Add a Coroutine Lazy Type

Document #:	P3552R0
Date:	2024-01-13
Project:	Programming Language C++
Audience:	Concurrency Working Group (SG1) Library Evolution Working Group (LEWG) Library Working Group (LWG)
Reply-to:	Dietmar Kühl (Bloomberg) <dkuhl@bloomberg.net> Maikel Nadolski <maikel.nadolski@gmail.com>

C++20 added support for coroutines that can improve the experience writing asynchronous code. C++26 added the sender/receiver model for a general interface to asynchronous operations. The expectation is that users would use the framework using some coroutine type. To support that, a suitable class needs to be defined and this proposal is providing such a definition.

Just to get an idea what this proposal is about: here is a simple `Hello, world` written using the proposed coroutine type:

```
#include <execution>
#include <iostream>
#include <task>

namespace ex = std::execution;

int main() {
    return std::get<0>(*ex::sync_wait([]->ex::lazy<int> {
        std::cout << "Hello, world!\n";
        co_return co_await ex::just(0);
    }()));
}
```

1 The Name

Just to get it out of the way: the class (template) used to implement a coroutine task needs to have a name. In previous discussions, SG1 requested that the name `task` is retained and LEWG chose `lazy` as an alternative. It isn't clear whether the respective reasoning is still relevant. To the authors, the name matters much less than various other details of the interface. Thus, the text is written in terms of `lazy`. The name is easily changed (prior to standardisation) if that is desired.

2 Prior Work

This proposal isn't the first to propose a coroutine type. Prior proposals didn't see any recent (post introduction of sender/receiver) update, although corresponding proposals were discussed informally on multiple occasions. There are also implementations of coroutine types based on a sender/receiver model in active use. This section provides an overview of this prior work, and where relevant, of corresponding discussions. This section is primarily for motivating requirements and describing some points in the design space.

2.1 P1056: Add lazy coroutine (coroutine task) type

The paper describes a `task`/`lazy` type (in P1056r0 the name was `task`; the primary change for P1056r1 is changing the name to `lazy`). The fundamental idea is to have a coroutine type which can be `co_await`ed: the interface of `lazy` consists of move constructor, deliberately no move assignment, a destructor, and `operator co_await()`. The proposals don't go into much detail on how to eventually use a coroutine, but it mentions that there could be functions like `sync_await(task<To>)` to await completion of a task (similar to `execution::sync_wait(sender)`) or a few variations of that.

A fair part of the paper argues why `future.then()` is *not* a good approach to model coroutines and their results. Using `future` requires allocation, synchronisation, reference counting, and scheduling which can all be avoided when using coroutines in a structured way.

The paper also mentions support for symmetric transfer and allocator support. Both of these are details on how the coroutine is implemented.

Discussion for P1056r0 in SG1

— The task doesn't really have anything to do with concurrency.
— Decomposing a task cheaply is fundamental. The HALO Optimisations help.
— The `task` isn't move assignable because there are better approaches than using containers to hold them. It is move constructible as there are no issues with overwriting a potentially live task.
— Resuming where things complete is unsafe but the task didn't want to impose any overhead on everybody.
— There can be more than one `task` type for different needs.
— Holding a mutex lock while `co_await`ing which may resume on a different thread is hazardous. Static analysers should be able to detect these cases.
— Votes confirmed the no move assignment and forwarding to LEWG assuming the name is not `task`.
— Votes against deal with associated executors and a request to have strong language about transfer between threads.

2.2 P2506: std::lazy: a coroutine for deferred execution

This paper is effectively restating what P1056 said with the primary change being more complete proposed wording. Although sender/receiver were discussed when the paper was written but `std::execution` hadn't made it into the working paper, the proposal did *not* take a sender/receiver interface into account.

Although there were mails seemingly scheduling a discussion in LEWG, we didn't manage to actually locate any discussion notes.

2.3 cppcoro

This library contains multiple coroutine types, algorithms, and some facilities for asynchronous work. For the purpose of this discussion only the task types are of interest. There are two task types `cppcoro::task` and `cppcoro::shared_task`. The key difference between `task` and `shared_task` is that the latter can be copied and awaited by multiple other coroutines. As a result `shared_task` always produces an lvalue and may have slightly higher costs due to the need to maintain a reference count.

The types and algorithms are pre-sender/receiver and operate entirely in terms for awaiters/awaitables. The interface of both task types is a bit richer than that from P1056/P2506. Below `t` is either a `cppcoro::task<T>` or a `cppcoro::shared_task<T>`:

— The task objects can be move constructed and move assigned; `shared_task<T>` object can also be copy constructed and copy assigned.
— Using `t.is_ready()` it can be queried if `t` has completed.
— Using `co_await t` awaits completion of `t`, yielding the result. The result may be throwing an exception if the coroutine completed by throwing.
— Using `co_await t.when_ready()` allows synchronising with the completion of `t` without actually getting the result. This form of synchronisation won't throw any exception.

— `cppcoro::shared_task<T>` also supports equality comparisons.

In both cases, the task starts suspended and is resumed when it is `co_await`ed. This way a continuation is known when the task is resumed, which is similar to `start(op)`ing an operation state `op`. The coroutine body needs to use `co_await` or `co_return`. `co_await` expects an awaitable or an awaiter as argument. Using `co_yield` is not supported. The implementation supports symmetric transfer but doesn't mention allocators.

The `shared_task<T>` is similar to `split(sender)`: in both cases, the same result is produced for multiple consumers. Correspondingly, there isn't a need to support a separate `shared_task<T>` in a sender/receiver world. Likewise, throwing of results can be avoid by suitably rewriting the result of the `set_error` channel avoiding the need for an operation akin to `when_ready()`.

2.4 libunifex

`unifex` is an earlier implementation of the sender/receiver ideas. Compared to `std::execution` it is lacking some of the flexibilities. For example, it doesn't have a concept of environments or domains. However, the fundamental idea of three completion channels for success, failure, and cancellation and the general shape of how these are used is present (even using the same names for `set_value` and `set_error`; the equivalent of `set_stopped` is called `set_done`). `unifex` is in production use in multiple places. The implementation includes a `unifex::task<T>`.

