

stopped_by

Document Number: P4390R0

Date: 2026-10-18

Reply-to: Robert Leahy <rleahy@rleahy.ca>

Audience: SG1, LEWG

Abstract

This paper proposes `std::execution::stopped_by` which allows asynchronous operations to be stopped, in response to a stop request, by a second asynchronous operation.

Background

While C++26 brings us `std::execution` the history of asynchronous operations does not begin or end there. Not all software is written in C++, and therefore we're left not only with a history of C-style asynchronous APIs but also a future thereof.

The sort of C-style asynchronous APIs mentioned above commonly make use of an opaque pointer (a “closure”) to identify an operation, for example:

- `io_uring_cqe::user_data` and `io_uring_sqe::user_data`
- `epoll_event::data`
- `OVERLAPPED*`
- `uv_req_t::data`

This list is far from exhaustive.

Operation states and the “receiver contract” allow `std::execution`-based adaptations of C-style asynchronous operations to provision a “closure” pointer without allocation: A pointer to, or into, the operation state.

`std::execution`, however, is not merely concerned with initiating and completing asynchronous operations, but also with allowing them to be stopped. A stable pointer to, or into, the operation state (which `std::execution` trivially provides per above) is insufficient to bring stop support to adaptations of C-style APIs.

Unlike straightforward, stop-request-unaware adaptations of C-style asynchronous APIs (which can still benefit from abstraction [1]) handling stop requests is infamously fraught. Not only are there the races most people are casually aware of, but `std::execution` brings another dimension: Neglecting to emit stop-request-handling machinery when the operation's stop token satisfies `std::unstoppable_token`.

Motivation

Consider `ares_getaddrinfo`, an asynchronous operation provided by `libcares`, which is a C library:

```
typedef void (*ares_addrinfo_callback)(void *arg, int status,
                                       int timeouts,
                                       struct ares_addrinfo *result);

void ares_getaddrinfo(ares_channel_t *channel, const char *name,
                     const char* service,
                     const struct ares_addrinfo_hints *hints,
                     ares_addrinfo_callback callback,
                     void *arg);
```

Note that this is exactly the sort of operation described above: It is provided by a C-style API, and uses a closure pointer to identify operations.

`libcares` operations can be stopped by:

```
void ares_cancel(ares_channel_t *channel);
```

Note that this function is parameterized only by the channel, which means it stops all outstanding operations thereagainst, a state of affairs which one of the maintainers of `libcares` characterizes as “a bit absurd” [13]. As such this example supposes the existence of:

```
void ares_cancel_by_arg(ares_channel_t *channel, void *arg);
```

Which the author of this paper has implemented [4] but which has not yet been merged because it is blocked on the adoption of another pending merge [9].

Now consider the task of bringing this operation within the penumbra of `std::execution` so it can be consumed from idiomatic C++26 code:

```
struct ares_getaddrinfo_sender {
    using sender_concept = std::execution::sender_tag;

    ares_channel_t* channel;
    const char* name;
    const char* service;
    const ares_addrinfo_hints* hints;

    template<class Self, class Env>
    static constexpr auto get_completion_signatures() {
```

```

    return std::execution::completion_signatures<
        std::execution::set_value_t(ares_addrinfo*),
        std::execution::set_error_t(int),
        std::execution::set_stopped_t()>{};
}

template<class Rcvr>
struct operation {
    using operation_state_concept = std::execution::operation_state_tag;

    using token_t = std::execution::stop_token_of_t<
        std::execution::env_of_t<Rcvr>>;

    ares_channel_t* channel;
    const char* name;
    const char* service;
    const ares_addrinfo_hints* hints;
    Rcvr rcvr;
    std::atomic<unsigned> outstanding{2};

    struct on_stop {
        operation& self;

        void operator()() && noexcept {
            if (self.outstanding.fetch_add(1, std::memory_order_relaxed) == 0) {
                return;
            }
            ::ares_cancel_by_arg(self.channel, &self);
            self.maybe_complete();
        }
    };

};

using callback_t = std::stop_callback_for_t<token_t, on_stop>;

std::optional<callback_t> stop_callback;
std::variant<
    std::monostate,
    ares_addrinfo*,
    int,
    std::execution::set_stopped_t> completion;

void deliver() noexcept {
    std::visit([this](auto&& result) noexcept {

```

```

        using T = std::remove_cvref_t<decltype(result)>;
        if constexpr (std::same_as<T, ares_addrinfo*>) {
            std::execution::set_value(std::move(rcvr), std::move(result));
        } else if constexpr (std::same_as<T, int>) {
            std::execution::set_error(std::move(rcvr), std::move(result));
        } else if constexpr (std::same_as<T, std::execution::set_stopped_t>)
        {
            std::execution::set_stopped(std::move(rcvr));
        }
    }, completion);
}

```

```

void maybe_complete() noexcept {
    if (outstanding.fetch_sub(1, std::memory_order_release) != 1) {
        return;
    }
    std::atomic_thread_fence(std::memory_order_acquire);
    stop_callback.reset();
    deliver();
}

```

```

void complete(int status, ares_addrinfo* result) noexcept {
    if (status == ARES_SUCCESS) {
        completion.template emplace<1>(result);
    } else if (status == ARES_ECANCELLED) {
        completion.template emplace<3>(std::execution::set_stopped);
    } else {
        completion.template emplace<2>(status);
    }
    maybe_complete();
}

