It’s Scopes All the Way Down

Document Number: P3955R0
Date: 2026-01-14
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG, SG1

Abstract

This paper lays out an approach to achieving the same effect as regular, synchronous scopes in the asynchronous domain (i.e. `std::execution` [1]), including asynchronous construction and destruction of objects.

Background

Even C has scopes. Unlike C++, however, nothing (except consuming and allocating stack space) can be accomplished on their boundaries.

The power of C++ constructors and destructors, referred to as “RAII” (even when resources are not being acquired) is the ability to inject arbitrary code into the rising and falling edges of a scope. This not only trivializes tasks which are error prone in languages such as C (allocating and freeing memory, for example) but also allows for a wide range of creative and heterodox applications.

Unfortunately C++ is a fundamentally synchronous language. Entering and leaving a scope, and therefore the power of RAII, is only available synchronously. Even where C++ attempts to be an asynchronous language it does not extend the power of its scopes thereto, saying instead (§11.4.5.1 [class.ctor.general]):

“A constructor shall not be a coroutine.”

And (§11.4.7 [class.dtor]):

“A destructor shall not be a coroutine.”

But C++ is more than a language, it is also a library. So much so that C++ refuses fundamental features of other languages, leaving them instead to the library. Sum and product types, for example. An asynchronous simulacrum of itself, as another ([2] at §3):

“It is becoming apparent that all the sender/receiver features are language features being implemented in library.”

Out of the box in C++26 this simulacrum, `std::execution`, featured an upgraded version of the C++ language construct of a function (ibid.):

"Sender/Receiver itself is the implementation of an `async-function`*."*

Unlike the built in function call protocol the asynchronous operations of `std::execution` are:

- Fundamentally asynchronous,
- Integrate cancellation as a first-class completion [3], and
- Offer more flexible representations of completions, including homogeneous completions for both the value and error channels, and value completions which send multiple types (i.e. explicit sum and product types are not needed in the completion signatures of a sender)

This however just projects functions into the asynchronous domain. C has functions, but does not have RAII. `malloc` and `free` are functions, `unique_ptr` is something else: An object.

To evolve its asynchronous ecosystem and achieve parity with its synchronous ecosystem C++ needs not only asynchronous functions, but also asynchronous objects [2].

Discussion

Prior Art

Kirk Shoop has previously explored this space [2]. With his permission the author of this paper is continuing that exploration.

What Is an "Asynchronous Object?"

A synchronous object is an object whose lifetime begins via a synchronous function (i.e. a constructor) and ends via a synchronous function (i.e. a destructor). Similarly ([2] at §1.1):

"An `async-object` *is an object [...] that has* `async-function`*s to construct and destruct the object."*

Put differently: An asynchronous object is an object whose constructor and destructor are asynchronous.

Why Are Asynchronous Objects Needed?

With the idea of an asynchronous object described the next question is: Why is such a concept needed or useful? Are asynchronous constructors and destructors really needed? Are they something users of C++ would ever require or reach for?

Consider an object which represents and manages a TCP connection to a server. If it has an invariant that it is connected (and that it is an error for it not to be) then it cannot be said to be fully constructed until the asynchronous process of connecting (and perhaps resolving a DNS address or trying other endpoints) has completed. Two phase init could clearly be used here, but that is widely regarded as an antipattern in general (the Google style guide, for example, has historically been much-maligned for featuring/requiring it). Therefore an asynchronous constructor (or other factory) is needed.

But the above-described object still has a synchronous destructor: The POSIX `close` system call on the file descriptor representing the connected socket. However the existence of io_uring, a fundamentally asynchronous way to make Linux system calls, muddies these waters as it features an asynchronous equivalent even of `close`: `IORING_OP_CLOSE`.

