`async_scope` – Creating scopes for non-sequential concurrency

Document #:	P3149R1
Date:	2024-03-13
Project:	Programming Language C++
Audience:	SG1 Parallelism and Concurrency LEWG Library Evolution

Reply-to:	Ian Petersen <ispeters@meta.com> Ján Ondrušek <ondrusek@meta.com> Jessica Wong <jesswong@meta.com> Kirk Shoop <kirk.shoop@gmail.com> Lee Howes <lwh@fb.com> Lucian Radu Teodorescu <lucteo@lucteo.ro>

Contents

1 Changes

1.1 R1

— Add implementation experience
— Incorporate pre-meeting feedback from Eric Niebler

1.2 R0

— First revision

2 Introduction

[P2300R7] lays the groundwork for writing structured concurrent programs in C++ but it leaves three important scenarios under- or unaddressed:

1. progressively structuring an existing, unstructured concurrent program;
2. starting a dynamic number of parallel tasks without "losing track" of them; and
3. opting in to eager execution of sender-shaped work when appropriate.

This paper describes the utilities needed to address the above scenarios within the following constraints:

— *No detached work by default;* as specified in [P2300R7], the `start_detached` and `ensure_started` algorithms invite users to start concurrent work with no built-in way to know when that work has finished.
  — Such so-called "detached work" is undesirable; without a way to know when detached work is done, it is difficult know when it is safe to destroy any resources referred to by the work. Ad hoc solutions to this shutdown problem add unnecessary complexity that can be avoided by ensuring all concurrent work is "attached".

  - [P2300R7]'s introduction of structured concurrency to C++ will make async programming with C++ much easier but experienced C++ programmers typically believe that async C++ is "just hard" and that starting async work *means* starting detached work (even if they are not thinking about the distinction between attached and detached work) so adapting to a post-[P2300R7] world will require unlearning many deprecated patterns. It is thus useful as a teaching aid to remove the unnecessary temptation of falling back on old habits.
- *No dependencies besides [P2300R7];* it will be important for the success of [P2300R7] that existing code bases can migrate from unstructured concurrency to structured concurrency in an incremental way so tools for progressively structuring code should not take on risk in the form of unnecessary dependencies.

The proposed solution comes in five parts:

- `sender auto nest(sender auto&& snd, auto&& scope);`
- `template <class Scope, class Sender> concept async_scope;`
- `void spawn(sender auto&& snd, async_scope auto&& scope, auto&& env);`
- `sender auto spawn_future(sender auto&& snd, async_scope auto&& scope, auto&& env);` and
- `struct counting_scope`.

2.1 Implementation experience

The general concept of an async scope to manage work has been deployed broadly at Meta. Code written with Folly's coroutine library, [`folly::coro`], uses [`folly::coro::AsyncScope`] to safely launch awaitables. Most code written with Unifex, an implementation of an earlier version of the *Sender/Receiver* model proposed in [P2300R7], uses [`unifex::v1::async_scope`], although experience with the v1 design led to the creation of [`unifex::v2::async_scope`], which has a smaller interface and a cleaner definition of responsibility.

As an early adopter of Unifex, [rsys] (Meta's cross-platform voip client library) became the entry point for structured concurrency in mobile code at Meta. We originally built rsys with an unstructured asynchrony model built around posting callbacks to threads in order to optimize for binary size. However, this came at the expense of developer velocity due to the increasing cost of debugging deadlocks and crashes resulting from race conditions.

We decided to adopt Unifex and refactor towards a more structured architecture to address these problems systematically. Converting an unstructured production codebase to a structured one is such a large project that it needs to be done in phases. As we began to convert callbacks to senders/tasks, we quickly realized that we needed a safe place to start structured asynchronous work in an unstructured environment. We addressed this need with `unifex::v1::async_scope` paired with an executor to address a recurring pattern:

Before

```
// Abstraction for thread that has the ability
// to execute units of work.
class Executor {
 public:
  virtual void add(Func function) noexcept = 0;
};

// Example class
class Foo {
  std::shared_ptr<Executor> exec_;

 public:
  void doSomething() {
     auto asyncWork = [&]() {
       // do something
     };
     exec_->add(asyncWork);
  }
};
```