```

```

void start() & noexcept {
    ::ares_getaddrinfo(
        channel, name, service, hints,
        [] (void* arg, int status, int, ares_addrinfo* result) noexcept {
            auto& self = *static_cast<operation*>(arg);
            self.complete(status, result);
        },
        this);
}

```

```

stop_callback.emplace(
    std::execution::get_stop_token(std::execution::get_env(rcvr)),

```

```

        on_stop{*this});
        maybe_complete();
    }
};

template<class Rcvr>
auto connect(Rcvr rcvr) const noexcept {
    return operation<Rcvr>{channel, name, service, hints, std::move(rcvr)};
}
};

std::execution::sender auto ares_getaddrinfo(
    ::ares_channel_t* const channel,
    const char* const name,
    const char* const service,
    const struct ::ares_addrinfo_hints* const hints) noexcept
{
    return ares_getaddrinfo_sender{channel, name, service, hints};
}

```

Note the large amount of boilerplate. This boilerplate can be divided into two broad categories:

- Sender and operation state scaffolding, which is addressed by `started_by`, which is proposed in a different paper [1]
- Stop request support, which is the addressed by `stopped_by` and is therefore the concern of this paper

Focusing on the boilerplate which integrates stop request support, since that's the remit of this paper, note the complexity of the boilerplate: Carefully-chosen atomic operations, template metaprogramming to extract the stop token type and determine its corresponding stop callback type, careful construction of the stop callback at initiation, and corresponding destruction at completion.

Not only is the boilerplate discussed above not specific to adapting `ares_getaddrinfo` to `std::execution`, but due to its complexity it's difficult to write, especially correctly.

Due to the above the stop request boilerplate should be extracted into a separate algorithm wherein the complexity discussed above can be contained and concentrated: `stopped_by`, which is proposed by this paper.

To bolster the above if one examines the specification-as-code proposed by this paper one will notice there's an immense amount of overlap with the code in the example above.

Design

`stopped_by` is a pipeable sender adaptor object which supports the following signatures:

```
std::execution::sender auto stopped_by(std::execution::sender auto&& sndr,  
                                       auto&& request_stop,  
                                       auto&& classifier);
```

```
std::execution::sender auto stopped_by(std::execution::sender auto&& sndr,  
                                       auto&& request_stop);
```

The first parameter, the child sender, is the “main operation.” This represents the operation to which `stopped_by` is adding stop request handling.

The second parameter is a sender factory. The sender returned by this factory represents an asynchronous operation which stops the main operation. This factory receives as a parameter a mutable, lvalue reference to the operation state of the main operation. This allows it to identify and communicate with the operation it is stopping.

The third parameter, if present, is a “classifier,” a predicate which determines whether or not a certain completion emitted by the main operation indicates that the operation was stopped and should therefore be coalesced to `std::execution::set_stopped_t()`.

If, when a `stopped_by` sender is connected, the receiver’s environment yields a stop token which satisfies `std::unstoppable_token`, then the child sender is connected directly thereby eliminating all stop request handling machinery.

Example

An implementation of the earlier wrapper using `started_by` [1] and `stopped_by` follows:

```
struct classifier {  
    constexpr bool operator()(std::execution::set_value_t,  
                              struct ::ares_addrinfo*) && noexcept  
    {  
        return false;  
    }  
    constexpr bool operator()(std::execution::set_error_t,  
                              const int status) && noexcept  
    {  
        return status == ARES_ECANCELLED;  
    }  
};
```

```

std::execution::sender auto ares_getaddrinfo_impl(
    ::ares_channel_t* const channel,
    const char* const name,
    const char* const service,
    const struct ::ares_addrinfo_hints* const hints)
{
    return started_by< [1]
        std::execution::completion_signatures<
            std::execution::set_value_t(struct ::ares_addrinfo*),
            std::execution::set_error_t(int)>>(
                [channel, name, service, hints]<typename Op>(Op& op) noexcept {
                    ::ares_getaddrinfo(
                        channel, name, service, hints,
                        [](void* arg, int status, int, ares_addrinfo* result) noexcept {
                            auto& op = *static_cast<Op*>(arg);
                            if (status == ARES_SUCCESS) {
                                op.complete(std::execution::set_value, std::move(result));
                            } else {
                                op.complete(std::execution::set_error, std::move(status));
                            }
                        },
                        std::addressof(op));
                });
}

```

```

std::execution::sender auto ares_getaddrinfo(
    ::ares_channel_t* const channel,
    const char* const name,
    const char* const service,
    const struct ::ares_addrinfo_hints* const hints)
{
    return stopped_by(
        ares_getaddrinfo_impl(channel, name, service, hints),
        [channel](std::execution::operation_state auto& op) noexcept {
            return
                std::execution::just() |
                std::execution::then([channel, &op]() noexcept {
                    ::ares_cancel_by_arg(channel, std::addressof(op));
                });
        },
    );
}

```

```
    classifier{}));  
}
```

Note the reduction in boilerplate. The vast majority of lines of the above example are not particular to the task of adapting `ares_getaddrinfo` to `std::execution`.

Also note that if one were repeatedly wrapping many different `libcares` asynchronous operations one could extract the `stopped_by` machinery into a common wrapper which could be shared across all such operations: Once stop support is realized for a single operation from that library, it can be reused across all of them.

Lastly note that this implementation is superior to the original, verbose, complex example: Per the design laid out above it eliminates the overhead of stop request-handling machinery (including advertisement of `std::execution::set_stopped_t()` when `std::unstoppable_token` is satisfied by the stop token provided by the receivers environment.