POSIX system calls and io_uring are beyond the purview of the standard, however, and for justification of asynchronous destructors we need not look that far. C++26 actually contains such a justification: Async scopes [4]. Consider the functionality of the synchronous destructor of `execution::simple_counting_scope` (§33.14.2.2.2 [exec.simple.counting.ctor]):

"If `state` *is not one of* `joined`*,* `unused`*, or* `unused-and-closed`*, invokes* `terminate`*. Otherwise, has no effects."*

The scope is transitioned to the *`joined`* state by obtaining a sender via the scope's `join` member function, connecting it, starting the resulting operation state, and allowing that asynchronous operation to run to completion. That is to say: `execution::simple_counting_scope` has manual, asynchronous clean up which must be performed before the object can be allowed to go out of scope. Clean up is the raison d'être of RAII and one of C++'s greatest strengths, but because C++ doesn't have asynchronous RAII the clean up story for this type is more reminiscent of C: Make sure you follow the right idiom, make sure you don't forget, make sure whoever is maintaining the code after or with you does the same.

A narrow solution to the specific problem above has been proposed: `execution::let_async_scope` [5] (this will be returned to later in this paper), but we can do better. We already have the principle of RAII which addresses this class of problem, we just need to project it into the asynchronous domain.

Prior Design

The prior design (from [2]) centered around types which modelled the `execution::async_object` and `::async_object_constructible_from` concepts. Note that a type which modelled the async object concept was not itself the object which would be created or interacted with, but rather a factory which could be used to produce an object. One can think of an async object in this design as a partially-curried, but asynchronous, version of a constructor, in the same way a sender is a fully curried, but asynchronous, version of a function.

An async object was required to be shaped in the following way (id. at §4.2.0.1):

```
struct async_object {
  using object = /* ... */;
  using handle = /* ... */;
  using storage = /* ... */;
  execution::sender auto async_destruct(storage&) const noexcept;
  template<typename... Args>
  execution::sender auto async_construct(storage&, Args&&...) const;
};
```

With:

- `object` being the actual type of the object to asynchronous construct and destroy,
- `handle` being the type through which above-mentioned object will be accessed (i.e. it is the result of async construction), and
- `storage` being the type which provides storage for the object

The utility `execution::make_packaged_async_object` was proposed (id. at §4.2.0.6) to curry arguments into an async object thereby rendering it “default async-constructible” (i.e. having an `async_construct` member function which accepts no trailing arguments). This meant that the above-described async object design followed the model of iterative currying already embodied by senders and receivers wherein:

1. Arguments are applied into a sender by a sender factory, and then
2. A receiver is applied into an operation state by `execution::connect`

As follows:

1. Arguments are applied into an async object to yield an async object whose `async_construct` member function is unary (i.e. requires only storage),
2. Storage is applied into an async object to yield a sender, and then
3. A receiver is applied into an operation state by `execution::connect`

This applicative model is powerful because it allows elegant separation of concerns, as evidenced by the generic algorithms of `std::execution`.

Issues With the Prior Design

Reference Qualification

The most serious issue with the previous design is that it requires that `async_construct` and `async_destruct` be invocable on const lvalue async objects. This means that either:

- Curried arguments must be copyable into the senders yielded by the above-mentioned member functions (so they can be propagated from a const object), or
- The async object must remain within its lifetime until the completion of asynchronous construction and destruction thereof

Even the latter of the two above possibilities, which has the fewest performance implications, has a problem. Even if the async object remains within its lifetime, thereby allowing the constructor and destructor senders (and their resulting asynchronous operations) to refer to the contents thereof there's still no way to use those contents consumptively (e.g. by moving therefrom) because the capture occurred from a const-qualified object.

It might initially seem that the design can easily be extended to rectify the above: Simply cause the `async_object` concept to check for invocability of `async_construct` and `async_destruct` with the const-ness and reference-ness of whatever type is provided (for example `execution::sender_to` does this with connectability). However the protocol consists of two member function invocations: Both `async_construct` and `async_destruct` must be called for the corresponding object to be created and destroyed. This means that if `async_construct` is called on an rvalue-qualified object there must be a subsequent invocation of `async_destruct` which might then arguably constitute a use after move.

Curried or Not?