After

```
// Utility class for executing async work on an
// async_scope and on the provided executor
class ExecutorAsyncScopePair {
  unifex::v1::async_scope scope_;
  ExecutorScheduler exec_

 public:
  void add(Func func) {
    scope_.detached_spawn_call_on(
      exec_, func);
  }

  auto cleanup() {
    return scope_.cleanup();
  }
};

// Example class
class Foo {
 std::shared_ptr<ExecutorAsyncScopePair> exec_;

 public:
  ~Foo() {
    sync_wait(exec_->cleanup());
  }

  void doSomething() {
    auto asyncWork = [&]() {
      // do something
    };


   exec_->add(asyncWork);
  }
};
```

This broadly worked but we discovered that the above design coupled with the v1 API allowed for too many redundancies and conflated too many responsibilities (scoping async work, associating work with a stop source, and transferring scoped work to a new scheduler).

We learned that making each component own a distinct responsibility will minimize the confusion and increase the structured concurrency adoption rate. The above example was an intuitive use of async_scope because the concept of a "scoped executor" was familiar to many engineers and is a popular async pattern in other programming languages. However, the above design abstracted away some of the APIs in async_scope that explicitly asked for a scheduler, which would have helped challenge the assumption engineers made about async_scope being an instance of a "scoped executor".

Cancellation was an unfamiliar topic for engineers within the context of asynchronous programming. The `v1::async_scope` provided both `cleanup()` and `complete()` to give engineers the freedom to decide between canceling work or waiting for work to finish. The different nuances on when this should happen and how it happens ended up being an obstacle that engineers didn't want to deal with.

Over time, we also found redundancies in the way `v1::async_scope` and other algorithms were implemented and identified other use cases that could benefit from a different kind of async scope. This motivated us to create `v2::async_scope` which only has one responsibility (scope), and `nest` which helped us improve maintainability and flexibility of Unifex.

The unstructured nature of `cleanup()`/`complete()` in a partially structured codebase introduced deadlocks when engineers nested the `cleanup()`/`complete()` sender in the scope being joined. This risk of deadlock remains with `v2::async_scope::join()` however, we do think this risk can be managed and is worth the tradeoff in exchange for a more coherent architecture that has fewer crashes. For example, we have experienced a significant reduction in these types of deadlocks once engineers understood that `join()` is a destructor-like operation that needs to be run only by the scope's owner. Since there is no language support to manage async lifetimes automatically, this insight was key in preventing these types of deadlocks. Although this breakthrough was a result of strong guidance from experts, we believe that the simpler design of `v2::async_scope` would make this a little easier.

We strongly believe that async_scope was necessary for making structured concurrency possible within rsys, and we believe that the improvements we made with `v2::async_scope` will make the adoption of P2300 more accessible.

3 Motivation

3.1 Motivating example

Let us assume the following code:

```
namespace ex = std::execution;

struct work_context;
struct work_item;
void do_work(work_context&, work_item*);
std::vector<work_item*> get_work_items();

int main() {
    static_thread_pool my_pool{8};
    work_context ctx; // create a global context for the application

    std::vector<work_item*> items = get_work_items();
    for (auto item : items) {
        // Spawn some work dynamically
        ex::sender auto snd = ex::transfer_just(my_pool.get_scheduler(), item) |
                              ex::then([&](work_item* item) { do_work(ctx, item); });
        ex::start_detached(std::move(snd));
    }
    // `ctx` and `my_pool` are destroyed
}
```

In this example we are creating parallel work based on the given input vector. All the work will be spawned on the local `static_thread_pool` object, and will use a shared `work_context` object.

Because the number of work items is dynamic, one is forced to use `start_detached()` from [P2300R7] (or something equivalent) to dynamically spawn work. [P2300R7] doesn't provide any facilities to spawn dynamic work and return a sender (i.e., something like `when_all` but with a dynamic number of input senders).

Using `start_detached()` here follows the *fire-and-forget* style, meaning that we have no control over, or awareness of, the completion of the async work that is being spawned.

At the end of the function, we are destroying the `work_context` and the `static_thread_pool`. But at that

point, we don't know whether all the spawned async work has completed. If any of the async work is incomplete, this might lead to crashes.

[P2300R7] doesn't give us out-of-the-box facilities to use in solving these types of problems.