Discussion

Lineage

`std::execution::stopped_by` is the result of decomposing an algorithm called `externally_completed` which the author has presented in several conference talks [2][3][4][5].

The other half of the above-mentioned decomposition is left to a different paper [1].

Stop Requests vs. Cancellation

Many C-style asynchronous APIs provide a primitive for “cancellation.” In C++ however we speak of “stop requests.” There’s a valuable distinction between how these are typically understood: Cancellation is usually understood as attempting to induce an operation to exhibit a certain, erroneous outcome (e.g. `ECANCELED`). Requesting stop, on the other hand, is simply an indication to the operation that a degraded version of success [6] (i.e.

`std::execution::set_stopped_t()` is acceptable, and that the operation should end as soon as possible.

The above may at first seem pedantic but it informs the design of `stopped_by`: When a stop is requested, and the main operation completes in some way other than `std::execution::set_stopped_t()`, no attempt is made to coalesce that completion thereto. The fact that the main operation has completed is sufficient.

In addition to the obvious arguments regarding raciness we can arrive at the correctness of the above differently: Imagine `stopped_by` did attempt the coalescing rejected above, and imagine one was attempting to enrich `IORING_OP_OPENAT` with stop support. There would then exist the

possibility that `IORING_OP_OPENAT` succeeds (and creates a file description which ought to be closed), and that `stopped_by` coalesces that successful completion to `std::execution::set_stopped_t()`. This would leak a file descriptor [3] and is therefore plainly undesirable.

Note that the above concern is concrete: `ares_getaddrinfo`, used as an example above, generates an owning pointer upon successful completion. Therefore wantonly discarding value completions that happen to arrive after stop is requested would result in a memory leak.

Closure Sharing

Enriching an already-running asynchronous operation with stop request support implies that the already-running asynchronous operation has an identity. An abstraction for adding such stop request support must therefore include some way to identify the operation being cancelled.

C-style APIs provide prior art in the form of a closure pointer. For example `CancelIoEx` accepts an `OVERLAPPED*` identifying the operation whose cancellation is requested. Above this closure pointer was discussed vis-à-vis `std::execution` with the conclusion that, given the “receiver contract,” the operation state can be used to easily provide a closure pointer. Thinking about this further, however, something else becomes clear: An operation’s identity is its operation state.

From the above a shape by which a generic algorithm can request stop against another, already-running operation emerges:

```
void request_stop(std::execution::operation_state auto& target);
```

This gives us an idea of the shape of `stopped_by`:

```
std::execution::sender auto stopped_by(  
    std::execution::sender auto&& child,  
    auto&& request_stop);
```

Synchronous vs. Asynchronous

The shape established by the preceding section has a limitation: Emitting the stop request must be synchronous. This is a problematic assumption given that even constructors and destructors have cause to be asynchronous [7]. To make this concrete: `IORING_OP_ASYNC_CANCEL` emits a stop request, and is fundamentally asynchronous.

We therefore arrive at the final shape:

```
std::execution::sender auto request_stop(  
    std::execution::operation_state auto& target);
```

I.e. the injected customization point is a sender factory, rather than a regular, synchronous function.

Completion Signatures of Requesting Stop

Above, before the introduction of an asynchronous shape for requesting stop, the following synchronous function signature was given:

```
void request_stop(std::execution::operation_state auto& target);
```

Note that this signature not only specifies that the operation returns no value, but also that it may throw an exception. This suggests that the asynchronous version might have completion signatures as follows [8]:

```
std::execution::completion_signatures<  
    std::execution::set_value_t(),  
    std::execution::set_error_t(std::exception_ptr)>
```

However this raises an important question: How should `stopped_by` react to failures to request stop?

To begin exploring this question we consider the shape of `stopped_by`, and its ramifications for the asynchronous operation its sender creates:

```
std::execution::sender auto stopped_by(  
    std::execution::sender auto&& child,  
    auto&& request_stop);
```

This shape suggests that `stopped_by`'s asynchronous operation:

1. Starts the asynchronous operation resulting from `child`
2. Installs a stop callback against the environment's stop token
3. Upon invocation of above stop callback connects and starts the sender returned by `request_stop` invoked with the operation state of the operation resulting from `child`

This suggests that when the stop request operation fails the overall operation is in one of two situations:

- The child operation has already completed, or
- The child operation is still executing

In the former case failure to request stop can be ignored: The intention was for the operation to end as soon as possible, and it has already ended.

We therefore examine the latter case. A naïve approach would be to immediately end the overall operation with `std::execution::set_error_t(std::exception_ptr)`. This has a

problem: Ending the overall asynchronous operation signals that its operation state may be safely destroyed, however the main operation is still executing and therefore depending on its operation state, which is a subobject of the overall operation's operation state. Therefore this approach is unworkable (as it leads to UB).

There is a second solution: Wait for the main operation to complete. However at this point the main operation is no longer making forward progress, which was the point of requesting stopped, and therefore the logic from the first bullet applies.

Due to the above this paper requires the request stop operation to have the following completion signatures:

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

Because there is no reasonable way for `stopped_by` to react to a failure therefrom.