Because of the unfixed arity of `async_construct` (i.e. it may have trailing arguments after the reference to the storage) the previous design provides two concepts: `execution::async_object` and `::async_object_constructible_from` with only the latter checking the entire protocol. I.e. it is possible to satisfy `execution::async_object` and have no `async_construct` member function at all, because it's impossible to check the `async_construct` member function without knowing which arguments you expect to invoke it with (in some sense this is similar to `execution::sender` and `::sender_in`).

An advantage of the above-described design is that it mirrors regular, synchronous C++ objects. Such objects can have many constructors, callable in many different ways. But there's an important difference between regular, synchronous objects and async objects: An async object

is not the object it constructs. The async object represents an indirection, hence the need for the `object` nested type.

The above formulation creates blurred categories: An arbitrary async object is an async object, but so is the async object created by `execution::packaged_async_object`. Despite being in the same category they don't support the same operations, and therefore can't be interacted with generically.

This creates a question with respect to the single responsibility principle: What is the responsibility of the caller of `async_construct`? Is it to manage the lifetime of a certain object, or is it to select and apply the arguments which will select the object to be constructed?

`std::execution` itself already contains an example of an alternate design: The arguments to the asynchronous function call represented by a sender are provided to a sender factory which generates a sender. This sender is isomorphic to a fully-curried synchronous function. There is a single responsibility in selecting the arguments. Elsewhere, in another domain of responsibility, that sender is connected and started. This decomposition is foundational to the generic power of `std::execution` and its asynchronous algorithms and should be reflected in the design of asynchronous RAII.

Object Primacy

Constructors and destructors are fundamentally associated with objects (since they respectively start and end the lifetimes thereof). Objects have lifetimes which are associated with scopes (note that a scope may be lexical or more abstract). Therefore constructors and destructors are ways to run code when a scope is entered or exited, however they're still coupled to an object. This friction can be seen in the idiom of the "scope guard" [6] wherein an object is synthesized solely to perform some action when its lifetime ends. In some sense this "scope guard" object is not a "real" object and is instead an abuse of objects to instrument the exit from a scope, since C++ does not have a native way to access such functionality.

The prior design of async RAII too engages in the above-described object primacy: Async objects in that design are always associated with storage thereby marrying the running of code when entering and exiting a scope to the provisioning of storage and the creation of an object.

Given how common the scope guard idiom is in C++ the design of async RAII should not cling to the above-described limitation of C++ RAII in the same way that `std::execution` didn't cling to the limitations on the structure of synchronous function return (i.e. unlike regular, synchronous functions the asynchronous functions can report completion in many ways with each yielding potentially many values).

New Design

Scopes

The fundamental building block of the design proposed by this paper is a scope. As discussed above such scopes are not coupled to storage or objects. Entering a scope requires some action be performed, and leaving the scope also requires some action be performed. Both of these actions are represented by senders and can therefore be performed asynchronously.

Unlike the prior design wherein a caller had immediate access to construction (i.e. entering a scope) and destruction (i.e. leaving a scope) by calling the appropriate member functions of an async object in this design the “enter scope sender” (which performs the action necessary to enter the scope) yields the “exit scope sender” (which performs the action necessary to leave the scope) upon completion, i.e. the enter scope sender is a higher order sender. This avoids the problem of rvalue qualification because the enter scope sender is consumed to form the enter scope operation, which then yields another entity, the exit scope sender, which may then be consumed later in turn when the time comes to leave the scope.

Scope Algorithms

The fundamental scope algorithm is `within`. It establishes a scope by running an enter scope sender. Thereafter it allows a child operation to run (in that scope since the exit scope sender has not yet run), and then before completing the overall operation runs the exit scope sender yielded by the enter scope sender.

Enter scope senders can be composed in one of two ways: In sequence, or in series. These two patterns are reified by two scope sender adaptors:

- `enter_scopes` combines N enter scope senders into an enter scope sender that enters all N scopes in parallel and then yields an exit scope sender that exits all N scopes in parallel
- nest_scopes combines N enter scope senders into an enter scope sender that enters each scope in order and then yields an exit scope sender that exits each scope in reverse order

Note that if any enter scope sender’s operation fails the appropriate exit scope sender’s operation will be synthesized and allowed to run before the overall operation fails.