This paper proposes the `counting_scope` facility that would help us avoid the invalid behavior. With `counting_scope`, one might write safe code this way:

```
namespace ex = std::execution;

struct work_context;
struct work_item;
void do_work(work_context&, work_item*);
std::vector<work_item*> get_work_items();

int main() {
    static_thread_pool my_pool{8};
    work_context ctx;         // create a global context for the application
    ex::counting_scope scope; // create this *after* the resources it protects

    std::vector<work_item*> items = get_work_items();
    for (auto item : items) {
        // Spawn some work dynamically
        ex::sender auto snd = ex::transfer_just(my_pool.get_scheduler(), item) |
                              ex::then([&](work_item* item) { do_work(ctx, item); });

        // start `snd` as before, but associate the spawned work with `scope` so that it can
        // be awaited before destroying the resources referenced by the work (i.e. `my_pool`
        // and `ctx`)
        ex::spawn(std::move(snd), scope); // NEW!
    }

    // wait for all nested work to finish
    this_thread::sync_wait(scope.join()); // NEW!

    // `ctx` and `my_pool` are destroyed *after* they are no longer referenced
}
```

Simplifying the above into something that fits in a Tony Table to highlight the differences gives us:

Before

```
namespace ex = std::execution;

struct context;
ex::sender auto work(const context&);

int main() {
  context ctx;

  ex::sender auto snd = work(ctx);

  // fire and forget
  ex::start_detached(std::move(snd));

  // `ctx` is destroyed, perhaps before
  // `snd` is done
}
```

After

```
namespace ex = std::execution;

struct context;
ex::sender auto work(const context&);

int main() {
  context ctx;
  ex::counting_scope scope;

  ex::sender auto snd = work(ctx);

  // fire, but don't forget
  ex::spawn(std::move(snd), scope);

  // wait for all work nested within scope
  // to finish
  this_thread::sync_wait(scope.join());

  // `ctx` is destroyed once nothing
  // references it
}
```

Please see below for more examples.

3.2 `counting_scope` is step forward towards Structured Concurrency

Structured Programming [Dahl72] transformed the software world by making it easier to reason about the code, and build large software from simpler constructs. We want to achieve the same effect on concurrent programming by ensuring that we *structure* our concurrent code. [P2300R7] makes a big step in that direction, but, by itself, it doesn't fully realize the principles of Structured Programming. More specifically, it doesn't always ensure that we can apply the *single entry, single exit point* principle.

The `start_detached` sender algorithm fails this principle by behaving like a `GOTO` instruction. By calling `start_detached` we essentially continue in two places: in the same function, and on different thread that executes the given work. Moreover, the lifetime of the work started by `start_detached` cannot be bound to the local context. This will prevent local reasoning, which will make the program harder to understand.

To properly structure our concurrency, we need an abstraction that ensures that all async work that is spawned has a defined, observable, and controllable lifetime. This is the goal of `counting_scope`.

4 Examples of use

4.1 Spawning work from within a task

Use a `counting_scope` in combination with a `system_context` from [P2079R2] to spawn work from within a task and join it later:

```
namespace ex = std::execution;

int main() {
    ex::system_context ctx;
    int result = 0;
```

```
    ex::counting_scope scope;

    ex::scheduler auto sch = ctx.scheduler();

    ex::sender auto val = ex::on(sch, ex::just() | ex::then([&result, sch, &scope]() {
        int val = 13;

        auto print_sender = ex::just() | ex::then([val] {
            std::cout << "Hello world! Have an int with value: " << val << "\n";
        });

        // spawn the print sender on sch to make sure it completes before shutdown
        ex::spawn(scope, ex::on(sch, std::move(print_sender)));

        return val;
    })) | ex::then([&result](auto val) { result = val });

    ex::spawn(scope, std::move(val));

    this_thread::sync_wait(scope.join());

    std::cout << "Result: " << result << "\n";
}

// The counting scope ensured that all work is safely joined, so result contains 13
```

4.2 Starting work nested within a framework

In this example we use the `counting_scope` within a class to start work when the object receives a message and to wait for that work to complete before closing.

```
namespace ex = std::execution;

struct my_window {
    class close_message {};

    ex::sender auto some_work(int message);

    ex::sender auto some_work(close_message message);

    void onMessage(int i) {
        ++count;
        ex::spawn(scope, ex::on(sch, some_work(i)));
    }

    void onClickClose() {
        ++count;
        ex::spawn(scope, ex::on(sch, some_work(close_message{})));
    }

    ex::system_scheduler sch;
    ex::counting_scope& scope;
    int count{0};
};
```

```
int main() {
    // keep track of all spawned work
    ex::counting_scope scope;
    ex::system_context ctx;
    my_window window{ctx.get_scheduler(), scope};
    // wait for all work nested within scope to finish
    this_thread::sync_wait(scope.join());
    // all resources are now safe to destroy
    return window.count;
}
```