Surveying C-style cancellation APIs makes the above finding less dire than it may seem. Failures from C-style cancel APIs fall primarily into two broad categories:

- Allocation failure, which suggests that the operation should be redesigned to avoid this [9]
- Indication that the operation to cancel could not be found (e.g. `IORING_OP_ASYNC_CANCEL`'s operation ending with `-ENOENT`), because it has already completed, which is an indication that we're in the case of the first bullet from above

Note that `IORING_OP_ASYNC_CANCEL` has another failure modality, `-EALREADY`, which indicates that "the request was found but already completing," which arguably falls under the latter bullet above.

For similar reasons to the above the proposed design also prohibits:

- Invocation of the stop request sender factory, and
- Connecting the resulting sender

From throwing.

Completing vs. Suppressing

Certain C cancellation APIs (e.g. `IORING_OP_ASYNC_CANCEL`) cause the outstanding operation, if any, to end in error. This is inline with the `std::execution/structured` concurrency formulation that asynchronous operations, once initiated (via `std::execution::start`) complete exactly once.

Other C cancellation APIs (e.g. `EPOLL_CTL_DEL`), on the other hand, suppress the outstanding operation (i.e. once interest has been withdrawn via `EPOLL_CTL_DEL` subsequent calls to `epoll_wait` will not return anything about the targeted file descriptor).

The above dichotomy raises a question for the design of `stopped_by`: Should the provided stop request sender factory, or the resulting asynchronous operation, be able to signal that completion of the main operation was suppressed (i.e. that the operation will never report completion)?

If the answer to this question were “yes” it would raise uncomfortable questions about `std::execution`’s model. The standard is currently quite clear (§33.3 [exec.async.ops], emphasis added):

*“An asynchronous operation shall not execute a completion operation before its start operation has begun executing. After its start operation has begun executing, **exactly one completion operation** shall execute.”*

Adding the ability for completion to be suppressed would introduce a carve out to this general requirement.

The workaround is simple: Since the operation being stopped and the request stop sender factory must be cooperating (since they share a closure pointer) the request stop operation can simply induce the main operation to complete via some API known to both of them when suppression occurs.

Lastly, there’s another argument against this approach: Due to races, seemingly-simple suppression APIs aren’t always simple. Consider the above example of `EPOLL_CTL_DEL`: A successful call thereto does indeed remove interest, thereby suppressing further reports through `epoll_wait`. However it does nothing to claw back interest reports which may have been yielded by `epoll_wait`, but which the servicing thread has not yet processed.

Consequently simply withdrawing interest (via `EPOLL_CTL_DEL`) is insufficient to render it safe to destroy the operation state. This reveals the elegance of `std::execution`’s model: Rather than attempting to determine, from the outside, whether or not an operation is complete (and it’s therefore safe to destroy its operation state and move on) one waits for the operation itself to announce its completion.

Classification of Stopped

The split between `started_by` [1] and `stopped_by` is underpinned by the single responsibility principle [10]. Concretely: An asynchronous operation provided by a C-style API should be adapted in a straightforward way (through an implementation simplified by `started_by`), and then stop request support should be added by enriching that sender via `stopped_by`. This implies that the wrapper for the asynchronous operation provided via a C-style API passes

through completion relatively undisturbed. Put concretely: Cancellation will be reported via `std::execution::set_error_t(T)` rather than `std::execution::set_stopped_t()`.

Accordingly, if `stopped_by` is to provide an idiomatic interface, there must be a customization point which can classify completions as indicating that a stop request was received, and actioned.

This suggests an additional refinement to the interface of `stopped_by`:

```
std::execution::sender auto stopped_by(  
    std::execution::sender auto&& child,  
    auto&& request_stop,  
    auto&& classifier);
```

Where `classifier` is an optional parameter which, if provided, must be rvalue invocable with the signatures suggested by the following:

```
template<class Tag, class... ResultDatums>  
bool operator()(Tag, const ResultDatums&...) noexcept;
```

Whenever the main operation completes that completion is passed to the “classifier” and, if it returns true, the completion is discarded and coalesced to `set_stopped_t()`.

Note that, in principle, throwing classifiers could be supported, however this proposal does not allow them.

Above it is said that the classifier ought to be “rvalue invocable.” Given that its job is simply classification of completion signatures this might initially seem odd. However consider the shape of an asynchronous operation, such as the main operation whose completion is being classified (§33.3 [exec.async.ops], emphasis added):

*“After [the] start operation [of an asynchronous operation] has begun executing, **exactly one completion operation** shall execute.”*

As such the classifier will be invoked exactly once. As such invocation thereof is consumptive and it should proceed against an rvalue.

The last question regarding the classifier concerns what to do with completions which arrive when no stop has been requested: Should they be passed to the classifier or not?

At first the answer might seem obvious: Of course not, since no stop has been requested the laws of causality indicate that the completion is not due to a stop request, and therefore oughtn’t be coalesced to `std::execution::set_stopped_t()`. Put differently: A completion cannot be due to a stop request if the completion temporally precedes the request thereof.

“Temporally precedes” immediately evokes a “happens before relationship.” However establishing such a relationship, in general, has a cost. We therefore have a trade off: Save a

single invocation of the classifier at the cost of establishing a happens before relationship between completion and stop request. Making the assumption classification should be cheap (e.g. a single branch) this starts to indicate a non-obvious answer: Invoke the classifier every time.

This answer immediately raises another concern: What about erroneous classification? If the classifier is invoked for every completion, and those completions do not temporally follow a stop request, then perhaps the classifier erroneously classifies such a completion as `std::execution::set_stopped_t()`.