Objects

Objects combine a scope with storage. Entering the scope constructs an object in the storage, and exiting the scope destroys that object. An async object now has the following interface:

```
struct async_object {
  using type = /* ... */;
  execution::enter_scope_sender auto operator()(type*);
};
```

Note that since this entity has a single basis operation (i.e. invocation) rvalue qualification of invocation is trivially supportable.

The above design also reflects and extends the currying model of `std::execution`. The workflow to run an asynchronous operation consists of:

1. Currying arguments into the sender,
2. Connecting the sender to a receiver, and then
3. Starting the asynchronous operation

Whereas synthesis of an asynchronously-constructed and -destroyed object consists of:

1. Currying constructor arguments into the async object,
2. Currying the storage pointer into the enter scope sender,
3. Connecting the sender to a receiver, and then
4. Starting the asynchronous operation

With only one responsibility present at each step (selection of arguments, followed by selection of storage, followed by selection of continuation).

Object Algorithms

The fundamental async object algorithm is `lifetime`. It accepts N async objects and provides storage for all of them. Once the asynchronous operation is underway it constructs all objects in parallel and then provides references thereto to a user-provided invocable which synthesizes a sender in response (like `execution::let_value` et al.). This sender is connected and started and once it completes the exit scope senders are connected and started to destroy all async objects after which the operation ends.

Commentary on Naming

Maybe we should've called a "scope" [4] a "nursery." The author is not particularly attached to the names in this paper and imagines they will be changed.

Note that none of the names in this paper, unlike the previous one [2], feature the `async_` prefix. This is because the author believes that entities in the `std::execution` namespace are presumptively async [7].

Proposal

This does not purport to be formal wording, it merely aims to describe the proposed design for the purpose of feedback.

Concepts

`exit_scope_sender`

Unary concept. Modeled if the type:

- Models `execution::sender`, and
- Does not throw when decay-copied or moved

`exit_scope_sender_in`

Binary concept. Let the arguments be `Sender` and `Env`. Modeled if:

- `Sender` models `execution::exit_scope_sender`,
- `Sender` models `execution::sender_in<Env>`,
- `Sender` is nothrow connectable to any receiver with environment type `Env` [8], and
- `execution::completion_signatures_of_t<Sender, Env>` is `execution::completion_signatures<execution::set_value_t()>`

`enter_scope_sender`

Equivalent to `execution::sender`. Without an environment nothing further can be gleaned.

`enter_scope_sender_in`

Binary concept. Let the arguments be `Sender` and `Env`. Modeled if:

- `Sender` models `execution::enter_scope_sender`,
- `Sender` models `execution::sender_in<Env>`,
- `execution::completion_signatures_of_t<Sender, Env>` contains exactly one value completion signature, and that completion signature is unary, and the type of the sole argument models `execution::exit_sender_in<Env>`, and
- As a semantic requirement: The above-mentioned exit sender undoes the actions of the enter scope sender which sent it

`object`

Unary concept. Modeled if the type:

- Has (after removing cv- and ref-qualification) a nested type alias `type` which denotes an object type, and
- Is unary invocable with a pointer to `type` and that invocation yields a type which models `execution::enter_scope_sender`

`object_in`

Binary concept. Let the arguments be `Object` and `Env`. Modeled if:

- `Object` models `execution::object`,
- The unary invocation described in the above-mentioned concept yields a type which models `execution::enter_scope_sender_in<Env>`, and
- As a semantic requirement: The above-mentioned enter sender forms an asynchronous operation which constructs an object of type `type` in the storage indicated by the provided pointer if it completes successfully (note that the fact that the resulting exit sender destroys said object is implicit in the semantic requirements of `execution::enter_scope_sender_in`)

Type Aliases

`exit_scope_sender_of_t`

Binary type alias accepting an enter scope sender and an environment. Yields the type of the exit scope sender which an asynchronous operation formed from that enter scope sender yields on completion.