4.3 Starting parallel work

In this example we use the `counting_scope` within lexical scope to construct an algorithm that performs parallel work. This uses the `let_value_with` [letvwthunifex] algorithm implemented in [libunifex] which simplifies in-place construction of a non-moveable object in the `let_value_with` algorithm's *`operation-state`* object. Here `foo` launches 100 tasks that concurrently run on some scheduler provided to `foo`, through its connected receiver, and then the tasks are asynchronously joined. This structure emulates how we might build a parallel algorithm where each `some_work` might be operating on a fragment of data.

```
namespace ex = std::execution;

ex::sender auto some_work(int work_index);

ex::sender auto foo(ex::scheduler auto sch) {
    return unifex::let_value_with([]() noexcept { return ex::counting_scope{}; },
                    [sch](ex::counting_scope& scope) {
                        return ex::schedule(sch) | ex::then([] {
                            std::cout << "Before tasks launch\n";
                        }) | ex::then([sch, &scope] {
                            // Create parallel work
                            for (int i = 0; i < 100; ++i)
                                ex::spawn(scope, ex::on(sch, some_work(i)));
                        }) | ex::let_value([&scope]() noexcept {
                            // Join the work with the help of the scope
                            return scope.join();
                        });
                    }) |
            ex::then([] { std::cout << "After tasks complete\n"; });
}
```

4.4 Listener loop in an HTTP server

This example shows how one can write the listener loop in an HTTP server, with the help of coroutines. The HTTP server will continuously accept new connection and start work to handle the requests coming on the new connections. While the listening activity is bound in the scope of the loop, the lifetime of handling requests may exceed the scope of the loop. We use `counting_scope` to limit the lifetime of the request handling without blocking the acceptance of new requests.

```
namespace ex = std::execution;

task<size_t> listener(int port, io_context& ctx, static_thread_pool& pool) {
    size_t count{0};
    listening_socket listen_sock{port};
```

```
    ex::counting_scope work_scope;

    while (!ctx.is_stopped()) {
        // Accept a new connection
        connection conn = co_await async_accept(ctx, listen_sock);
        count++;

        // Create work to handle the connection in the scope of `work_scope`
        conn_data data{std::move(conn), ctx, pool};
        ex::sender auto snd = ex::just(std::move(data)) |
                              ex::let_value([](auto& data) {
                                return handle_connection(data);
                              });
        ex::spawn(scope, std::move(snd));
    }

    // Continue only after all requests are handled
    co_await work_scope.join();

    // At this point, all the request handling is complete
    co_return count;
}
```

[libunifex] has a very similar example HTTP server at [io_uring HTTP server] that compiles and runs on Linux-based machines with `io_uring` support.

5 Async Scope, usage guide

An async scope is a type that implements a "bookkeeping policy" for senders that have been `nest()`ed within the scope. Depending on the policy, different guarantees can be provided in terms of the lifetimes of the scope and any nested senders. The `counting_scope` described in this paper defines a policy that has proven useful while progressively adding structure to existing, unstructured code at Meta, but other useful policies are possible. By defining `spawn()` and `spawn_future()` in terms of the more fundamental `nest()`, and leaving the definition of `nest()` to the scope, this paper's design leaves the set of policies open to extension by user code or future standards.

An async scope's implementation of `nest()`:

- must allow an arbitrary sender to be nested within the scope without eagerly starting the sender;
- must not return a sender that adds new value or error completions to the completions of the sender being nested;
- may fail to nest a new sender by returning an "unnested" sender that completes with `set_stopped` when run without running the sender that failed to nest;
- may fail to nest a new sender by eagerly throwing an exception during the call to `nest()`; and
- is expected to be "cheap" like other sender adaptor objects.

More on these items can be found below in the sections below.

