`unless_stop_requested`

Document Number: P3892R0
Date: 2025-10-27
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes adding an algorithm which neglects to start an asynchronous operation if at the time it would've been started a stop request therefor is already outstanding.

Background

P3887 [1] proposes modifying `std::execution::when_all` to:

- Remove early stop detection, and
- Make its `set_stopped_t()` completion signature conditional on any of its children exposing that same completion signature

One of the justifications presented by that paper is that (ibid., emphasis added):

*"[T]he responsibility [of `std::execution::when_all`], ha[s] nothing to do with eager checking for stop requests, **functionality which can easily be added by a separate algorithm**."*

For support of this claim it references a section of a conference talk ([2] at 26:55-28:22) given by the author himself (also the author of this paper) wherein use of the "separate algorithm" alluded to above is demonstrated (therein called `if_not_stopped`). Notably no implementation of that algorithm is shown or otherwise provided.

Discussion

Stop Requests

The stop mechanism in `std::execution` is built on top of the machinery added to support `std::jthread` [3]: Stop sources, tokens, and requests. Naming is certainly not something LEWG takes lightly and therefore it stands to reason that we can read into these precise terms, particularly "requests." When something is "requested" it may be denied and verily P0660 alludes to this (emphasis added):

"*[T]he use of the term 'request' [...] clearly communicate[s] the asynchronous and* ***cooperative*** *nature of the abstraction.*"

The fact that stop request handling is "cooperative" suggests that a stop request may have no effect: The requested party may elect not to cooperate. This interpretation is bolstered when one conceptualizes the result of handling a stop request as "serendipitous-success" [4]. If actioning a request to stop induces serendipitous-success why would the recipient of such a request prefer serendipitous-success to what we might call "true success" or even failure? These lattermost modalities at least convey information about what happened, and if available shouldn't we prefer the modality which conveys more information?

One can imagine a use case for exactly this behavior: When a timeout expires a stop may be requested (to induce the operation being timed out to promptly end, a behavior which is critical for the "receiver contract" ([5] at §1.5.1) which lies at the center of the structured concurrency of `std::execution`). Now let us imagine at that exact instant true success or failure occurs. Should the operation being timed out end with serendipitous-success, or should it prefer to yield the true result of the operation the consumer was actually waiting on?

While the above-described logic makes sense in isolation as one considers ever larger, composed parts of a system it becomes problematic. Larger components are liable to contain possibly-unbounded loops and if the components thereof are always eagerly ready to complete (and therefore never check for stop requests in accordance with the above) the emergent property of application of the above principle is an operation which, in unfortunate circumstances, is unstoppable.

This caveat might lead one to believe that stop checking should be built in at the framework level, and that P3887 is misguided therefore in removing it from a foundational algorithm such as `std::execution::when_all`. But `std::execution::when_all` is not a looping construct and therefore does not fall under the analysis laid out above. Instead we would expect to see early stop handling in looping constructs (none of which have currently been standardized).

When we survey such constructs (as written or considered outside the standard), however, we find they do not feature eager stop checking [6][7][8]. Instead one of the things we see is eager stop checking injected at a well-chosen point via the algorithm proposed by this paper ([2] at 26:55-28:22).

Even if one still wanted to argue for pervasive, framework-level eager stop checking there would also be the performance cost to reckon with. Eagerly checking for stop has a cost, and if this were pervasive that cost would be linear in the number of composed components. This would discourage users from composing larger parts from smaller ones, exactly the sort of problem the design of the STL aims to avoid (and one which is elsewhere being addressed in the context of `std::execution` [9]).

Naming

When the author first implemented the proposed algorithm he called it `if_not_stopped` ([2] at 27:57-28:23). The following names have been considered:

- `if_not_stopped` (the original name)
- `when_not_stopped`
- `upon_not_stopped`
- `unless_stopped`
- `unless_stop_requested` (the name proposed by this paper)

The above names all consist of two parts which will be discussed separately below.

Logical Operator

The name aims to describe the condition under which the child operation is started (see below) but in order to do that it must establish the truth value of that condition. This involves naming the logical operator to use.