`type_of_object_t`

Determines the type of object constructed by an async object. Equivalent to `std::remove_cvref_t<T>::type`.

`enter_scope_sender_of_object_t`

Determines the type of enter scope sender yielded by an async object. Equivalent to `std::invoke_result_t<T, execution::type_of_object_t<T>*>`.

Sender Factories

`enter_scopes`

N-ary sender factory which accepts enter scope senders. Result is an enter scope sender which behaves as follows:

When connected: Connects each of the child senders provided to the factory to form child operation states (possibly with a different environment than that of the final receiver to allow for a separate stop source to emit stop requests, see following).

When the resulting operation state is started: Starts each child operation state.

When each child operation completes:

- If a value completion, store the exit scope sender, otherwise
- If this is the first non-value completion, store this completion and request all other children stop (see above)

When all children have completed: After doing the above for the final completing child:

- If no child completed with a non-value completion, complete with `execution::set_value` sending the result of `execution::when_all(...)` (note this relies on LWG4502 being resolved) of all stored exit scope senders (see above), otherwise
- Complete sending the stored non-value completion (see above)

`nest_scopes`

Composes N enter scope senders into a single enter scope sender which enters the scopes represented by the N provided senders in sequence (i.e. the first scope will be entered, and then the second, and so on, with them being exited in the reverse order). More formally:

When connected: Connects each of the child senders provided to the factory to form child operation states.

When the resulting operation state is started: Starts the first child operation state.

When each child operation completes:

- If this is the operation created from an enter scope sender
    - If a value completion, store the exit scope sender and start the next operation state, otherwise
    - Store this completion, connect all stored exit scope senders (see above), and start the final exit scope sender (this runs them in reverse order)
- If this is the operation created from an exit scope sender, start the previous operation state

When all children have completed:

- If no enter scope operation failed, complete with an exit scope sender which is equivalent to `exec::sequence` [9] over all exit scope senders in reverse order, otherwise
- Complete with the stored non-value completion (see above)

`within`

Binary sender factory whose first argument models `execution::enter_sender` and whose second argument models `execution::sender` (referred to below as the “other sender”). Result is a sender which behaves as follows:

When connected: Connects the provided enter sender and decay-copies the other sender into the operation state.

When the resulting operation state is started: Starts the child operation state (see above).

When the enter sender’s operation completes:

- If a non-value completion, complete the overall operation immediately with that completion, otherwise
- Decay-copy the exit sender, optionally move the other sender (perhaps to extract it from the active alternative of a variant), connect the other sender, and start the resulting operation state

When the other sender’s operation completes: Store the completion, optionally move the exit sender (same reason as above), connect the exit sender, and start the resulting operation state.

When the exit sender’s operation completes: Complete sending the stored completion (see above).

`lifetime`

N-ary sender factory whose first argument is an invocable and whose trailing arguments are async objects (see concept above). The invocable must be invocable with `Ts&...` where `Ts...` is a pack containing the type of each async object, respectively (see type alias above). Result is a sender which behaves as follows:

When connected:

1. Invokes all provided async objects with a pointer to appropriate storage within the operation state,
2. Passes all senders synthesized in step 1 as a pack to `execution::enter_scopes`
3. Synthesizes a sender which, when connected and started:
    a. Invokes the invocable with a pack of references, each referring to an object in the above-mentioned storage,
    b. Connects the resulting sender, and then
    c. Starts the resulting operation state
4. Passes the senders synthesized in steps 2 and 3, respectively, to `execution::within`, and then
5. Connects the sender synthesized in step 4

When started: Starts the child operation state.

Note that the receiver provided to this operation's connect function is simply passed through when connecting the `execution::within` sender in step 4.

Classes

`sync_object`

This is a minimal archetype of the `execution::object` & `::object_in` concepts. It simply allows regular C++ objects with regular synchronous constructors and destructors to be managed by `execution::lifetime`.