As `unifex` is sender/receiver-based, its `unifex::task<T>` is implemented such that `co_await` can deal with senders in addition to awaitables or awaiters. Also, `unifex::task<T>` is *scheduler affine*: the coroutine code resumes on the same scheduler even if a sender completed on a different scheduler. The task's scheduler is taken from the receiver it is `connect`ed to. The exception for rescheduling on the task's scheduler is explicitly awaiting the result of `schedule(sched)` for some scheduler `sched`: the operation changes the task's scheduler to be `sched`. The relevant treatment is in the promise type's `await_transform()`:

— If a sender `sndr` which is the result of `schedule(sched)` is `co_await`ed, the corresponding `sched` is installed as the task's scheduler and the task resumes on the context completing `sndr`. Feedback from people working with unifex suggests that this choice for changing the scheduler is too subtle. While it is considered important to be able to explicitly change the scheduler a task executes on, doing so should be more explicit.
— For both senders and awaiters being awaited, the coroutine will be resumed on the task's current scheduler when the task is scheduler affine. In general that is done by continuing with the senders result on the task's scheduler, similar to `continues_on(sender, scheduler)`. The rescheduling is avoided when the sender is tagged as not changing scheduler (using a `static constexpr` member named `blocking` which is initialised to `blocking_kind::always_inline`).
— If a sender is `co_await`ed it gets `connect`ed to a receiver provided by the task to form an awaiter holding an operation state. The operation state gets `start`ed by the awaiter's `await_suspend`. The receiver arranges for a `set_value` completion to become a value returned from `await_resume`, a `set_error` completion to become an exception, and a `set_done` completion to resume a special "on done" coroutine handle rather than resuming the task itself effectively behaving like an uncatchable exception (all relevant state is properly destroyed and the coroutine is never resumed).

When `co_await`ing a sender `sndr` there can be at most one `set_value` completion: if there are more than one `set_value` completions the promise type's `await_transform` will just return `sndr` and the result cannot be `co_await`ed (unless it is also given an awaitable interface). The result type of `co_await sndr` depends on the number of arguments to `set_value`:

— If there are no arguments for `set_value` then the type of `co_await sndr` will be `void`.
— If there is exactly one argument of type `T` for `set_value` then the type of `co_await sndr` will be `T`.
— If there are more than one arguments for `set_value` then the type of `co_await sndr` will be `std::tuple<T1, T2, ...>` with the corresponding argument types.

If a receiver doesn't have a scheduler, it can't be `connect()`ed to a `unifex::task<T>`. In particular, when using a `unifex::async_scope scope` it isn't possible to directly call `scope.spawn(task)` with a `unifex::task<T> task` as the `unifex::async_scope` doesn't provide a scheduler. The `unifex::async_scope` provides a few variations of `spawn()` which take a scheduler as argument.

`unifex` provides some sender algorithms to transform the sender result into something which may be more suitable to be `co_await`ed. For example, `unifex::done_as_optional(sender)` turns a successful completion for a type `T` into an `std::optional<T>` and the cancellation completion `set_done` into a `set_value` completion with a disengaged `std::optional<T>`.

The `unifex::task<T>` is itself a sender and can be used correspondingly. To deal with scheduler affinity a type erased scheduler `unifex::any_scheduler` is used.

The `unifex::task<T>` doesn't have allocator support. When creating a task multiple objects are allocated on the heap: it seems there is a total of 6 allocations for each `unifex::task<T>` being created. After that, it seems the different `co_await`s don't use a separate allocation.

The `unifex::task<T>` doesn't directly guard against stack overflow. Due to rescheduling continuations on a scheduler when the completion isn't always inline, the issue only arises when `co_await`ing many senders with `blocking_kind::always_inline` or when the scheduler resumes inline.

2.5 stdexec

The `exec::task` in stdexec is somewhat similar to the unifex task with some choices being different, though:

— The `exec::task<T, C>` is also scheduler affine. The chosen scheduler is unconditionally used for every `co_await`, i.e., there is no attempt made to avoid scheduling, e.g., when the `co_await`ed sender completes inline.
— Unlike unifex, it is OK if the receiver's environment doesn't provide a scheduler. In that case an inline scheduler is used. If an inline scheduler is used there is the possibility of stack overflow.
— It is possible to `co_await just_error(e)` and `co_await just_stopped()`, i.e., the sender isn't required to have a `set_value_t` completion.

The `exec::task<T, C>` also provides a *context* `C`. An object of this type becomes the environment for receivers `connect()`ed to `co_await`ed senders. The default context provides access to the task's scheduler. In addition an `in_place_stop_token` is provides which forwards the stop requests from the environment of the receiver which is connected to the task.

Like the unifex task `exec::task<T, C>` doesn't provide any allocator support. When creating a task there are two allocations.

3 Objectives

Also see sender/receiver issue 241.

Based on the prior work and discussions around corresponding coroutine support there is a number of required or desired features (listed in no particular order):

1. A coroutine task needs to be awaiter/awaitable friendly, i.e., it should be possibly to `co_await` awaitables which includes both library provided and user provided ones. While that seems obvious, it is possible to create an `await_transform` which is deleted for awaiters and that should be prohibited.

2. When composing sender algorithms without using a coroutine it is common to adapt the results using suitable algorithms and the completions for sender algorithms are designed accordingly. On the other hand, when awaiting senders in a coroutine it may be considered annoying having to transform the result into a shape which is friendly to a coroutine use. Thus, it may be reasonable to support rewriting certain shapes of completion signatures into something different to make the use of senders easier in a coroutine task. See the section on the result type for `co_await` for a discussion.

3. A coroutine task needs to be sender friendly: it is expected that asynchronous code is often written using coroutines awaiting senders. However, depending on how senders are treated by a coroutine some senders

may not be awaitable. For example neither unifex nor stdexec support `co_await`ing senders with more than one `set_value` completion.

4. It is possibly confusing and problematic if coroutines resume on a different execution context than the one they were suspended on: the textual similarity to normal functions makes it look as if things are executed sequentially. Experience also indicates that continuing a coroutine on whatever context a `co_await`ed operation completes frequently leads to issues. Senders could, however, complete on an entirely different scheduler than where they started. When composing senders (not using coroutines) changing contexts is probably OK because it is done deliberately, e.g., using `continues_on`, and the way to express things is new with fewer attached expectations.