From the plain language description of the algorithm it is clear that we want to neglect to start the child operation when a stop request is outstanding. Put differently we want to start the child operation when a stop request is not outstanding. We must figure out how to express this negation in the name of the algorithm.

Of the options considered above the following names for this negation emerge, with respective arguments:

- `if_not`: This is plain and to the point but reuses two keywords from the language connected by an underscore, also it is two words connected by an underscore rather than just one word
- `when_not`: This reuses “when” which already has use in `std::execution::when_all` (and the not-yet-standardized counterpart `when_any`)
- `upon_not`: This reuses “upon” which already has use in `std::execution::upon_error` and `::upon_stopped`
- `unless`: “Unless” unambiguously expresses “if not” without being two words

As such this paper proposes `unless`.

Condition

In addition to the logical operation applied to the condition (see above) the name of the algorithm must also express the condition under which the operation will be started (or not started).

The author’s first attempt to name the algorithm (`if_not_stopped`) made use of `stopped`. However it was pointed out (by Lewis Baker) that “stopped” describes a completion signal (i.e.

`set_stopped`) rather than a condition. The condition is best articulated as `stop_requested` since that is the name of the member function required by the `stoppable_token` concept ([10] at §33.3.3) (models of which are used to communicate the stop requests being discussed by this algorithm's name).

As such this paper proposes the name `unless_stop_requested`.

Proposal

[execution.syn]

```
struct spawn_future_t { unspecified };
struct unless_stop_requested_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 unless_stop_requested_t unless_stop_requested{};

// [exec.cmplsig], completion signatures
```

[exec.unless.stop.requested]

Add the above-named section with contents as follows:

`unless_stop_requested` adapts a single input sender into a sender which starts the operation represented by the input sender if and only if a stop has not been requested at the time at which the composed operation is started.

The name `unless_stop_requested` denotes a customization point object. For subexpression `sndr`, if `decltype((sndr))` does not satisfy `sender`, `unless_stop_requested(sndr)` is ill-formed.

Otherwise, the expression `unless_stop_requested(sndr)` is expression-equivalent to:

```
transform_sender(
      get-domain-early(sndr),
      make-sender(unless_stop_requested, {}, sndr))
```

Except that `sndr` is only evaluated once.

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

```
namespace std::execution {
  template<>
  struct impls-for<unless_stop_requested_t> : default-impls {
    static constexpr auto start = see below;
  };
}
```

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

```
  []<class State, class Rcvr, class Op>(
          State&, Rcvr& rcvr, Op& op) noexcept -> void
  {
    if constexpr (
          !unstoppable_token<stop_token_of_t<env_of_t<Rcvr>>>)
    {
          if (get_stop_token(get_env(rcvr)).stop_requested()) {
            set_stopped(std::move(rcvr));
            return;
          }
    }
```

```
            start(op);
        }
```

Implementation Experience

This paper has been implemented against nVidia's reference implementation of `std::execution` [11].

Acknowledgements

The author would like to acknowledge Ville Voutilainen for encouraging him to write this paper and to thank:

- Ville Voutilainen and Lewis Baker for naming input,
- Ian Petersen for code review of the implementation, and
- Bryan St. Amour for wording review

References

[1] R. Leahy. Make when_all a Ronseal Algorithm P3887R0
[2] R. Leahy. Evolving C++ Networking with Senders & Receivers (Part 2). Core C++ 2024
[3] N. Josuttis et al. Stop Token and Joining Thread P0660R10
[4] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[5] J. Hoberock et al. A Unified Executors Proposal for C++ P0443R14
[6] https://github.com/facebookexperimental/libunifex/blob/b6bedebc4d87eda5e31b364585a84576013eae67/include/unifex/repeat_effect_until.hpp
[7] https://github.com/NVIDIA/stdexec/blob/138e136fa4b93e7e096a4968eaac1b0c94f0d255/include/exec/repeat_effect_until.hpp
[8] R. Leahy. Extending std::execution Further: Higher-Order Senders and the Shape of Asynchronous Programs. C++ on Sea 2025
[9] L. Baker. Reducing operation-state sizes for subobject child operations P3425R1
[10] M. Dominiak et al. std::execution P2300R10
[11] https://github.com/NVIDIA/stdexec/pull/1667