Example implementation:

```
template<typename T, typename... Args>
  requires
    is_constructible_v<T, Args...> &&
    is_destructible_v<T>
struct sync_object {

  using type = T;

  template<typename... Ts>
    requires (is_constructible_v<Args, Ts> && ...)
  constexpr explicit sync_object(Ts&&... ts) noexcept(
    (is_nothrow_constructible_v<Args, Ts> && ...))
    : args_((Ts&&)ts...)
  {}

  template<typename Self>
  constexpr execution::enter_scope_sender auto operator()(
    this Self&& self,
    type* storage)
      noexcept(
        is_nothrow_constructible_v<
          tuple<Args...>,
          decltype(std::forward_like<Self>(declval<tuple<Args...>&>()>)
  {
      return
        execution::just(std::forward<Self>(self).args_) |
        execution::then([storage](tuple<Args...>&& tuple) noexcept(
          is_nothrow_constructible_v<
```

```
            T,
            Args...>)
        {
          const auto ptr =
            new(storage) T(make_from_tuple<T>(std::move(tuple)));
          return
            execution::just() |
            //  It's important we capture ptr not storage because storage
            //  just points to storage whereas ptr actually points to an
            //  object
            execution::then([ptr]() noexcept {
              ptr->~T();
            });
        });
  }

private:
  tuple<Args...> args_;

};
```

This utility provides a superior alternative to the idiom of `execution::just(...) | execution::let_value(...)` ([10] at 40:06) which is commonly used to place one or more objects in an operation state and thereby attach them to an asynchronous scope. Note that this utility is superior to the aforementioned idiom because:

- It does not require that the object(s) be movable (ibid.)
- It has lower storage requirements in the status quo [11]

Put differently, whereas local variables with automatic storage duration are attached to the scope of the regular, synchronous function which contains them `execution::lifetime(..., execution::sync_object<...>(...))` attaches a variable to the scope of the asynchronous function represented by the sender yielded by the invocable which is the first argument to `execution::lifetime`.

Examples

`let_async_scope`

As mentioned earlier in the paper the join operation of `execution::simple_counting_scope` and `::counting_scope` is simply an asynchronous destructor. There's no need for a new

algorithm `execution::let_async_scope` [5], we can simply expose both of the aforementioned types as async objects:

```
template<typename Scope>
struct scope_object {
  using type = Scope;

  execution::enter_scope_sender auto operator()(type* storage) const noexcept
  {
    return
      execution::just(storage) |
      execution::then([](type* storage) {
        const auto ptr = new(storage) type();
        return
          ptr->join() |
          execution::then([ptr]() noexcept {
            ptr->~type();
          });
      });
  }

};
```

Asynchronous Mutex

A straightforward design of an asynchronous mutex using `std::execution` might look like this ([10] at 29:37):

```
struct async_mutex {
  template<std::execution::sender Sender>
  std::execution::sender auto with(Sender&& sender);
};
```

Notice that the `with` member function, which is a sender factory, isn't nullary. This is because it needs to wrap another operation so that it can:

- Acquire the lock before running that other operation, and
- Release the lock after running that other operation

Notably the shape of the above API makes it difficult (or impossible) to acquire multiple mutexes in parallel. But also note that the above description of the `with` member function mirrors our understanding of a scope exactly: An action taken upon entering the scope, and a corresponding action taken upon leaving the scope.

With the above understanding, and the primitives proposed by this paper, we can reformulate our asynchronous mutex as:

```
struct async_mutex {
  std::execution::enter_scope_sender auto acquire();
};
```

Now the mutex can be acquired and released, and an operation run while holding the lock, using the `within` operation proposed by this paper:

```
async_mutex m;
auto critical_section =
  std::execution::just() |
  std::execution::then([&]() {
    std::cout << "Holding the async mutex" << std::endl;
  });
auto snd = std::execution::within(m.acquire(), std::move(critical_section));
```

But this only mirrors the functionality available with the previous design. What if we wish to acquire and release several mutexes at the same time? With the new design this is trivial:

```
async_mutex a, b, c;
auto critical_section =
  std::execution::just() |
  std::execution::then([&]() {
    std::cout << "Holding all async mutexes" << std::endl;
  });
auto snd = std::execution::within(
  std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
  std::move(critical_section));
```

