Why Did You Stop?

Document Number: P4373R0
Date: 2026-09-10
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes that, in certain cases, consumers be able to determine that asynchronous operations only send `std::execution::set_stopped_t()` when requested.

Background

Asynchronous operations in C++ (i.e. implemented using `std::execution`) can report completions using three different “dispositions:”

- Value (i.e. success)
- Error (i.e. failure)
- Stopped

The last of these may also be referred to as:

- Canceled
- Serendipitous-success [1]

Regardless of how one refers to the disposition, stopped completions are meant to indicate that the operation detected that its continued forward progress was unnecessary, and therefore it ended. Note this is distinct from error completions wherein the operation’s forward progress is necessary, but it cannot continue due to some impediment.

“[D]etected that its continued forward progress was unnecessary” is useful for describing the situations in which stopped completions are sent, but it isn’t very specific. How might an operation perform this detection?

Overwhelmingly this determination is made via stop tokens. Either by polling them as to whether or not a stop is requested (by calling their `stop_requested` member function), or by subscribing for an eager notification when a stop is requested (by constructing an instance of a type obtained via `std::execution::stop_callback_for_t`).

Operations are provided with a stop token via their environment. Particularly an environment is queried as to the stop token it provides via the `std::get_stop_token` query.

Discussion

One might expect that operations which are capable of stopping (i.e. which advertise a `set_stopped_t()` completion via `get_completion_signatures` and `completion_signatures_of_t`) only stop in response to a stop request (where “stop request” is understood to be an impulse transmitted via stop token provided via the receiver’s environment). This is false. Asynchronous operations can send any completion in response to any stimulus whatsoever.

This, in general, is not an oversight. Consider an asynchronous operation which reads from a signalfd. It reads a `signalfd_siginfo` structure which indicates `SIGINT` was received. The requirement is that when this signal is received the application stop. It would be perfectly reasonable therefore for the operation reading from the signalfd to realize this completion as `set_stopped_t()`. Note that while this is arguably in response to a stop request the request is from the user, realized as a `SIGINT`, rather than via a stop token.

The takeaway from the above is that, in general, just because a sender advertises a `set_stopped_t()` completion does not mean one can assume that:

- It only sends such a completion when a stop request is delivered via the associated stop token, or
- That it stops in response to such requests at all

Which would be a fine state of affairs if neither of these pieces of information were actionable. But this is false. The first bullet, particularly, provides an opportunity for a better/improved `std::execution::when_all`.

Modulo decay-copying [2] the completions of a `std::execution::when_all` sender are:

- A single value completion which is the concatenation of the value completions of all children (note that `std::execution::when_all` only admits children with a single value completion), and
- The error and stopped completions of all children

When any child completes in error or with stopped `std::execution::when_all` emits a stop request into all other children. The fact that a single child didn’t complete successfully renders the completion *fait accompli* and therefore their “forward progress [is] unnecessary.”

This leads to a chicken and egg problem [3]. `std::execution::when_all` advertises `set_stopped_t()` when any of its children advertise such a completion, but:

- The child may only advertise `set_stopped_t()` because `std::execution::when_all` injects a stop token thereinto, and

- The child may only send `set_stopped_t()` in response to a stop request emitted by `std::execution::when_all` because another child completed in error or with stopped

Consider a concrete example. Imagine a multishot sender `sndr` with respect to two environments:

- `std::execution::env<>` and
- `std::execution::env<std::execution::prop<std::get_stop_token_t, std::inplace_stop_token>>`

The sender advertises the following completion signatures in both environments:

- `std::execution::set_value_t()`
- `std::execution::set_error_t(std::exception_ptr)`

And in the latter environment, which has a stop token, the sender additionally advertises `std::execution::set_stopped_t()`. Moreover it only sends this completion in response to a stop requested through the `std::inplace_stop_token`.

Now consider `std::execution::when_all(sndr, sndr)`. It advertises the following completion signatures in accordance with the formulation above:

- `std::execution::set_value_t()`
- `std::execution::set_error_t(std::exception_ptr)`
- `std::execution::set_stopped_t()`

And finally consider an operation formed by connecting the above-described sender in an environment whose type is `std::execution::env<>`. That operation has these outcomes:

- Both children succeed thereby causing the composed operation to send `std::execution::set_value_t()`
- One child completes, and then the other fails, thereby causing the composed operation to send `std::execution::set_error_t(std::exception_ptr)`
- One child fails, stop is requested, and the other stops in response thereto, thereby causing the composed operation to send `std::execution::set_error_t(std::exception_ptr)`
- One child fails, stop is requested, and the other racily succeeds, thereby causing the composed operation to send `std::execution::set_error_t(std::exception_ptr)`
- One child fails, stop is requested, and the other racily also fails, thereby causing the composed operation to send `std::execution::set_error_t(std::exception_ptr)`

Put differently: Absent a stop request each child either succeeds or fails. Because there's no stop token in the environment `std::execution::when_all` receives stop requests will only be emitted when a child fails. Once a child fails the result of `std::execution::when_all` is determined (i.e. it is that failure). Therefore despite the fact children can end with `std::execution::set_stopped_t()` that completion will never be forwarded through

`std::execution::when_all` due to the causal relationship between that completion, the emission of a stop request, and the completion of the other child.

Despite the above `std::execution::when_all` advertises `std::execution::set_stopped_t()`. The reason for this is that while `std::execution` allows a consumer to observe whether or not an operation emits `std::execution::set_stopped_t()` it doesn't permit them to determine why that emission occurs. Therefore `std::execution::when_all` is implemented as if `std::execution::set_stopped_t()` can be emitted unprovoked by a stop request, which we, possessing non-generic information about its children, know to be impossible.

However there's one piece of information about the above scenario that has heretofore not been discussed: The fact that the children don't send `std::execution::set_stopped_t()` when connected in an environment without a stop token. If they sent `std::execution::set_stopped_t()` absent the provocation of a stop request it stands to reason they wouldn't start announcing said completion only when a stop token is provided, that stop token must be causal.

Despite the fact the above logic is sound, the standard doesn't enforce it, therefore user-provided senders are permitted to be arbitrarily insane in this regard, advertising `std::execution::set_stopped_t()` completions only when their environment is enriched with a stop token, but then sending said completions unprovoked by that token.

Proposal

Insert the following wording somewhere in [exec]:

Two queryable objects `env1` and `env2` are *stop-token-equivalent* if, for every query object `q` other than `get_stop_token` and every pack of arguments `args`, the expressions

```
q(env1, args...)
```

and

```
q(env2, args...)
```

are expression-equivalent.

Let `env1` and `env2` be stop-token-equivalent queryable objects of types `Env1` and `Env2`, respectively, for which

- `unstoppable_token<stop_token_of_t<Env1>>` is `true`, and
- `unstoppable_token<stop_token_of_t<Env2>>` is `false`.

Let `Sndr` be a sender type for which `sender_in<Sndr, Env1>` and `sender_in<Sndr, Env2>` are both `true`. If `set_stopped` is not a permissible completion for `Sndr` and `Env1` but is a permissible completion for `Sndr` and `Env2`, then the behavior of an asynchronous operation created by connecting a sender of type `Sndr` to a receiver `rcvr` whose environment has type `Env2` is undefined if it invokes `set_stopped(rcvr)` when

```
get_stop_token(get_env(rcvr)).stop_requested()
```

is `false`.

[*Note*: This allows algorithms to determine that an operation sends a stopped completion only when stop is requested. —*end note*]

References

[1] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[2] R. Leahy. Stop the Decay P4288R1
[3] R. Leahy. when_all Oughtn't Hallucinate set_stopped P4269R1