   To bring these two views together a coroutine task should be scheduler affine by default, i.e., it should normally resume on the same scheduler. There should probably also be an explicit way to opt out of scheduler affinity when the implications are well understood.

   Note that scheduler affinity does *not* mean that a task is always continuing on the same thread: a scheduler may refer to a thread pool and the task will continue on one of the threads (which also means that thread local storage cannot be used to propagate contexts implicitly; see the discussion on environments below).

5. When using coroutines there will probably be an allocation at least for the coroutine frame (the HALO optimisations can't always work). To support the use in environments where memory allocations using `new`/`delete` aren't supported the coroutine task should support allocations using allocators.

6. Receivers have associated environments which can support an open set of queries. Normally, queries on an environment can be forwarded to the environment of a `connect()`ed receiver. Since the coroutine types are determined before the coroutine's receiver is known and the queries themselves don't specify a result type that isn't possible when a coroutine provides a receiver to a sender in a `co_await` expression. It should still be possible to provide a user-customisable environment from the receiver used by `co_await` expressions. One aspect of this environment is to forward stop requests to `co_await`ed child operations. Another is possibly changing the scheduler to be used when a child operation queries `get_scheduler` from the receiver's environment. Also, in non-asynchronous code it is quite common to pass some form of context implicitly using thread local storage. In an asynchronous world such contexts could be forwarded using the environment.

7. The coroutine should be able to indicate that it was canceled, i.e., to get `set_stopped()` called on the task's receiver. `std::execution::with_awaitable_senders` already provided this ability senders being `co_await`ed but that doesn't necessarily extend to the coroutine implementation.

8. Similar to indicating that a task got canceled it would be good if a task could indicate that an error occurred without throwing an exception which escapes from the coroutine.

9. In general a task has to assume that an exception escapes the coroutine implementation. As a result, the task's completion signatures need to include `set_error_t(std::exception_ptr)`. If it can be indicated to the task that no exception will escape the coroutine, this completion signature can be avoided.

10. When many `co_await`ed operations complete synchronously, there is a chance for stack overflow. It may be reasonable to have the implementation prevent stack overflow by using a suitable scheduler sometimes.

11. In some situations it can be useful to somehow schedule an asynchronous clean-up operation which is triggered upon coroutine exit. See the section on asynchronous clean-up below for more discussing

12. The `lazy` coroutine provided by the standard library may not always fit user's needs although they may need/want various of the facilities. To avoid having users implement all functionality from scratch `lazy` should use specified components which can be used by users when building their own coroutine. The components `as_awaitable` and `with_awaitable_sender` are two parts of achieving this objective but there are likely others.

    The algorithm `std::execution::as_awaitable` does turn a sender into an awaitable and is expected to be used by custom written coroutines. Likewise, it is intended that custom coroutines use the CRTP class template std::execution::with_awaitable_senders. It may be reasonable to adjust the functionality of these components instead of defining the functionality specific to a lazy<...> coroutine task.

It is important to note that different coroutine task implementations can live side by side: not all functionality has to be implemented by the same coroutine task. The objective for this proposal is to select a set of features which provides a coroutine task suitable for most uses. It may also be reasonable to provide some variations as different names. A future revision of the standard or third party libraries can also provide additional variations.

4 Design

This section discusses various design options for achieving the listed objectives. Most of the designs are independent of each other and can be left out if the consensus is that it shouldn't be used for whatever reason.

4.1 Template Declaration for lazy

Coroutines can use co_return to produce a value. The value returned can reasonably provide the argument for the set_value_t completion of the coroutines. As the type of a coroutine is defined even before the coroutine body is given, there is no way to deduce the result type. The result type is probably the primary customisation and should be the first template parameter which gets defaulted to void for coroutines not producing any value. For example:

```
int main() {
    ex::sync_wait([]->ex::lazy<>{
        int result = co_await []->ex::lazy<int> { co_return 42; }();
        assert(result == 42);
    }());
}
```

The inner coroutines completes with set_value_t(int) which gets translated to the value returned from co_await (see co_await result type below for more details). The outer coroutine completes with set_value_t().

Beyond the result type there are a number of features for a coroutine task which benefit from customisation or for which it may be desirable to disable them because they introduce a cost. As many template parameters become unwieldy, it makes sense to combine these into a [defaulted] context parameter. The aspects which benefit from customisation are at least:

— Customising the environment for child operations. The context itself can actually become part of the environment.
— Disable scheduler affinity and/or configure the strategy for obtaining the coroutine's scheduler.
— Configure allocator awareness.
— Indicate that the coroutine should be noexcept.
— Define additional error types.

The default context should be used such that any empty type provides the default behaviour instead of requiring a lot of boilerplate just to configure a particular aspect. For example, it should be possible to selectively enable allocator support using something like this:

```
struct allocator_aware_context {
    using allocator_type = std::pmr::polymorphic_allocator<std::byte>;
};
template <typename T>
using my_lazy = ex::lazy<T, allocator_aware_context>;
```

Using various different types for task coroutines isn't a problem as the corresponding objects normally don't show up in containers. Tasks are mostly co_awaited by other tasks, used as child senders when composing work graphs, or maintained until completed using something like a counting_scope. When they are used in a container, e.g., to process data using a range of coroutines, they are likely to use the same result type and context types for configurations.

4.2 `lazy` Completion Signatures

The discussion above established that `lazy<T, C>` can have a successful completion using `set_value_t(T)`. The coroutine completes accordingly when it is exited using a matching `co_return`. When `T` is `void` the coroutine also completes successfully using `set_value()` when floating off the end of the coroutine or when using a `co_return` without an expression.

If a coroutine exits with an exception completing the corresponding operation with `set_error(std::exception_ptr)` is an obvious choice. Note that a `co_await` expression results in throwing an exception when the awaited operation completes with `set_error(E)` (see below), i.e., the coroutine itself doesn't necessarily need to `throw` an exception itself.

Finally, a `co_await` expression completing with `set_stoppped()` results in aborting the coroutine immediately (see below) and causing the coroutine itself to also complete with `set_stopped()`.

The coroutine implementation cannot inspect the coroutine body to determine how the different asynchronous operations may complete. As a result, the default completion signatures for `lazy<T>` are

```
ex::completion_signatures<
    ex::set_value_t(T),  // or ex::set_value_t() if T == void
    ex::set_error_t(std::exception_ptr),
    ex:set_stopped_t()
>;
```

Support for reporting an error without exception may modify the completion signatures.