5.1 Definitions

```
namespace { // exposition-only

template <class Env>
struct spawn-env; // exposition-only
```

```
template <class Env>
struct spawn-receiver { // exposition-only
    void set_value() noexcept;
    void set_stopped() noexcept;

    spawn-env<Env> get_env() const noexcept;
};

template <class Env>
struct future-env; // exposition-only

template <valid-completion-signatures Sigs>
struct future-sender; // exposition-only

template <sender Sender, class Env>
using future-sender-t = // exposition-only
    future-sender<completion_signatures_of_t<Sender, future-env<Env>>>;

}

template <sender Sender>
auto nest(Sender&& snd, auto&& scope)
    noexcept(noexcept(scope.nest(std::forward<Sender>(snd)))
    -> decltype(scope.nest(std::forward<Sender>(snd)));

template <class Scope, class Sender>
concept async_scope =
    sender<Sender> &&
    requires(Scope&& scope, Sender&& snd) {
      { nest(std::forward<Sender>(snd), std::forward<Scope>(scope)) } -> sender;
    };

template <sender Sender, async_scope<Sender> Scope, class Env = empty_env>
  requires sender_to<Sender, spawn-receiver<Env>>
void spawn(Sender&& snd, Scope&& scope, Env env = {});

template <sender Sender, async_scope<Sender> Scope, class Env = empty_env>
future-sender-t<Sender, Env> spawn_future(Sender&& snd, Scope&& scope, Env env = {});

struct counting_scope {
    counting_scope() noexcept;
    ~counting_scope();

    // counting_scope is immovable and uncopyable
    counting_scope(const counting_scope&) = delete;
    counting_scope(counting_scope&&) = delete;
    counting_scope& operator=(const counting_scope&) = delete;
    counting_scope& operator=(counting_scope&&) = delete;

    template <sender S>
    struct nest-sender; // exposition-only

    template <sender S>
    [[nodiscard]] nest-sender<std::remove_cvref_t<S>> nest(S&& s) & noexcept(
```

```
            std::is_nothrow_constructible_v<std::remove_cvref_t<S>, S>);

    struct join-sender; // exposition-only

    [[nodiscard]] join-sender join() noexcept;

    // observers in the spirit of std::weak_ptr<T>::expired() and
    // std::shared_ptr<T>::use_count(); the values must be correct
    // when computed but may be stale by the time they can be observed

    [[nodiscard]] bool joined() const noexcept;

    [[nodiscard]] bool join_started() const noexcept;

    [[nodiscard]] size_t use_count() const noexcept;
};
```

5.2 execution::nest

```
template <sender Sender>
auto nest(Sender&& snd, auto&& scope) noexcept(noexcept(scope.nest(std::forward<Sender>(snd)))
    -> decltype(scope.nest(std::forward<Sender>(snd)));
```

Attempts to associate the given sender with the given scope in a scope-defined way. When successful, the return value is an "associated sender" with the same behaviour and possible completions as the input sender, plus the additional, scope-specific behaviours that are necessary to implement the scope's bookkeeping policy. When the attempt fails, `nest()` may either eagerly throw an exception, or return a "unassociated sender" that, when started, unconditionally completes with `set_stopped()`.

A call to `nest()` does not start the given sender and is not expected to incur allocations.

When `nest()` returns an associated sender:

— connecting and starting the associated sender connects and starts the given sender;
— the associated sender is multi-shot if the input sender is multi-shot and single-shot otherwise; and
— the associated sender has exactly the same completions as the input sender.

When `nest()` returns an unassociated sender:

— the input sender is discarded and will never be connected or started;
— the unassociated sender is multi-shot; and
— the unassociated sender must only complete with `set_stopped()`.

5.3 execution::async_scope

```
template <class Scope, class Sender>
  concept async_scope =
    // only senders can be nested within scopes
    sender<Sender> &&
    // a scope is anything that can nest senders within itself
    requires(Scope&& scope, Sender&& snd) {
      { nest(std::forward<Sender>(snd), std::forward<Scope>(scope)) } -> sender;
    };
```

As described above, an async scope is a type that implements a bookkeeping policy for senders. The `nest()` algorithm is the means by which senders are submitted as the subjects of such a policy so any type that permits

senders to be `nest()`ed with it satisfies the `async_scope` concept.

5.4 `execution::spawn()`

```
namespace { // exposition-only

template <class Env>
struct spawn-env; // exposition-only

template <class Env>
struct spawn-receiver { // exposition-only
    void set_value() noexcept;
    void set_stopped() noexcept;

    spawn-env<Env> get_env() const noexcept;
};

}

template <sender Sender, async_scope<Sender> Scope, class Env = empty_env>
  requires sender_to<Sender, spawn-receiver<Env>>
void spawn(Sender&& snd, Scope&& scope, Env env = {});
```

Invokes `nest(std::forward<Sender>(snd), std::forward<Scope>(scope))` to associate the given sender with the given scope and then eagerly starts the resulting sender.

