unstoppable_token` ([1] at §33.3.3) and `std::never_stop_token` (id. at §33.3.7)).

Unreachable Stop

As with other facilities in P2300 `std::execution::when_all` is specified in terms of code. This has proven to be an extremely problematic specification strategy as irrelevant minutiae which happens to be a consequence of the code chosen is burned into the specification and irrevocably guaranteed, there is no gray area, no implementation latitude, only code whose every behavior must be exactly emulated in order for an implementation to be conforming.

The above-articulated issue arises with respect to `std::execution::when_all`'s stop handling.

The specification of `std::execution::when_all` ([1] at §4.20.11) specifies that *`state-type`* shall have a member `disp` of type *`disposition`*. *`disposition`* is an enumeration with three enumerators:

- *`started`*
- *`error`*
- *`stopped`*

With this given the completion logic is specified in terms of actions to take for each possible value of `disp`:

1. "*If `disp` is equal to* `disposition::started` *[...]*"
2. "*Otherwise, if `disp` is equal to* `disposition::error` *[...]*"
3. "*Otherwise [...]*"

Note that no enumerator of *`disposition`* is conditionally defined. Even if none of the children are capable of sending a stopped completion signal:

- The *`stopped`* enumerator is still present, and
- The logic indicated in the third step above must, arguably, still be emitted even though it is statically known to be unreachable

Note that the second bullet above is borne out by at least three implementations of `when_all` [4][5][6] including two which purport to be reference implementations of P2300 [4][5].

Consequences

Either of the two previously-described qualities mean that, even given a set of child senders none of which send `set_stopped_t()`, a `std::execution::when_all` sender will unconditionally report that it can send `set_stopped_t()`.

The former of the above-described qualities means that even if all child senders of `std::execution::when_all` are stop token unaware (i.e. they do not stop in response to stop requests (for a motivating example see [7] at §3.2) `std::execution::when_all` will check for a stop request and honor it (immediately before starting the child operations, but at no other point), perhaps belying the reasonable expectations of the consumer.

Discussion

Library Issue

The fact that `std::execution::when_all` arguably requires the receiver to support stopped completion signals even when it is statically known one can never be emitted would arguably be a defect were it not for the fact `std::execution::when_all` unconditionally sends stopped due to the fact it enriches the child operations with early stop request checking.

If early stop request checking is removed the requirement that the downstream receiver support the emission of a stopped completion signal even when it is statically known never to be emitted (because none of the children emit it) should be unambiguously removed through a wording tweak.

Single Responsibility Principle

The first quote from P2300 in the background section articulates a coherent single responsibility for `std::execution::when_all`: An operation which completes once all its children complete. This articulation, and the responsibility which springs therefrom, have nothing to do with eager checking for stop requests, functionality which can easily be added by a separate algorithm ([8] at 26:55-28:22).

Appeals to the single responsibility principle, however, can fall on deaf ears among “pragmatists.” Concerns buttressed by the platonic principles of software engineering are secondary to “real world use cases” and the convenience imagined to be wrought by wanton coupling.

Fortunately we do not need to anchor our critique of the current formulation of `std::execution::when_all` solely in the platonic domain. Given that "*all the sender/receiver features are language features being implemented in library*" ([9] at §3) and given that C++ supports mandatory, orderly, well-defined clean up (i.e. RAII) we need only to find examples of sender algorithms which constitute such clean up and which must be run (and not skipped due to a stop request detected early) for program correctness.

For the aforementioned we need not look far. Async scopes must be joined for correctness ([10] at §5.6.5 and §5.7.6) and generalized async RAII has been proposed [9]. Both of these have at their core sender algorithms. Due to the reasons given above these algorithms will be unusable with `std::execution::when_all` thereby belying one of the motivating principles of `std::execution` ([1] at §1.2):

"Care about all reasonable use cases, domains, and platforms."

Principle of Least Surprise

The above-discussed quote not only lends itself to analysis through the lens of the single responsibility principle, but also through the lens of the principle of least surprise. Upon seeing an algorithm called `std::execution::when_all` users might imagine that it completes **when all** child operations complete (i.e. precisely the articulation from the previously-mentioned quote). Doing anything more (as the current specification demands) is astonishing because `std::execution::when_all` does more than what it says on the tin (i.e. it's not a "Ronseal [11] algorithm").

Were it the case that the additional behavior is advantageous and/or benign the above might not be an issue. Users either wouldn't notice or wouldn't be astonished by a slight or benign divergence from their reasonable expectation. Unfortunately this is not the case for the reasons laid out in the preceding section: When the behavior of `std::execution::when_all` diverges from users' reasonable expectations it is precisely because it introduces subtle incorrect behavior. `set_stopped_t()` appears as a completion signature where it wasn't expected (because no child reports that completion signature). Operations which the user reasonably believed would be started and allowed to run are skipped.

There is a narrower, less astonishing, Ronseal algorithm inside `std::execution::when_all`. The version which appears when the red herring behavior with respect to stop requests is removed (and perhaps later re-added as a separate algorithm as discussed in the preceding section). C++26 should ship that, not what's currently in the working draft.

Proposal

[exec.when.all]

- Otherwise, evaluates:
  ```
  if constexpr (sends-stopped) {
    on_stop.reset();
    set_stopped(std::move(rcvr));
  } else {
    unreachable();
  }
  ```
  where *sends-stopped* equals `true` if and only if `completion_signatures_of_t<Sndrs, when-all-env<Env>>` contains `set_stopped_t()` for any `Sndrs`.

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 {
  state.on_stop.emplace(
    get_stop_token(get_env(rcvr)),
    on-stop-request{state.stop_src});
~~  if (state.stop_src.stop_requested()) {~~
~~    state.on_stop.reset();~~
~~    set_stopped(std::move(rcvr));~~
~~  } else {~~
  ~~(~~start(ops), ...);
~~}~~
}
```

Implementation Experience

This paper has been implemented against nVidia's reference implementation of `std::execution` [12].

Acknowledgements

The author would like to thank Bryan St. Amour for wording assistance.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] M. Dominiak et al. std::execution P2300R2
[3] M. Dominiak et al. std::execution P2300R3
[4] https://github.com/NVIDIA/stdexec/blob/138e136fa4b93e7e096a4968eaac1b0c94f0d255/include/stdexec/__detail/__when_all.hpp#L210-L228
[5] https://github.com/bemanproject/execution/blob/c587808532d26c47a3e2edebc810720224cbffe7/include/beman/execution/detail/when_all.hpp#L126-L157
[6] https://github.com/facebookexperimental/libunifex/blob/main/include/unifex/when_all.hpp#L249-L257
[7] E. Niebler. write_env and unstoppable Sender Adaptors P3284R4
[8] R. Leahy. Evolving C++ Networking with Senders & Receivers (Part 2). Core C++ 2024
[9] K. Shoop. async-object - aka async-RAII P2894R0
[10] I. Petersen et al. async_scope - Creating scopes for non-sequential concurrency P3149
[11] https://www.ronseal.com/
[12] https://github.com/NVIDIA/stdexec/pull/1666