4.3 `lazy` constructors and assignments

Coroutines are created via a factory function which returns the coroutine type and whose body uses one of the `co_*` function, e.g.

```
lazy<> nothing(){ co_return; }
```

The actual object is created via the promise type's `get_return_object` function and it is between the promise and coroutine types how that actually works: this constructor is an implementation detail. To be valid senders the coroutine type needs to be destructible and it needs to have a move constructor. Other than that, constructors and assignments either don't make sense or enable dangerous practices:

1. Copy constructor and copy assignment don't make sense because there is no way to copy the actual coroutine state.
2. Move assignment is rather questionable because it makes it easy to transport the coroutine away from referenced entities.

   Previous papers P1056 and P2506 also argued against a move assignment. However, one of the arguments doesn't apply to the `lazy` proposed here: There is no need to deal with cancellation when assigning or destroying a `lazy` object. Upon `start()` of `lazy` the coroutine handle is transferred to an operation state and the original coroutine object doesn't have any reference to the object anymore.
3. If there is no assignment, a default constructed object doesn't make much sense, i.e., `lazy` also doesn't have a default constructor.

Based on experience with Folly the suggestion was even stronger: `lazy` shouldn't even have move construction! That would mean that `lazy` can't be a sender or that there would need to be some internal interface enabling the necessary transfer. That direction isn't pursued by this proposal.

The lack of move assignment doesn't mean that `lazy` can't be held in a container: it is perfectly fine to `push_back` objects of this type into a container, e.g.:

```
std::vector<ex::lazy<>> cont;
cont.emplace_back([]->ex::lazy<> { co_return; }());
cont.push_back([]->ex::lazy<> { co_return; }());
```

The expectation is that most of the time coroutines don't end up in normal containers. Instead, they'd be managed by a `counting_scope` or hold on to by objects in a work graph composed of senders.

Technically there isn't a problem adding a default constructor, move assignment, and a `swap()` function. Based on experience with similar components it seems `lazy` is better off not having them.

4.4 Result Type For `co_await`

When `co_await`ing a sender `sndr` in a coroutine, `sndr` needs to be transformed to an awaitable. The existing approach is to use `execution::as_waitable(sndr)` [exex.as.awaitable] in the promise type's `await_transform` and `lazy` uses that approach. The awaitable returned from `as_awaitable(sndr)` has the following behaviour (`rcvr` is the receiver the sender `sndr` is connected to):

1. When `sndr` completes with `set_stopped(std::move(rcvr))` the function `unhandled_stopped()` on the promise type is called and the awaiting coroutine is never resumed. The `unhandled_stopped()` results in `lazy` itself also completing with `set_stopped_t()`.
2. When `sndr` completes with `set_error(std::move(rcvr), error)` the coroutine is resumed and the `co_await sndr` expression results in `error` being thrown as an exceptions (with special treatment for `std::error_code`).
3. When `sndr` completes with `set_value(std::move(rcvr), a...)` the expression `co_await sndr` produces a result corresponding the arguments to `set_value`:
    1. If the argument list is empty, the result of `co_await sndr` is `void`.
    2. Otherwise, if the argument list contains exactly one element the result of `co_await sndr` is `a...`.
    3. Otherwise, the result of `co_await sndr` is `std::tuple(a...)`.

Note that the sender `sndr` is allowed to have no `set_value_t` completion signatures. In this case the result type of the awaitable returned from `as_awaitable(sndr)` is declared to be `void` but `co_await sndr` would never return normally: the only ways to complete without a `set_value_t` completion is to complete with `set_stopped(std::move(rcvr)` or with `set_error(std::move(rcvr), error)`, i.e., the expression either results in the coroutine to be never resumed or an exception being thrown.

Here is an example which summaries the different supported result types:

```
lazy<> fun() {
    co_await ex::just();                               // void
    auto v = co_await ex::just(0);                     // int
    auto[i, b, c] = co_await ex::just(0, true, 'c');   // tuple<int, bool, char>
    try { co_await ex::just_error(0); } catch (int) {} // exception
    co_await ex::just_stopped();                       // cancel: never resumed
}
```

The sender `sndr` can have at most one `set_value_t` completion signature: if there are more than one `set_value_t` completion signatures `as_awaitable(sndr)` is invalid and fails to compile: users who want to `co_await` a sender with more than one `set_value_t` completions need to use `co_await into_variant(s)` (or similar) to transform the completion signatures appropriately. It would be possible to move this transformation into `as_awaitable(sndr)`.

Using effectively `into_variant(s)` isn't the only possible transformation if there are multiple `set_value_t` transformations. To avoid creating a fairly hard to use result object, `as_awaitable(sndr)` could detect certain usage patterns and rather create a result which is easier to use when being `co_await`ed. An example for this situation is the `queue.async_pop()` operation for concurrent queues: this operation can complete successfully in two ways:

1. When an object was extracted the operation completes with `set_value(std::move(rcvr), value)`.
2. When the queue was closed the operation completes with `set_value(std::move(rcvr))`.

Turning the result of `queue.async_pop()` into an awaitable using the current `as_awaitable(queue.async_pop())` ([exec.as.awaitable]) fails because the function accepts only senders with at most one `set_value_t` completion. Thus, it is necessary to use something like the below:

```
    lazy<> pop_demo(auto& queue) {
        // auto value = co_await queue.async_pop(); // doesn't work
        std::optional v0 = co_await (queue.async_pop() | into_optional);
        std::optional v1 = co_await into_optional(queue.async_pop());
    }
```

The algorithm `into_optional(sndr)` would determine that there is exactly one `set_value_t` completion with arguments and produce an `std::optional<T>` if there is just one parameter of type `T` and produce a `std::optional<std::tuple<T...>>` if there are more than one parameter with types `T...`. It would be possible to apply this transformation when a corresponding set of completions is detected. The proposal optional variants in sender/receiver goes into this direction.

