A sender query for completion behaviour

Document #:	P3206R0
Date:	2025-01-13
Project:	Programming Language C++
Audience:	SG1

Reply-to:	Maikel Nadolski <maikel.nadolski@gmail.com>

1 Abstract

A sender query is proposed to improve the lifetime management for child operation states. There is no wording in this paper yet.

2 Introduction

This proposal partially continues on [P2257R0]. It proposes the query `execution::get_completion_behaviour` that determines whether an async operation completes inline, synchronous or asynchronously with a call to `execution::start`. This corresponds to the completion guarantees in [P2257R0]. The current proposal does not include a query to investigate whether starting an operation will block an execution agent.

Knowing whether an operation completes synchronously changes the way how the lifetime of the operation state can be managed, and inline completion implies synchronous completion. No guarantees regarding the completion behaviour are being made, if no suitable overload for a sender and environment pair is being found.

Awaitables have a very similar way to convey this information by returning `true` or `false` for `await_ready()`. Without this proposal senders in `std::execution` are missing a tool to do so.

This query proves to be instrumental in optimizing the behavior for some algorithms, including but not limited to `sync_wait()`, `repeat()`-like algorithms, the `as_awaitable()` helper or a scheduler-affine coroutine task type.

2.1 `sync_wait` algorithm

A possible implementation of `sync_wait` synchronizes the completion of the input operation with some signaling mechanism. If the input operation is known to complete synchronously, `sync_wait` does not need the synchronization primitives in its implementation.

2.2 `repeat`-like algorithms

Consider a typical `repeat`-like sender algorithm that repeats the construction and initiation of an input operation.

A usual implementation reconstructs and starts a child operation from within the value completion of an intermediate receiver. If the completion behaviour of the child operation is not known, one needs to mitigate a possible recursive call stack e.g., by rescheduling on a lightweight scheduler.

If the child operation completes synchronously a `repeat`-like algorithm can be implemented with a while loop, similar to the following snippet:

```
bool stopped{false};
optional<connect_result_t<Sender, repeat_receiver>> child_op(
    std::in_place,
    emplace_from{[&] { return exection::connect(sender, repeat_receiver{&stopped}); }});
while (!stopped) {
  execution::start(*child_op);
  child_op.emplace(
      emplace_from{[&] { return exection::connect(sender, repeat_receiver{&stopped}); }});
}
```

2.3 as_awaitable

The current specification of `as_awaitable` transforms senders into awaitable that can not use symmetric transfer, even if it would be feasible, because the necessary information is not available. Consequently, starting the following code will run into stack-exhaustion on a typical machine

```
struct promise;

struct coroutine : std::coroutine_handle<promise>
{
    using promise_type = ::promise;
};

struct promise : std::execution::with_awaitable_senders<promise>
{
    coroutine get_return_object() { return {coroutine::from_promise(*this)}; }
    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {}
};

coroutine f() {
    // this will most likely produce stack-overflow
    for (int i = 0; i < 1'000'000; ++i) {
        co_await std::execution::just();
    }
}

int main()
{
    f();
}
```

Note, that senders are allowed to provide custom implementations for `as_awaitable` by providing the respective class member method. To the author it is not clear how this customization point scales with the composition of sender algorithms and synchronous senders. This proposal allows to statically detect synchronous or inline completion behaviour of the operation and transforms a sender into an awaitable that symmetrically transfers the control back to its parent coroutine in a scheduler-affine fashion. Transforming a synchronous sender can, for example, look like this

```
template <synchronous-single-sender<env_type> S>
struct awaiter {
  S sender;
  variant<monostate, single-value-result-of<S>, exception_ptr> result;
```

```
  bool await_ready() { return false; }

  template <class P>
  coroutine_handle<P> await_suspend(coroutine_handle<P> h) {
    auto op = connect(std::move(sender), receiver_t{&result, get_env(h.promise())});
    op.start();
    return h;
  }

  single-value-result-of<S>
  await_resume() {
    // Check the state of result and
    // either return the value or
    // rethrow the exception
  }
};
```

2.4 scheduler-affine task type

Consider a scheduler-affine coroutine task type that ensures that each awaited expression completes on the currently assigned scheduler.

Ideally, one wants to avoid to reschedule an await-expression if it did not change the current execution resource. This opens up the question on how to identify such senders and awaitables that complete on the scheduler that started them.

One family of senders are those whose value completion scheduler is known and is equal to the starting scheduler.

Another family of senders are those whose corresponding sender-awaitable will resume on the current thread of execution. This proposal helps to identify the second group of senders.