This initially seems persuasive until one realizes that “and those completions do not temporally follow a stop request” has nothing to do with the concern. Recall above we invoked “causality.” If a completion signature can be misclassified as `std::execution::set_stopped_t()` before a stop is requested, then it can equally be thusly miscategorized thereafter.

Consider an operation that sends `std::execution::set_value(..., 5)` when it is stopped, and also in some other case(s). Now two scenarios. The first:

1. Main operation completes with `std::execution::set_value(..., 5)`
2. Classifier categorizes it as `std::execution::set_stopped_t()`
3. Overall operation completes with `std::execution::set_stopped(...)`

The second:

1. Stop is requested
2. Main operation completes with `std::execution::set_value(..., 5)` (but not because of the stop request, it's simply a racy completion which asynchronous algorithms must be robust to)
3. Classifier categorizes it as `std::execution::set_stopped_t()`
4. Overall operation completes with `std::execution::set_stopped(...)`

Both of these situations are arguably problematic, but recording whether stop has been requested and predicating invocation of the classifier thereupon only addresses the former.

This points towards this being a structural problem with the underlying asynchronous operation: It doesn't properly communicate why it was induced to complete. We shouldn't pessimize a generic algorithm to half solve a problem that might exist with one of the algorithms it might compose.

Therefore this paper proposes unconditional invocation of the classifier.

Composition of Asynchronous Operations

`stopped_by` consists of at least two, and at most three, parts which must be coordinated:

- `std::execution::start` on the child operation
- Construction of the stop callback
- The request stop asynchronous operation

Note that the last bullet is conditional (hence “at least two, and at most three” above): It only exists (and must therefore only be coordinated) if a stop is actually requested.

The above decomposition might initially seem odd: Why are `std::execution::start` and construction of the stop callback separate parts which must be coordinated? Surely this is straightforward, synchronous code:

```
execution::start(op);
stop_callback.emplace(get_stop_token(get_env(rcvr)),
                      on-stop-request{*this});
```

Invoking `std::execution::start`, however, exits the synchronous domain and enters the asynchronous domain. The resulting asynchronous operation can proceed in the background, and complete, potentially, at any time thereafter, including *racily*.

A straightforward attempt to address the above, and reduce coordination overhead, might be to flip the order:

```
stop_callback.emplace(get_stop_token(get_env(rcvr)),
                      on-stop-request{*this});
execution::start(op);
```

This does address the above, and reduce coordination overhead, but it is also incorrect. Consider the case where a stop has already been requested when `stopped_by` is initiated:

1. Stop callback is constructed, invoking `on-stop-request{*this}` inline since stop is already requested
2. `std::execution::start(op)` initiates the main asynchronous operation

This attempts to stop an asynchronous operation before it has started. There are many possible outcomes of this but none of them are stopping an operation which isn't in progress.

Just as in the case of `std::execution::when_all` `stopped_by` must track the completion of all its parts, and complete only once all of them have completed. This means that completion of the main operation may need to be “suspended” to wait for outstanding parts to “arrive.”

Fortunately the proposed `std::execution::storage_for_completion_signatures` [11] greatly simplifies specification of this.

Storing a completion, and later retransmitting that completion to a receiver, is not without cost, however. And an analysis of asynchronous operations indicates they modally:

- Are not stopped, and

- Do not race with initiation

Taken together this means that almost all completions of `stopped_by` will be performed directly inline with a completion of its main operation. In such a situation the cost of storing the completion, and then retransmitted it need not be paid. As such the proposal optimizes for this situation and permits pass through of such completions.

`std::unstoppable_token`

The *raison d'être* of `stopped_by` is to enrich a sender with the ability to be stopped. However in certain situations a stop request will never come, such as when the stop token obtained from the receiver's environment satisfies `std::unstoppable_token`.

In this case all of the machinery added by `stopped_by` becomes unnecessary: A stop request will never arrive, and therefore none of what `stopped_by` does is necessary. Invocation of a classifier, advertisement of a `std::execution::set_stopped_t()` completion signature, and the overhead of coordinating the various “parts” discussed above can all be elided.

In fact when the stop token obtained from the receiver's environment satisfies `std::unstoppable_token` `stopped_by`'s implementation of `connect` can simply `if constexpr` down to:

```
auto&& [_, _, child] = sndr;
return execution::connect(std::forward_like<decltype(sndr)>(child),
                          std::move(rcvr));
```

Which is exactly what this paper proposes.

Completion Behavior of Child Operations

`stopped_by`'s implementation can also be optimized depending on when and where the involved asynchronous operations are able to complete.

Unfortunately, as of this writing, `std::execution` does not have a generic way to transmit this information. However a proposal was made to add this [12] which refers to the above-described property as “completion behaviour [sic]” (note that P3206 uses the British spelling of “behavior,” this paper will hereafter use the American spelling except in code examples).

The most significant optimization which the above would unlock would be eliminating `stopped_by` entirely (as is done in the case of `std::unstoppable_token`): If the main operation reports `completion_behaviour::inline_completion` or `::synchronous` then the operation completes before `std::execution::start` returns, therefore constructing the stop

callback will never have any effect, therefore `stopped_by` does nothing, and can be completely eliminated.

Unfortunately the above requires that the completion behavior be determined at compile time, from the type of the sender and the type of the environment, something P3206 does not support since its proposed basis operation `get_completion_behaviour` requires a sender and environment value. Perhaps the above provides motivation for an improvement to the API proposed by P3206.