But note that the above assumes a regular, synchronous, exterior scope in which the mutex objects can safely live. That is, there's an implicit assumption that `snd` will be connected, started, and run to completion before `a`, `b`, or `c` go out of scope. As mentioned in the discussion of the proposed `execution::sync_object`, one way to workaround this in the status quo is with `execution::just(...) | execution::let_value(...)`:

```
auto snd =
  std::execution::just(async_mutex{}, async_mutex{}, async_mutex{}) |
  std::execution::let_value([](
    async_mutex& a, async_mutex& b, async_mutex& c)
  {
    auto critical_section =
      std::execution::just() |
      std::execution::then([&]() {
```

```
      std::cout << "Holding all async mutexes" << std::endl;
    });
  return std::execution::within(
    std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
    std::move(critical_section));
});
```

This places the three asynchronous mutexes in the operation state formed when the sender is connected, thereby ensuring that the mutexes remain valid for the lifetime of the resulting asynchronous operation. Unfortunately however it assumes that `async_mutex` is movable to:

- Store the mutexes in `execution::just`,
- Compose `execution::just` with `execution::let_value`,
- Connect `snd`, and
- Propagate the mutexes from the operation state of `execution::just` to the operation state of `execution::let_value`

Not only is that quite a few moves, but mutexes are generally not movable. In the status quo this would be difficult to workaround but `execution::sync_object` provides a solution:

```
auto snd = std::execution::lifetime(
  [](async_mutex& a, async_mutex& b, async_mutex& c) {
    auto critical_section =
      std::execution::just() |
      std::execution::then([&]() {
        std::cout << "Holding all async mutexes" << std::endl;
      });
    return std::execution::within(
      std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
      std::move(critical_section));
  },
  std::execution::sync_object<async_mutex>{},
  std::execution::sync_object<async_mutex>{},
  std::execution::sync_object<async_mutex>{});
```

This works because `execution::sync_object<async_mutex>{}`, unlike `async_mutex{}`, doesn't actually construct an `async_mutex`. Instead it curries arguments to an asynchronous constructor which doesn't run until the sender yielded by `execution::lifetime` is connected and the resulting operation state is started. Since `execution::lifetime` asynchronously constructs and destroys async objects in storage within its operation state these `async_mutex` objects are never moved and therefore the fact that they are immovable is not an issue.

io_uring Socket Pair

The below example uses io_uring to:

1. Asynchronously create two sockets (via `IORING_OP_SOCKET`),
2. Listen on one of the sockets (via `IORING_OP_BIND` and `IORING_OP_LISTEN`),
3. Connect to the socket from step 2 from the other socket (via `IORING_OP_CONNECT` on the connecting side and `IORING_OP_ACCEPT` on the listening side),
4. Send a string from the socket connected in step 3 (via `IORING_OP_WRITE`),
5. Read that string from the socket accepted in step 3 (via `IORING_OP_READ`), and then
6. Asynchronously destroy all three sockets (listening, connecting, and accepted) (via `IORING_OP_CLOSE`)

```
const std::string_view sv("Hello world!");
std::vector<std::byte> buffer(sv.size());
std::this_thread::sync_wait(
  run_on_blocking_io_uring(
    [&](io_uring_context& ctx) {
      const socket_object object(
        ctx,
        AF_INET,
        SOCK_STREAM,
        IPPROTO_TCP);
      return std::execution::lifetime(
        [&](file_descriptor& server, file_descriptor& client)
          -> std::execution::task<void>
        {
          sockaddr_in addr;
          std::memset(&addr, 0, sizeof(addr));
          addr.sin_family = AF_INET;
          co_await bind(server, addr);
          co_await listen(server, SOMAXCONN);
          socklen_t out = sizeof(addr);
          if (getsockname(
            server.native_handle(),
            reinterpret_cast<sockaddr*>(&addr),
            &out) == -1)
          {
            throw std::runtime_error("getsockname failed");
          }
          co_await std::execution::when_all(
            [&]() -> std::execution::task<void> {
              co_await connect(client, addr);
```

```
            co_await write(
              client,
              std::span{
                reinterpret_cast<const std::byte*>(sv.data()),
                sv.size()});
          }(),
          std::execution::lifetime(
            [&](file_descriptor& accepted) {
              return read(accepted, buffer);
            },
            accept_object(server)));
      },
      object,
      object);
  },
  32,
  io_uring_params{}));
```

