# A query for completion behaviour of operation states


| | |
|---|---|
| Document #: | D3206R0 |
| Date: | 2025-01-08 |
| Project: | Programming Language C++ |
| Audience: | SG1 |
| Reply-to: | Maikel Nadolski <maikel.nadolski@gmail.com> |


## 1 Abstract


A query on operation states is proposed to improve the lifetime management for child operations. 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 synchronously, inline or asynchronously with a call to `execution::start`. This corresponds to the completion guarantees in [P2257R0]. Knowing whether an operation completes synchronously changes the way how the lifetime of the operation state can be managed. No guarantees regarding the completion behaviour are being made, if no suitable overload for an operation state is being found.

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

The current proposal does not include a query to investigate whether executing an operation will block an associated execution agent.

### 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 by e. g. 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 all senders into an 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

```
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 produce stack-overflow
    for (int i = 0; i < 1'000'000; ++i) {
        co_await std::execution::just();
    }
}

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

Note, that senders are free to provide custom implementations for `as_awaitable` by providing a respective 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 completion behaviour of the operation and transforms a sender into an awaitable that symmetrically transfers the control back to its parent coroutine on the calling thread. Transforming a synchronous sender can exemplarily look like this

```
template <synchronous-single-sender<receiver_t> 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 `op` denote an operation state. The customization point object `execution::get_completion_behaviour(op)` is proposed.

```
struct example_operation {
  // ...
  static constexpr auto get_completion_behaviour() noexcept -> execution::completion_behaviour;
};
```

### 3.1 `execution::get_completion_behaviour`

The return type of `get_completion_behaviour(op)` is convertible to `execution::completion_behaviour`.

**`completion_behaviour::always_inline`:** 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(op)` 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`.

- 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 to the implementation at libunifex is, that the query in this proposal acts on the operation state instead of on a sender. An alternative design could make this a query on a sender that additionally depends on a receiver environment similar to how completion signatures depend on an environment.

# 5 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
```

## 5.1 Sender Factories

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

### 5.1.1 schedule(run_loop::scheduler)

- `get_completion_behaviour`: `asynchronous`

### 5.1.2 just(), just_error(), just_stopped()

- `get_completion_behaviour`: `always_inline`

### 5.1.3 read_env()

- `get_completion_behaviour`: `always_inline`

## 5.2 Sender Adaptors

### 5.2.1 `finally(sender1, sender2)`, `continues_on(sender, scheduler)`, `starts_on(scheduler, sender)`

Let `s` denote the expression `finally(sender1, sender2)`. Let `op` denote the operation associated to `s` and `child1`, `child2` the child operation states of `op`.

- `get_completion_behaviour(op)` returns `min(get_completion_behaviour(child1), get_completion_behaviour(chi`

### 5.2.2 `then(sender, fn)`, `upon_error(sender, fn)`, `upon_stopped(sender, fn)`

Let `s` denote the expression `then(sender, fn)`. Let `op` denote the operation associated to `s` and `child` its child operation.

- `get_completion_behaviour(op)` returns `get_completion_behaviour(child)`

### 5.2.3 `let_value(sender, fn)`, `let_error(sender, fn)`, `let_stopped(sender, fn)`

Let `s` denote the expression `let_value(sender, fn)`. Let `op` be its resulting operation state and `child` denote the child operation of the input sender. Let `rs...` denote the set of all possible result-senders returned from `fn` and `rops...` their operation states.

— `get_completion_behaviour(op)` returns `min(get_completion_behaviour(child), get_completion_behaviour(rops`

### 5.2.4 `into_variant(sender)`, `stopped_as_optional(sender)`

Let `s` denote the expression `into_variant(sender, fn)`. Let `op` be its resulting operation state and `child` denote the child operation of the input sender.

— `get_completion_behaviour(op)` returns `get_completion_behaviour(child)`

### 5.2.5 `bulk(input, shape, fn)`

— `get_completion_behaviour(op)`: `always_inline`

### 5.2.6 `when_all(senders...)`

Let `s` denote the expression `when_all(senders...)` Let `op` be its resulting operation state and `children...` denote the child operations of the input senders.

— `get_completion_behaviour(op)` returns `min(get_completion_behaviour(children)...)`

### 5.2.7 `split(sender)`

— `get_completion_behaviour(op)`: dynamic value, since it depends on whether the shared input operation has already completed or not.

# 6 References

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