This proposal currently doesn't propose a change to `as_awaitable` ([exec.as.awaitable]). The primary reason is that there are likely many different shapes of completions each with a different desirable transformation. If these are all absorbed into `as_awaitable` it is likely fairly hard to reason what exact result is returned. Also, there are likely different options of how a result could be transformed: `into_optional` is just one example. It could be preferable to turn the two results into an `std::expected` instead. However, there should probably be some transformation algorithms like `into_optional`, `into_expected`, etc. similar to `into_variant`.

4.5 Scheduler Affinity

Coroutines look very similar to synchronous code with a few `co`-keywords sprinkled over the code. When reading such code the expectation is typically that all code executes on the same context despite some `co_await` expressions using senders which may explicitly change the scheduler. There are various issues when using `co_await` naïvely:

— Users may expect that work continues on the same context where it was started. If the coroutine simply resumes when the `co_await`ed senders calls a completion function code may execute some lengthy operation on a context which is expected to keep a UI responsive or which is meant to deal with I/O.
— Conversely, running a loop `co_await`ing some work may be seen as unproblematic but may actually easily cause a stack overflow if `co_await`ed work immediately completes (also see below).
— When `co_await`ing some work completes on a different context and later a blocking call is made from the coroutine which also ends up `co_await`ing some work from the same resource there can be a dead lock.

Thus, the execution should normally be scheduled on the original scheduler: doing so can avoid the problems mentioned above (assuming a scheduler is used which doesn't immediately complete without actually scheduling anything). This transfer of the execution with a coroutine is referred to as *scheduler affinity*. Note: a scheduler may execute on multiple threads, e.g., for a pool scheduler: execution would get to any of these threads, i.e., thread local storage is *not* guaranteed to access the same data even with scheduler affinity. Also, scheduling work has some cost even if this cost can often be fairly small.

The basic idea for scheduler affinity consists of a few parts:

1. A scheduler is determined when `start`ing an operation state which resulted from `connect`ing a coroutine to a receiver. This scheduler is used to resume execution of the coroutine. The scheduler is determined based on the receiver `rcvr`'s environment.

   ```
      auto scheduler = get_scheduler(get_env(rcvr));
   ```

2. The type of `scheduler` is unknown when the coroutine is created. Thus, the coroutine implementation needs to operate in terms of a scheduler with a known type which can be constructed from `scheduler`. The used scheduler type is determined based on the context parameter `C` of the coroutine type `lazy<T, C>` using

typename C::scheduler_type and defaults to any_scheduler if this type isn't defined. any_scheduler uses type-erasure to deal with arbitrary schedulers (and small object optimisations to avoid allocations). The used scheduler type can be parameterised to allow use of lazy contexts where the scheduler type is known, e.g., to avoid the costs of type erasure.

3. When an operation which is co_awaited completes the execution is transferred to the held scheduler using continues_on. Injecting this operation into the graph can be done in the promise type's await_transform:

```
template <ex::sender Sender>
auto await_transform(Sender&& sndr) noexcept {
    return ex::as_awaitable_sender(
        ex::continues_on(std::forward<Sender>(sndr),
                         this->scheduler);
    );
}
```

There are a few immediate issues with the basic idea:

1. What should happen if there is no scheduler, i.e., get_scheduler(get_env(rcvr)) doesn't exist?
2. What should happen if the obtained scheduler is incompatible with the coroutine's scheduler?
3. Scheduling isn't free and despite the potential problems it should be possible to use lazy without scheduler affinity.
4. When operations are known to complete inline the scheduler isn't actually changed and the scheduling operation should be avoided.
5. It should be possible to explicitly change the scheduler used by a coroutine from within this coroutine.

All of these issues can be addressed although there are different choices in some of these cases.

In many cases the receiver can provide access to a scheduler via the environment query. An example where no scheduler is available is when starting a task on a counting_scope. The scope doesn't know about any schedulers and, thus, the receiver used by counting_scope when connecting to a sender doesn't support the get_scheduler query, i.e., this example doesn't work:

```
ex::spawn([]->ex::lazy<void> { co_await ex::just(); }(), token);
```

Using spawn() with coroutines doing the actual work is expected to be quite common, i.e., it isn't just a theoretical possibility that lazy is used together with counting_scope. The approach used by unifex is to fail compilation when trying to connect a Task to a receiver without a scheduler. The approach taken by stdexec is to keep executing inline in that case. Based on the experience that silently changing contexts within a coroutine frequently causes bugs it seems failing to compile is preferable.

Failing to construct the scheduler used by a coroutine with the scheduler obtained from the receiver is likely an error and should be addressed by the user appropriately. Failing to compile is seems to be a reasonable approach in that case, too.

It should be possible to avoid scheduler affinity explicitly to avoid the cost of scheduling. Users should be very careful when pursuing this direction but it can be a valid option. One way to achieve that is to create an "inline scheduler" which immediately completes when it is start()ed and using this type for the coroutine. Explicitly providing a type inline_scheduler implementing this logic could allow creating suitable warnings. It would also allow detecting that type in await_transform and avoiding the use of continues_on entirely.

When operations actually don't change the scheduler there shouldn't be a need to schedule them again. In these cases it would be great if the continues_on could be avoided. At the moment there is no way to tell whether a sender will complete inline. Using a sender query which determines whether a sender always completes inline could avoid the rescheduling. Something like that is implemented for unifex: senders define a property blocking which can have the value blocking_kind::always_inline. The proposal A sender query for completion behaviour proposes a get_completion_behaviour(sndr, env) customisation point to address this need. The result can indicate that the sndr returns synchronously (using completion_behaviour::synchronous or completion_behaviour::inline_completion). If sndr returns synchronously there isn't a need to reschedule it.

In some situations it is desirable to explicitly switch to a different scheduler from within the coroutine and from then on carry on using this scheduler. unifex detects the use of co_await schedule(scheduler); for this purpose. That is, however, somewhat subtle. It may be reasonable to use a dedicated awaiter for this purpose and use, e.g.

```
auto previous = co_await co_continue_on(new_scheduler);
```

Using this statement replaces the coroutine's scheduler with the new_scheduler. When the co_await completes it is on new_scheduler and further co_await operations complete on new_scheduler. The result of co_awaiting co_continue_on is the previously used scheduler to allow transfer back to this scheduler. In stdexec the corresponding operation is called reschedule_coroutine.