Starting the nested sender involves a dynamic allocation of the sender's *`operation-state`*. The following algorithm determines which *Allocator* to use for this allocation:

— If `get_allocator(env)` is valid and returns an *Allocator* then choose that *Allocator*.
— Otherwise, if `get_allocator(get_env(snd))` is valid and returns an *Allocator* then choose that *Allocator*.
— Otherwise, choose `std::allocator<>`.

The *`operation-state`* is constructed by connecting the nested sender to a *`spawn-receiver`*. The *`operation-state`* is destroyed and deallocated after the spawned sender completes.

A *`spawn-receiver`*, `sr`, responds to `get_env(sr)` with an instance of a *`spawn-env`*`<Env>`, `senv`. The result of `get_allocator(senv)` is a copy of the *Allocator* used to allocate the *`operation-state`*. For all other queries, `Q`, the result of `Q(senv)` is `Q(env)`.

This is similar to `start_detached()` from [P2300R7], but the scope may observe and participate in the lifecycle of the work described by the sender. The `counting_scope` described in this paper uses this opportunity to keep a count of nested senders that haven't finished, and to prevent new work from being started once the `counting_scope`'s *`join-sender`* has been started.

The given sender must complete with `void` or `stopped` and may not complete with an error; the user must explicitly handle the errors that might appear as part of the *`sender-expression`* passed to `spawn()`.

User expectations will be that `spawn()` is asynchronous and so, to uphold the principle of least surprise, `spawn()` should only be given non-blocking senders. Using `spawn()` with a sender generated by `on(sched, `*`blocking-sender`*`)` is a very useful pattern in this context.

NOTE: A query for non-blocking start will allow `spawn()` to be constrained to require non-blocking start.

Usage example:

```
...
for (int i = 0; i < 100; i++)
    spawn(on(sched, some_work(i)), scope);
```

5.5 execution::spawn_future()

```
namespace { // exposition-only

template <class Env>
struct future-env; // exposition-only

template <valid-completion-signatures Sigs>
struct future-sender; // exposition-only

template <sender Sender, class Env>
using future-sender-t = // exposition-only
    future-sender<completion_signatures_of_t<Sender, future-env<Env>>>;

}

template <sender Sender, async_scope<Sender> Scope, class Env = empty_env>
future-sender-t<Sender, Env> spawn_future(Sender&& snd, Scope&& scope, Env env = {});
```

Invokes `nest(std::forward<Sender>(snd), std::forward<Scope>(scope))` to associate the given sender with the given scope, eagerly starts the resulting sender, and returns a *future-sender* that provides access to the result of the given sender.

Similar to `spawn()`, starting the nested sender involves a dynamic allocation of some state. `spawn_future()` chooses an *Allocator* for this allocation in the same way `spawn()` does: use the result of `get_allocator(env)` if that is a valid expression, otherwise use the result of `get_allocator(get_env(snd))` if that is a valid expression, otherwise use a `std::allocator<>`.

Unlike `spawn()`, the dynamically allocated state contains more than just an *operation-state* for the nested sender; the state must also contain storage for the result of the nested sender, however it eventually completes, and synchronization facilities for resolving the race between the nested sender's production of its result and the returned sender's consumption or abandonment of that result.

Also unlike `spawn()`, `spawn_future()` returns a *future-sender* rather than `void`. The returned sender, `fs`, is a handle to the spawned work that can be used to consume or abandon the result of that work. When `fs` is connected and started, it waits for the spawned sender to complete and then completes itself with the spawned sender's result. If `fs` is destroyed before being connected, or if `fs` *is* connected but then the resulting *operation-state* is destroyed before being started, then a stop request is sent to the spawned sender in an effort to short-circuit the computation of a result that will not be observed. If `fs` receives a stop request from its receiver before the spawned sender completes, the stop request is forwarded to the spawned sender and then `fs` completes; if the spawned sender happens to complete between `fs` forwarding the stop request and completing itself then `fs` may complete with the result of the spawned sender as if the stop request was never received but, otherwise, `fs` completes with `stopped` and the result of the spawned sender is ignored. The completion signatures of `fs` include `set_stopped()` and all the completion signatures of the spawned sender.