Were the above possible there would be another improvement that could be made: If the stop request operation reports `completion_behaviour::inline_completion` or `::synchronous` then the operation state therefor can be a function local variable, and the coordination discussed at length above can be partially elided.

Even if P3206 is later extended to support compile time interrogation of types there are still runtime optimizations that can be made. A later section of this paper presents a diff against its concrete proposal which illustrates what would be possible were P3206 the status quo.

Note that in that section there's a tension: The completion behavior needs to be inferred from a reference count rather than being interrogated directly, this is because at the point at which that information is useful the sender has been consumed to form an operation state. Perhaps this motivates the inclusion of an operation state-level determination as to completion behavior.

Lastly, consideration of this area reveals the usefulness of a new completion behavior beyond the taxonomy proposed by P3206: Asynchronous completion limited to the current thread. Were this behavior representable coordination between the construction of the stop callback and the completion of the main operation would be unnecessary, unlocking new optimizations.

Destruction of Stop Operation State

`stopped_by` connects the child sender when it is connected. The resulting operation state is destroyed when `stopped_by`'s operation state is destroyed.

`stopped_by`'s operation state conditionally contains another operation state: That of the stop operation. The wording provided by this paper stores this operation state as follows:

```
optional<stop-op-t> stop-op; // exposition only
```

An optional is necessary because:

- The operation state is conditionally present (i.e. only when stop has been requested), and

- When the operation state is present, it is not present from the moment whereat the `stopped_by` operation state is created (since it's only created later in response to a stop request)

We can be even more precise regarding the bounds of the lifetime of the stop operation state. The standard requires that, in general, asynchronous operations only make use of the stop token provided through the receiver's environment:

- After `start` is called, and
- Before the completion of the operation

(§33.7.1 [exec.recv.concepts]):

“Let `rcvr` be a receiver and let `op_state` be an operation state associated with an asynchronous operation created by connecting `rcvr` with a sender. Let `token` be a stop token equal to `get_stop_token(get_env(rcvr))`. `token` shall remain valid for the duration of the asynchronous operation's lifetime.

“[Note 1: This means that, unless it knows about further guarantees provided by the type of `rcvr`, the implementation of `op_state` cannot use `token` after it executes a completion operation. This also implies that any stop callbacks registered on `token` must be destroyed before the invocation of the completion operation. — end note]”

This means the lifetime of the stop operation state begins (via `std::execution::connect`) within the `stopped_by` operation's async lifetime (§33.3 [exec.async.ops]):

“The lifetime of an asynchronous operation, also known as the operation's async lifetime, begins when its start operation begins executing and ends when its completion operation begins executing.”

And since `std::execution` asynchronous operations always complete (§33.3 [exec.async.ops], emphasis added):

*“After [the] start operation [of an asynchronous operation] has begun executing, **exactly one completion operation** shall execute.”*

This means that if `stopped_by` is specified to destroy the stop operation state within the completion signal therefrom the overhead (i.e. a boolean and check in the destructor) of `std::optional` is unnecessary [16][17].

This is exactly the sort of “principled reduction of operation state lifetimes” LEWG voted in favor of in Wrocław (November 2024) [18] and is therefore the approach proposed by this paper.

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 stopped_by_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 stopped_by_t stopped_by{};
}

```

[exec.stopped_by]

Note: This is a new section. This depends on P4191 [14], P4288 [11] and P4354 [15].

`stopped_by` adapts a sender such that it may be stopped by an external asynchronous operation, and so that a completion thereof as a result of said external asynchronous operation can be appropriately coalesced to `set_stopped_t()`.

The name `stopped_by` denotes a pipeable sender adaptor object. For subexpressions `sndr`, `f`, and `pred` if:

- `decltype((sndr))` does not satisfy `sender`, or
- `decltype((f))` does not satisfy *movable-value*, or
- `decltype((pred))` does not satisfy *movable-value*

the expression `stopped_by(sndr, f, pred)` is ill-formed. Otherwise, the expression `stopped_by(sndr, f, pred)` is expression-equivalent to:

```
make_sender(stopped_by, product_type{f, pred}, sndr)
```

Let *no-classifier* be a unique, empty, exposition-only type.

For subexpressions `sndr` and `f`, the expression `stopped_by(sndr, f)` is expression-equivalent to `stopped_by(sndr, f, no-classifier{})`.

Let *archetype-receiver* denote the following exposition-only class template:

```

template<class Env>
struct archetype-receiver {

```

```

using receiver_concept = receiver_tag;

template<class... ResultDatums>
constexpr void set_value(ResultDatums&&...) && noexcept {}
template<class ResultDatum>
constexpr void set_error(ResultDatum&&) && noexcept {}
constexpr void set_stopped() && noexcept {}

constexpr Env get_env() const noexcept {}
};

```

Let *stop-factory* denote the following exposition-only concept:

```

template<class Factory, class Op, class Env>
concept stop-factory =
    same_as<
        completion_signatures<set_value_t()>,
        completion_signatures_of_t<
            invoke_result_t<Factory, Op&>,
            Env>> &&
        is_nothrow_connectable_in_v<
            invoke_result_t<Factory, Op&>,
            Env>;

```

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

```

namespace std::execution {
    template<>
        struct impls-for<stopped_by_t> : default-impls {
            static constexpr auto connect = see below;

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

```

```

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

```

Effects: Equivalent to:

```

using child_type = child-type<Sndr, 0>;
auto signatures = get_completion_signatures<child_type, Env...>();
get_stored_completion_signatures_type(signatures);

```