Another advantage of scheduling the operations on a scheduler instead of immediately continuing on the context where the operation completed is that it helps with stack overflows: when scheduling on a non-inline scheduler the call stack is unwound. Without that it may be necessary to inject scheduling just for the purpose of avoiding stack overflow when too many operations complete inline.

4.6 Allocator Support

When using coroutines at least the coroutine frame may end up being allocated on the heap: the HALO optimisations aren't always possible, e.g., when a coroutine becomes a child of another sender. To control how this allocation is done and to support environments where allocations aren't possible lazy should have allocator support. The idea is to pick up on a pair of arguments of type std::allocator_arg_t and an allocator type being passed and use the corresponding allocator if present. For example:

```
struct allocator_aware_context {
    using allocator_type = std::pmr::polymorphic_allocator<std::byte>;
};

template <typename...A>
ex::lazy<int, allocator_aware_context> fun(int value, A&&...) {
    co_return value;
}

int main() {
    // Use the coroutine without passing an allocator:
    ex::sync_wait(fun(17));

    // Use the coroutine with passing an allocator:
    using allocator_type = std::pmr::polymorphic_alloctor<std::byte>;
    ex::sync_wait(fun(17, std::allocator_arg, allocator_type()));
}
```

The arguments passed when creating the coroutine are made available to an operator new of the promise type, i.e., this operator can extract the allocator, if any, from the list of parameters and use that for the purpose of allocation. The matching operator delete gets passed only the pointer to release and the originally requested size. To have access to the correct allocator in operator delete the allocator either needs to be stateless or a copy needs to be accessible via the pointer passed to operator delete, e.g., stored at the offset size.

To avoid any cost introduced by type erasing an allocator type as part of the lazy definition the expected allocator type is obtained from the context argument C of lazy<T, C>:

```
using allocator_type = ex::allocator_of_t<C>;
```

This using alias uses typename C::allocator_type if present or defaults to std::allocator<std::byte>

otherwise. This `allocator_type` has to be for the type `std::byte` (if necessary it is possible to relax that constraint).

The allocator used for the coroutine frame should also be used for any other allocators needed for the coroutine itself, e.g., when type erasing something needed for its operation (although in most cases a small object optimisation would be preferable and sufficient). Also, the allocator should be made available to child operations via the respective receiver's environment using the `get_allocator` query. The arguments passed to the coroutine are also available to the constructor of the promise type (if there is a matching on) and the allocator can be obtained from there:

```
struct allocator_aware_context {
    using allocator_type = pmr::polymorphic_allocator<std::byte>;
};
fixed_resource<2048> resource;

ex::sync_wait([](auto&&, auto* resource)
        -> ex::lazy<void, allocator_aware_context> {
    auto alloc = co_await ex::read_env(ex::get_allocator);
    use(alloc);
}(allocator_arg, &resource));
```

4.7 Environment Support

When `co_await`ing child operations these may want to access an environment. Ideally, the coroutine would expose the environment from the receiver it gets `connect`ed to. Doing so isn't directly possible because the coroutine types doesn't know about the receiver type which in turn determines the environment type. Also, the queries don't know the type they are going to return. Thus, some extra mechanisms are needed to provide an environment.

A basic environment can be provided by some entities already known to the coroutine, though:

— The `get_scheduler` query should provide the scheduler maintained for scheduler affinity whose type is determined based on the coroutine's context using `ex::scheduler_of_t<C>`.
— The `get_allocator` query should provide the coroutine's allocator whose type is determined based on the coroutine's context using `ex::allocator_of_t<C>`. The allocator gets initialised when constructing the promise type.
— The `get_stop_token` query should provide a stop token from a stop source which is linked to the stop token obtained from the receiver's environment. The type of the stop source is determined from the coroutine's context using `ex::stop_source_of_t<C>` and defaults to `ex::inplace_stop_source`. Linking the stop source can be delayed until the first stop token is requested or omitted entirely if `stop_possible()` returns `false` or if the stop token type of the coroutine's receiver matches that of `ex::stop_source_of_t<C>`.

For any other environment query the context `C` of `lazy<T, C>` can be used. The coroutine can maintain an instance of type `C`. In many cases queries from the environment of the coroutine's `receiver` need to be forwarded. Let `env` be `get_env(receiver)` and `Env` be the type of `env`. `C` gets optionally constructed with access to the environment:

1. If `C::env_type<Env>` is a valid type the coroutine state will contain an object `own_env` of this type which is constructed with `env`. The object `own_env` will live at least as long as the `C` object maintained and `C` is constructed with a reference to `own_env`, allowing `C` to reference type-erased representations for query results it needs to forward.
2. Otherwise, if `C(env)` is valid the `C` object is constructed with the result of `get_env(receiver)`. Constructing the context with the receiver's environment provides the opportunity to store whatever data is needed from the environment to later respond to queries as well.
3. Otherwise, `C` is default constructed. This option typically applies if `C` doesn't need to provide any environment queries.

Any query which isn't provided by the coroutine but is available from the context `C` is forwarded. Any other query shouldn't be part of the overload set.

For example:

```
struct context {
    int value{};
    int query(get_value_t const&) const noexcept { return this->value; }
    context(auto const& env): value(get_value(env)) {}
};

int main() {
    ex::sync_wait(
        ex::write_env(
            []->demo::lazy<void, context> {
                auto sched(co_await ex::read_env(get_scheduler));
                auto value(co_await ex::read_env(get_value));
                std::cout << "value=" << value << "\n";
                // ...
            }(),
            ex::make_env(get_value, 42)
        )
    );
}
```

4.8 Support For Requesting Cancellation/Stopped

When a coroutine task executes the actual work it may listen to a stop token to recognise that it got canceled. Once it recognises that its work should be stopped it should also complete with `set_stopped(rcvr)`. There is no special syntax needed as that is the result of using `just_stopped()`:

```
co_await ex::just_stopped();
```

The sender `just_stopped()` completes with `set_stopped()` causing the coroutine to be canceled. Any other sender completing with `set_stopped()` can also be used.

4.9 Error Reporting

The sender/receiver approach to error reporting is for operations to complete with a call to `set_error(rcvr, err)` for some receiver object `rcvr` and an error value `err`. The details of the completions are used by algorithms to decide how to proceed. For example, if any of the senders of `when_all(sndr...)` fails with a `set_error_t` completion the other senders are stopped and the overall operation fails itself forwarding the first error. Thus, it should be possible for coroutines to complete with a `set_error_t` completion. Using a `set_value_t` completion using an error value isn't quite the same as these are not detected as errors by algorithms.