Note that io_uring operations which simply perform I/O (`IORING_OP_CONNECT`, `IORING_OP_WRITE`, et cetera) are presented such that they yield a regular sender (which can be `co_await`ed due to `std::execution::task` [12]) whereas io_uring operations which create a file descriptor (`IORING_OP_SOCKET` and `IORING_OP_ACCEPT`) are wrapped by an async object and reified via `execution::lifetime` to ensure the proper asynchronous destruction occurs on scope exit.

Importantly the above code is materially different from (note important differences in **bold**):

```
const std::string_view sv("Hello world!");
std::vector<std::byte> buffer(sv.size());
std::this_thread::sync_wait(
  run_on_blocking_io_uring(
    [&](io_uring_context& ctx) {
      file_descriptor server(
        ctx,
        co_await socket(
          ctx,
          AF_INET,
          SOCK_STREAM,
          IPPROTO_TCP));
      file_descriptor client(
        ctx,
        co_await socket(
          ctx,
          AF_INET,
```

```
      SOCK_STREAM,
      IPPROTO_TCP));
  sockaddr_in addr;
  std::memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  co_await bind(server, addr);
  co_await listen(server, SOMAXCONN);
  socklen_t out = sizeof(addr);
  if (getsockname(
    server.native_handle(),
    reinterpret_cast<sockaddr*>(&addr),
    &out) == -1)
  {
    throw std::runtime_error("getsockname failed");
  }
  co_await std::execution::when_all(
    [&]() -> std::execution::task<void> {
      co_await connect(client, addr);
      co_await write(
        client,
        std::span{
          reinterpret_cast<const std::byte*>(sv.data()),
          sv.size()});
    }(),
    [&]() -> std::exceution::task<void> {
      file_descriptor accepted(
        ctx,
        co_await accept(server));
      co_await read(accepted, buffer);
    }());
},
32,
io_uring_params{}));
```

Because in this example `file_descriptor` has been reimagined as a regular, synchronous object with a regular, synchronous destructor. Therefore the synchronous close syscall must be used for clean up rather than the asynchronous `IORING_OP_CLOSE`. Note that in both cases the sockets are constructed asynchronously (via `IORING_OP_SOCKET` and `IORING_OP_ACCEPT`), it is only in destruction that the examples materially differ.

Implementation Experience

The author has implemented this design on top of nVidia's stdexec [13]. The implementation is not yet published.

Acknowledgements

The author would like to thank Kirk Shoop for his permission to continue his exploration of this problem space.

The author would like to thank Eric Niebler and Mark Hoemmen for feedback on this paper.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] K. Shoop. async-object - aka async-RAII P2849R0
[3] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[4] I. Petersen et al. async_scope - Creating scopes for non-sequential concurrency P3149R7
[5] A. Williams. Let_async_scope P3296R4
[6] P. Sommerlad et al. Generic Scope Guard and RAII Wrapper for the Standard Library P0052R10
[7] R. Leahy. Rename async_scope_token P3685R0
[8] R. Leahy. When Do You Know connect Doesn't Throw? P3388R3
[9] https://github.com/NVIDIA/stdexec/blob/485160802ee5ca42ca4915e3a2330579efae4ea3/include/exec/sequence.hpp
[10] R. Leahy. Evolving C++ Networking with Senders & Receivers (Part 2). Core C++ 2024
[11] R. Leahy Of Operation States and Their Lifetimes P3373R2
[12] D. Kühl et al. Add a Coroutine Task Type P3552R3
[13] https://github.com/NVIDIA/stdexec