3 Proposal

Let `s` denote a sender and `env` an environment type.
I propose a customization point object `execution::get_completion_behaviour(s, env)` with an interface that is similar to `get_completion_signatures`. Authors of sender algorithms can define

1. a possibly env-dependent member method on the sender type

```
template <class InputSender>
struct example_sender_adaptor {
  // [...]

  template <class Self, class... Env>
    requires (sizeof...(Env) <= 1)
  constexpr auto get_completion_behaviour(this Self&& self, Env&&... env) noexcept {
    return execution::get_completion_behaviour(
        std::forward_like<Self>(self.sender), std::forward<Env>(env)...);
  }

  InputSender sender;
};
```

2. a type alias if the completion behaviour is statically known and does not depend on the environment

```
struct example_sender1 {
  // [...]

  using completion_behaviour = constant<execution::completion_behaviour::asynchronous>;
};
```

3.1 `execution::get_completion_behaviour`

The return type of `get_completion_behaviour(s, env)` is convertible to `execution::completion_behaviour`.

— `completion_behaviour::inline_completion`: The connected receiver's completion-signal will occur on the calling thread before `execution::start()` returns.

— `completion_behaviour::synchronous`: The connected receiver's completion-signal happens-before `execution::start()` returns.

— `completion_behaviour::asynchronous`: The connected receiver's completion-signal will not occur on the calling thread before `execution::start()` returns.

— `completion_behaviour::unknown`: The completion behaviour is unknown.

If `get_completion_behaviour(s, env)` is an invalid expression no guarantee will be made.

4 Implementation Experience

libunifex uses a `blocking(const Sender&) -> blocking_kind` query to provide this information for optimizations, which is very similar to `get_completion_behaviour`. It's valid values are

— `maybe`: the completion behaviour is not known
— `never`: the receiver will never be called on the current thread before `start()` returns.
— `always`: the receiver is guaranteed to be called on some thread strongly-happens-before `start()` returns.
— `always_inline`: the receiver is guaranteed to be called inline on the current thread before `start()` returns.

The main difference of this proposal to the implementation at libunifex is, that the query in this proposal acts additionally on the environment instead of only a sender. Note, that libunifex does not have receiver environments and the completion behaviour is considered to be a property of the operation.

5 Design Alternatives

It was considered to make the query directly dependent on the operation state instead of a sender and environment pair. While it would technically work to query the operation states directly, it would also require to instantiate those operation state types.

Another alternative is to consider a boolean predicate, such as `completes_synchronously(s, env)`, instead. Note, that `inline_completion` implies `synchronous` completion. In practise, standardized senders that are synchronous will complete inline but it is the synchronous completion property that enables the optimizations. The current proposal can distinguish between both completion types, inline and synchronous, and therefore it was chosen over the boolean predicate formulation.

6 Implications on Sender Factories and Adaptors

The following section describes how to compute the values for the query for each sender algorithm in `std::execution` and its default implementations. Domain specializations are allowed to change those values.

For sake of computations we assume the following total order of values:

```
unknown < asynchronous < synchronous < always_inline
```

6.1 Sender Factories

In general, each sender factory needs to provide the information from its respective implementation.

6.1.1 schedule(run_loop::scheduler)

— `get_completion_behaviour`: `asynchronous`

6.1.2 just(), just_error(), just_stopped()

— `get_completion_behaviour`: `inline_completion`

6.1.3 read_env()

— `get_completion_behaviour`: `inline_completion`

6.2 Sender Adaptors

6.2.1 finally(sender1, sender2), continues_on(sender, scheduler), starts_on(scheduler, sender)

Let `s` denote the expression `finally(sender1, sender2)`.

— `get_completion_behaviour(s, env)` returns
`min(get_completion_behaviour(sender1, env), get_completion_behaviour(sender2, env))`

6.2.2 then(sender, fn), upon_error(sender, fn), upon_stopped(sender, fn)

Let `s` denote the expression `then(sender, fn)`.

— `get_completion_behaviour(s, env)` returns `get_completion_behaviour(sender, env)`

6.2.3 let_value(sender, fn), let_error(sender, fn), let_stopped(sender, fn)

Let `s` denote the expression `let_value(sender, fn)`. Let `rs...` denote the set of all possible result-senders returned from `fn`.

— `get_completion_behaviour(s, env)` returns
`min(get_completion_behaviour(sender, env), get_completion_behaviour(rs, env)...)`

6.2.4 into_variant(sender), stopped_as_optional(sender)

Let `s` denote the expression `into_variant(sender, fn)`.

— `get_completion_behaviour(s, env)` returns `get_completion_behaviour(sender, env)`

6.2.5 bulk(sender, shape, fn)

Let `s` denote the expression `bulk(sender, shape, fn)`.

— `get_completion_behaviour(s, env)`: `get_completion_behaviour(sender, env)`

6.2.6 when_all(senders...)

Let `s` denote the expression `when_all(senders...)`.

— `get_completion_behaviour(s, env)` returns `min(get_completion_behaviour(senders, env)...)`

6.2.7 `split(sender)`

Let `s` denote the expression `split(sender)`.

— `get_completion_behaviour(s, env)`:
  — Returns a dynamic value. The operation of `split(sender)` completes inline if the input operation has completed before starting a new copy of split. Otherwise the query returns `get_completion_behaviour(sender, env)`.

7 References

[P2257R0] Dalton M. Woodard. 2020-11-22. Blocking is an insufficient description for senders and receivers. https://wg21.link/p2257r0