The error reporting used for `unifex` and stdexec is to turn an exception escaping from the coroutine into a `set_error_t(std::exception_ptr)` completion: when `unhandled_exception()` is called on the promise type the coroutine is suspended and the function can just call `set_value(r, std::get_current_exception())`. There are a few limitations with this approach:

1. The only supported error completion is `set_error_t(std::exception_ptr)`. While the thrown exception can represent any error type and `set_error_t` completions from `co_await`ed operations resulting in the corresponding error being thrown it is better if the other error types can be reported, too.
2. To report an error an exception needs to be thrown. In some environments it is preferred to not throw exception or exceptions may even be entirely banned or disabled which means that there isn't a way to report errors from coroutines unless a different mechanism is provided.
3. To extract the actual error information from `std::exception_ptr` the exception has to be rethrown.

4. The completion signatures for `lazy<T, C>` necessarily contain `set_error_t(std::exception_ptr)` which is problematic when exceptions are unavailable: `std::exception_ptr` may also be unavailable. Also, without exception as it is impossible to decode the error. It can be desirable to have coroutine which don't declare such a completion signature.

Before going into details on how errors can be reported it is necessary to provide a way for `lazy<T, C>` to control the error completion signatures. Similar to the return type the error types cannot be deduced from the coroutine body. Instead, they can be declared using the context type `C`:

— If present, `typename C::error_signatures` is used to declare the error types. This type needs be a specialisation of `completion_signatures` listing the valid `set_error_t` completions.
— If this nested type is not present, `completion_signatures<set_error_t(std::exception_ptr)>` is used as a default.

The name can be adjusted and it would be possible to use a different type list template and listing the error types. The basic idea would remain the same, i.e., the possible error types are declared via the context type.

Reporting an error by having an exception escape the coroutine is still possible but it doesn't necessarily result in a `set_error_t`: If an exception escapes the coroutine and `set_error_t(std::exception_ptr)` isn't one of the supported the `set_error_t` completions, `std::terminate()` is called. If an error is explicitly reported somehow, e.g., using one of the approaches described below, and the error type isn't supported by the context's `error_signatures`, the program is ill-formed.

The discussion below assumes the use of the class template `with_error<E>` to indicate that the coroutine completed with an error. It can be as simple as

```
template <typename E> struct with_error{ E error; };
```

The name can be different although it shouldn't collide with already use names (like `error_code` or `upon_error`). Also, in some cases there isn't really a need to wrap the error into a recognisable class template. Using a marker type probably helps with readability and avoiding ambiguities in other cases.

Besides exceptions there are three possible ways how a coroutine can be exited:

1. The coroutine is exited when using `co_return`, optionally with an argument. Flowing off the end of a coroutine is equivalent to explicitly using `co_return;` instead of flowing off. It would be possible to turn the use of

   ```
   co_return with_error{err};
   ```

   into a `set_error(std::move(rcvr), err)` completion.

   One restriction with this approach is that for a `lazy<void, C>` the body can't contain `co_return with_error{e};`: the `void` result requires that the promise type contains a function `return_void()` and if that is present it isn't possible to also have a `return_value(T)`.

2. When a coroutine uses `co_await a;` the coroutine is in a suspended state when `await_suspend(...)` of some awaiter is entered. While the coroutine is suspended it can be safely destroyed. It is possible to complete the coroutine in that state and have the coroutine be cleaned up. This approach is used when the awaited operation completes with `set_stopped()`. It is possible to call `set_error(std::move(rcvr), err)` for some receiver `rcvr` and error `err` obtained via the awaitable `a`. Thus, using

   ```
   co_await with_error{err};
   ```

   could complete with `set_error(std::move(rcvr), err)`.

   Using the same notation for awaiting outstanding operations and returning results from a coroutine is, however, somewhat surprising. The name of the awaiter may need to become more explicit like `exist_coroutine_with_error` if this approach should be supported.

3. When a coroutine uses `co_yield v;` the promise member `yield_value(T)` is called which can return an awaiter `a`. When `a`'s `await_suspend()` is called, the coroutine is suspended and the operation can complete accordingly. Thus, using

    ```
    co_yield with_error{err};
    ```

    could complete with `set_error(std::move(rcvr), err)`. Using `co_yield` for the purpose of returning from a coroutine with a specific result seems more expected than using `co_await`.

There are technically viable options for returning an error from a coroutine without requiring exceptions. Whether any of them is considered suitable from a readability point of view is a separate question.

One concern which was raised with just not resuming the coroutine is that the time of destruction of variables used by the coroutine is different. The promise object can be destroyed before completing which might address the concern.

Using `co_await` or `co_yield` to propagate error results out of the coroutine has a possibly interesting variation: in both of these case the error result may be conditionally produced, i.e., it is possible to complete with an error sometimes and to produce a value at other times. That could allow a pattern (using `co_yield` for the potential error return):

```
auto value = co_yield when_error(co_await into_expected(sender));
```

The subexpression `into_expected(sender)` could turn the `set_value_t` and `set_error_t` into a suitable `std::expected<V, std::variant<E...>>` always reported using a `set_value_t` completion (so the `co_await` doesn't throw). The corresponding `std::expected` becomes the result of the `co_await`. Using `co_yield` with `when_error(exp)` where `exp` is an expected can then either produce `exp.value()` as the result of the `co_yield` expression or it can result in the coroutine completing with the error from `exp.error()`. Using this approach produces a fairly compact approach to propagating the error retaining the type and without using exceptions.

4.10 Avoiding Stack Overflows

It is easy to use a coroutine to accidentally create a stack overflow because loops don't really execute like loops. For example, a coroutine like this can easily result in a stack overflow:

```
ex::sync_wait(ex::write_env(
    []() -> ex::lazy<void> {
        for (int i{}; i < 1000000; ++i)
            co_await ex::just(i);
    }(),
    ex::make_env(ex::get_scheduler, ex::inline_scheduler{})
));
```

The reason this innocent looking code creates a stack overflow is that the use of `co_await` results in some function calls to suspend the coroutine and then further function calls to resume the coroutine (for a proper explanation see, e.g., Lewis Baker's Understanding Symmetric Transfer). As a result, the stack grows with each iteration of the loop until it eventually overflows.