```

using data_type = data-type<Sndr>;
using input_factory_type = decltype(get<0>(declval<data_type>()));
using input_classifier_type = decltype(get<1>(declval<data_type>()));
using factory_type = decay_t<input_factory_type>;
using classifier_type = decay_t<input_classifier_type>;

if constexpr (!is_constructible_v<factory_type, input_factory_type> ||
              !is_constructible_v<classifier_type, input_classifier_type>)
{
    throw unspecified-exception();
}

if constexpr (!is_same_v<no-classifier, classifier_type>) {
    template for (constexpr auto signature :
                  template_arguments_of(^decltype(signatures)))
    {
        constexpr auto tag = return_type_of(signature);
        if (tag != ^set_stopped_t) {
            auto args = parameter_types_of(signature);
            ranges::transform(args, args.begin(), meta::remove_reference);
            ranges::transform(args, args.begin(), meta::add_const);
            ranges::transform(args, args.begin(), meta::add_lvalue_reference);
            args.insert(args.begin(), tag);

            if (!is_nothrow_invocable_r_type(^bool,
                                             ^const classifier_type&,
                                             args))
            {
                throw unspecified-exception();
            }
        }
    }
}

if constexpr (
    (sizeof...(Env) == 1) &&
    requires {
        typename completion_signatures_of_t<child_type, Env...>;
    })
{
    using receiver_type = archetype-receiver<Env...>;
    using operation_state_type = connect_result_t<child_type, receiver_type>;

```

```

    if constexpr (!stop-factory<factory_type, operation_state_type, Env...>) {
        throw unspecified-exception{};
    }
}

```

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

```

template<class Sndr, class Factory, class Classifier, class Rcvr>
struct stopped-by-state {
    using operation_state_concept = operation_state_tag;

    struct receiver {                                     // exposition only
        using receiver_concept = receiver_tag;

        stopped-by-state& state;                         // exposition only

        template<class... ResultDatums>
        constexpr void set_value(ResultDatums&&... datums) && noexcept {
            state.complete(execution::set_value,
                           std::forward<ResultDatums>(datums)...);
        }

        template<class ResultDatum>
        constexpr void set_error(ResultDatum&& datum) && noexcept {
            state.complete(execution::set_error,
                           std::forward<ResultDatum>(datum));
        }

        constexpr void set_stopped() && noexcept {
            state.complete(execution::set_stopped);
        }

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

    struct stop-receiver {                               // exposition only
        using receiver_concept = receiver_tag;

        stopped-by-state& state;                         // exposition only

        constexpr void set_value() && noexcept {
            auto& state = this->state;

```

```

        state.stop-op.reset();
        state.maybe-complete();
    }

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

struct on-stop-request { // exposition only
    stopped-by-state& state; // exposition only

    constexpr void operator()() && noexcept {
        if (state.outstanding.fetch_add(1, memory_order_relaxed) == 0) {
            return;
        }

        auto& op = state.stop-op.emplace(emplace-from([&] {
            return connect(INVOKE(std::move(state.factory), state.op),
                               stop-receiver{state}));
        }));
        execution::start(op);
    }
};

using op-t = connect_result_t<Sndr, receiver>; // exposition only
using stop-sender-t = invoke_result_t<Factory, op-t>; // exposition only
using stop-op-t =
    connect_result_t<stop-sender-t, stop-receiver>; // exposition only
using completion-signatures-t =
    completion_signatures_of_t<Sndr, env_of_t<Rcvr>>; // exposition only
using stopped-completion-signatures-t = see below; // exposition only
using storage-t = storage_for_completion_signatures<
    conditional_t<same_as<no-classifier, Classifier>,
                 completion-signatures-t,
                 stopped-completion-signatures-t>>; // exposition only
using stop-callback-t = stop_callback_for_t<
    stop_token_of_t<env_of_t<Rcvr>>,
    on-stop-request>; // exposition only

Rcvr rcvr; // exposition only
Factory factory; // exposition only
Classifier classifier; // exposition only

```

```

    atomic<unsigned> outstanding{2}; // exposition only
    op-t op; // exposition only
    optional<stop-op-t> stop-op; // exposition only
    storage-t storage; // exposition only
    optional<stop-callback-t> stop-callback; // exposition only

```

```

constexpr void maybe-complete() noexcept { // exposition only
    if (outstanding.fetch_sub(1, memory_order_release) != 1) {
        return;
    }
    atomic_thread_fence(memory_order_acquire);

```

```

    stop-callback.reset();

```

```

    std::move(storage).complete(std::move(rcvr));
}

```

```

template<class Tag, class... ResultDatums>
constexpr void complete(Tag tag, ResultDatums&&... datums) noexcept {
    if constexpr (
        !same_as<no-classifier, Classifier> &&
        !same_as<Tag, set_stopped_t>)
    {
        if (INVOKE<bool>(std::move(classifier), Tag{}, as_const(datums)...)) {
            storage.arrive(set_stopped);
            maybe-complete();
            return;
        }
    }
}

```

```

    if (outstanding.load(memory_order_relaxed) == 1) {
        atomic_thread_fence(memory_order_acquire);
        stop-callback.reset();
        if (outstanding.load(memory_order_acquire) == 1) {
            tag(std::move(rcvr), std::forward<ResultDatums>(datums)...);
            return;
        }
    }
}

```

```

    storage.arrive(tag, std::forward<ResultDatums>(datums)...);
    maybe-complete();
}

```