With senders it is also not possible to use symmetric transfer to combat the problem: to achieve the full generality and composing senders, there are still multiple function calls used, e.g., when producing the completion signal. Using `get_completion_behaviour` from the proposal A sender query for completion behaviour could allow detecting senders which complete synchronously. In these cases the stack overflow could be avoided relying on symmetric transfer.

When using scheduler affinity the transfer of control via a scheduler which doesn't complete immediately does avoid the risk of stack overflow: even when the `co_await`ed work immediately completes as part of the `await_suspend` call of the created awaiter the coroutine isn't immediately resumed. Instead, the work is scheduled and the coroutine is suspended. The thread unwinds its stack until it reaches its own scheduling and picks up the next entity to execute.

When using `sync_wait(sndr)` the `run_loop`'s scheduler is used and it may very well just resume the just suspended coroutine: when there is scheduling happening as part of scheduler affinity it doesn't mean that work gets scheduled on a different thread!

The problem with stack overflows does remain when the work resumes immediately despite using scheduler affinity. That may be the case when using an inline scheduler, i.e., a scheduler with an operation state whose `start()` immediately completes: the scheduled work gets executed as soon as `set_value(std::move(rcvr))` is called.

Another potential for stack overflows is when optimising the behaviour for work which is known to not move to another scheduler: in that case there isn't really any need to use `continue_on` to get back to the scheduler where the operation was started! The execution remained on that scheduler all along. However, not rescheduling the work means that the stack isn't unwound.

Since `lazy` uses scheduler affinity by default, stack overflow shouldn't be a problem and there is no separate provision required to combat stack overflow. If the implementation chooses to avoid rescheduling work it will need to make sure that doing so doesn't cause any problems, e.g., by rescheduling the work sometimes. When using an inline scheduler the user will need to be very careful to not overflow the stack or cause any of the various other problems with executing immediately.

4.11 Asynchronous Clean-Up

Asynchronous clean-up of objects is an important facility. Both `unifex` and stdexec provide some facilities for asynchronous clean-up in their respective coroutine task. Based on the experience the recommendation is to do something different!

The recommended direction is to support asynchronous resources independent of a coroutine task. For example the async-object proposal is in this direction. There is similar work ongoing in the context of Folly. Thus, there is currently not plan to support asynchronous clean-up as part of the `lazy` implementation. Instead, it can be composed based on other facilities.

5 Caveats

The use of coroutines introduces some issues which are entirely independent of how specific coroutines are defined. Some of these were brought up on prior discussions but they aren't anything which can be solved as part of any particular coroutine implementation. In particular:

1. As `co_await`ing the result of an operation (or `co_yield`ing a value) may suspend a coroutine, there is a potential to introduce problems when resources which are meant to be held temporarily are held when suspending. For example, holding a lock to a mutex while suspending a coroutine can result in a different thread trying to release the lock when the coroutine is resumed (scheduler affinity will move the resumed coroutine to the same scheduler but not to the same thread).
2. Destroying a coroutine is only safe when it is suspended. For the task implementation that means that it shall only call a completion handler once the coroutine is suspended. That part is under the control of the coroutine implementation. However, there is no way to guard against users explicitly destroying a coroutine from within its implementation or from another thread while it is not suspended: that's akin to destroying an object while it being used.
3. Debugging asynchronous code doesn't work with the normal approaches: there is generally no suitable stack as work gets resumed from some run loop which doesn't tell what set up the original work. To improve on this situation, *async stack traces* linking different pieces of outstanding work together can help. At CppCon 2024 Ian Petersen and Jessica Wong presented how that may work (watch the video). Implementations should consider adding corresponding support and enhance tooling, e.g., debuggers, to pick up on async stack traces. However, async stack support itself isn't really something which one coroutine implementation can enable.

While these issues are important this proposal isn't the right place to discuss them. Discussion of these issues should be delegated to suitable proposals wanting to improve this situation in some form.

6 Questions

This section lists questions based on the design discussion above. Each one has a recommendation and a vote is only needed if there opinions deviating from the recommendation.

— Result type: expand `as_awaitable(sndr)` to support more than one `set_value_t(T...)` completion? Recommendation: no.
— Result type: add transformation algorithms like `into_optional`, `into_expected`? Recommendation: no, different proposals.
— Scheduler affinity: should `lazy` support scheduler affinity? Recommendation: yes.
— Scheduler affinity: require a `get_scheduler()` query on the receiver's environments? Recommendation: yes.
— Scheduler affinity: add a definition for `inline_scheduler` (using whatever name) to support disabling scheduler affinity? Recommendation: yes.
— Allocator support: should `lazy` support allocators (default `std::allocator<std::byte>`)? Recommendation: yes.
— Error reporting: should it be possible to return an error without throwing an exception? Recommendation: yes.
— Error reporting: how should errors be reported? Recommendation: using 'co_yield with_error(e).
— Error reporting: should `co_yield when_error(expected)` be supported? Recommendation: yes (although weakly).
— Clean-up: should asynchronous clean-up be supported? Recommendation: no.

7 Implementation

An implementation of `lazy` as proposed in this document is available from `beman::lazy`. This implementation hasn't received much use, yet, as it is fairly new. It is setup to be buildable and provides some examples as a starting point for experimentation.

Coroutine tasks very similar although not identical to the one proposed are used in multiple projects. In particular, there are three implementations in wide use:

— `Folly::Task`
— `unifex::Task`
— `stdexec::task`

The first one (`Folly::Task`) isn't based on sender/receiver. Usage experience from all three have influenced the design of `lazy`.

8 Acknowledgements

We would like to thank Ian Petersen, Alexey Spiridonov, and Lee Howes for comments on drafts of this proposal and general guidance.

9 Proposed Wording

The intent is to have all relevant wording in place before the Hagenberg meeting.

Based on the discussion the wording would get Entities to describe:

— `inline_scheduler`
— `any_scheduler`
— `lazy`
— any internally used tool
— allocator_of_t exposition-only?
— scheduler_of_t exposition-only?

— stop_source_of_t exposition-only?