```
constexpr stopped-by-state(Sndr&& sndr, Factory f, Classifier c, Rcvr rcvr)
    noexcept(
        is_nothrow_move_constructible_v<Factory>    &&
        is_nothrow_move_constructible_v<Classifier> &&
        noexcept(connect(declval<Sndr>>(), declval<receiver>()))))
    : rcvr(std::move(rcvr)),
      factory(std::move(f)),
      classifier(std::move(c)),
      op(connect(std::forward<Sndr>(sndr), receiver{*this}))) {}

constexpr void start() & noexcept {
    execution::start(op);

    stop_callback.emplace(get_stop_token(get_env(rcvr)),
                          on-stop-request{*this});

    maybe-complete();
}

};
```

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

```

        std::forward_like<Sndr>(pred),
        std::move(rcvr));
    }
}

```

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

Let `Child` denote `child-type<Sndr, 0>`, let `Factory` denote `decay_t<decltype(get<0>(declval<data-type<Sndr>>()))>`, let `Classifier` denote `decay_t<decltype(get<1>(declval<data-type<Sndr>>()))>`, and let `Env` denote `env_of_t<Rcvr>`. Let `State` denote `stopped-by-state<Child, Factory, Classifier, Rcvr>` and let `Op` denote `State::op-t`. The expression in the requires-clause is equivalent to:

```
unstoppable_token<stop_token_of_t<Env>> || stop-factory<Factory, Op, Env>
```

Expanded Proposal

Note: This section assumes P3206 [12]. It presents a diff relative to the above proposal.

[...]

Let `stopped-by-state` denote the following exposition-only class template:

```
template<class Sndr, class Factory, class Classifier, class Rcvr>
struct stopped-by-state {

```

[...]

<code>Rcvr rcvr;</code>	<code>// exposition only</code>
<code>Factory factory;</code>	<code>// exposition only</code>
<code>Classifier classifier;</code>	<code>// exposition only</code>
<code>atomic<unsigned> outstanding{2};</code>	<code>// exposition only</code>
<code>op-t op;</code>	<code>// exposition only</code>
<code>optional<stop-op-t> stop-op;</code>	<code>// exposition only</code>
<code>storage-t storage;</code>	<code>// exposition only</code>
<code>optional<stop-callback-t> stop-callback;</code>	<code>// exposition only</code>

[...]

```
constexpr stopped-by-state(Sndr&& sndr, Factory f, Classifier c, Rcvr rcvr)
noexcept(
    is_nothrow_move_constructible_v<Factory>    &&
    is_nothrow_move_constructible_v<Classifier> &&

```

```

    noexcept(connect(declval<Sndr>(), declval<receiver>()))
: rcvr(std::move(rcvr)),
  factory(std::move(f)),
  classifier(std::move(c)),
  outstanding([&] noexcept {
    switch (get_completion_behaviour(sndr, get_env(rcvr))) {
    case completion_behaviour::inline_completion:
    case completion_behaviour::synchronous:
      return 1;
    default:
      return 2;
    }
  }()),
  op(connect(std::forward<Sndr>(sndr), receiver{*this})) {}

constexpr void start() & noexcept {
  const auto synchronous = outstanding.load(memory_order_relaxed) == 1;

  execution::start(op);

  if (synchronous) {
    return;
  }

  stop_callback.emplace(get_stop_token(get_env(rcvr)),
    on-stop-request{*this});

  maybe-complete();
}
};

[...]
```

Implementation Experience

The author has implemented the predecessor to this algorithm, `externally_completed`, and used it in the implementation of an `io_uring` execution context [2][3], and integration of `libcares` asynchronous operations into the `std::execution` ecosystem [4]. The lock-free algorithm used has also been discussed at length [5].

In making `externally_completed` ready for proposal the author realized that it could be split in two: `stopped_by` (proposed by this paper) and `started_by` [1].

stopped_by has been implemented against Nvidia's stdexec [19].

References

- [1] R. Leahy. started_by P4394R0
- [2] R. Leahy. Towards Async Everything Part 1: Senders as the Lowest Layer C++Now 2026
- [3] R. Leahy. Towards Async Everything Part 2: Scopes, Construction, and Destruction C++Now 2026
- [4] R. Leahy. The Async Pattern Graveyard ACCU on Sea 2026
- [5] R. Leahy. What Are We Synchronizing? CppCon 2026
- [6] K. Shoop et al. Cancellation is serendipitous-success P1677R2
- [7] R. Leahy. It's Scopes All the Way Down P3955R2
- [8] R. Leahy. Throwing Violation Handlers Are Post Hoc Library Design P4254R0
- [9] <https://github.com/c-ares/c-ares/pull/1154>
- [10] <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>
- [11] R. Leahy. Stop the Decay P4288R2
- [12] M. Nadolski. A Sender Query for Completion Behaviour P3206R0
- [13] <https://github.com/c-ares/c-ares/issues/639>
- [14] R. Leahy. is_nothrow_connectable_in P4191R1
- [15] R. Leahy. No More transform_sender Workarounds P4354R0
- [16] R. Leahy. Extending std::execution Further: Higher-Order Senders and the Shape of Asynchronous Programs C++ on Sea 2025
- [17] R. Leahy. It's Scopes All the Way Down P3955R2
- [18] R. Leahy. Of Operation States and Their Lifetimes P3373R4
- [19] <https://github.com/NVIDIA/stdexec/pull/2346>