The receiver, `fr`, that is connected to the nested sender responds to `get_env(fr)` with an instance of *future-env*`<Env>`, `fenv`. The result of `get_allocator(fenv)` is a copy of the *Allocator* used to allocate the dynamically allocated state. The result of `get_stop_token(fenv)` is a stop token that will be "triggered" (i.e. signal that stop is requested) by the returned *future-sender* when it is dropped or receives a stop request itself. For all other queries, `Q`, the result of `Q(fenv)` is `Q(env)`.

This is similar to `ensure_started()` from [P2300R7], but the scope may observe and participate in the lifecycle of the work described by the sender. The `counting_scope` described in this paper uses this opportunity to keep a count of nested senders that haven't finished, and to prevent new work from being started once the `counting_scope`'s *join-sender* has been started.

Unlike `spawn()`, the sender given to `spawn_future()` is not constrained on a given shape. It may send different

types of values, and it can complete with errors.

NOTE: there is a race between the completion of the given sender and the start of the returned sender. The spawned sender and the returned *`future-sender`* use the synchronization facilities in the dynamically allocated state to resolve this race.

Cancelling the returned sender requests cancellation of the given sender, `snd`, but does not affect any other senders.

Usage example:

```
...
sender auto snd = spawn_future(on(sched, key_work()), scope) | then(continue_fun);
for (int i = 0; i < 10; i++)
    spawn(on(sched, other_work(i)), scope);
return when_all(scope.join(), std::move(snd));
```

5.6 `execution::counting_scope`

```
struct counting_scope {
    counting_scope() noexcept;
    ~counting_scope();

    counting_scope(const counting_scope&) = delete;
    counting_scope(counting_scope&&) = delete;
    counting_scope& operator=(const counting_scope&) = delete;
    counting_scope& operator=(counting_scope&&) = delete;

    template <sender S>
    struct nest-sender; // exposition-only

    template <sender S>
    [[nodiscard]] nest-sender<std::remove_cvref_t<S>> nest(S&& s) & noexcept(
            std::is_nothrow_constructible_v<std::remove_cvref_t<S>, S>);

    struct join-sender; // exposition-only

    [[nodiscard]] join-sender join() noexcept;

    // observers in the spirit of std::weak_ptr<T>::expired() and
    // std::shared_ptr<T>::use_count(); the values must be correct
    // when computed but may be stale by the time they can be observed

    [[nodiscard]] bool joined() const noexcept;

    [[nodiscard]] bool join_started() const noexcept;

    [[nodiscard]] size_t use_count() const noexcept;
};
```

A `counting_scope` goes through three states during its lifetime:

1. open
2. closed/joining
3. joined

Instances start in the open state after being constructed. Connecting and starting a *`join-sender`* returned from

`join()` transitions the scope to the closed/joining state. Merely calling `join()` or connecting the *`join-sender`* does not change the scope's state—the *`operation-state`* must be started to close the scope. The scope transitions from the closed/joining state to the joined state when the *`join-sender`* completes. A scope must be in the joined state when its destructor starts; otherwise, the destructor invokes `std::terminate()`.

While a scope is open, calls to `nest(snd, scope)` will succeed (unless an exception is thrown by `snd`'s copy- or move-constructor while constructing the *`nest-sender`*). Each time a call to `nest(snd, scope)` succeeds, two things happen:

1. the scope's count of outstanding senders is incremented before `nest()` returns, and
2. the given sender, `snd`, is wrapped in a *`nest-sender`* and returned.

When a call to `nest()` succeeds, the returned *`nest-sender`* is an associated sender that acts like an RAII handle: the scope's internal count is incremented when the sender is created and decremented when the sender is "done with the scope", which happens when the sender is destroyed, its *`operation-state`* is destroyed, or its *`operation-state`* is completed. Moving a *`nest-sender`* transfers responsibility for decrementing the count from the old instance to the new one. Copying a *`nest-sender`* is permitted if the sender it's wrapping is copyable, but the copy may "fail" since copying requires incrementing the scope's count, which is only allowed when the scope is open; if copying fails, the new sender is an unassociated sender that behaves as if it were the result of a failed call to `nest()`.

While a scope is closed or joined, calls to `nest(snd, scope)` will always fail by discarding the given sender and returning an unassociated *`nest-sender`*. Failed calls to `nest()` do not change the scope's count. Unassociated *`nest-senders`* do not have a reference to the scope they came from and always complete with `stopped` when connected and started. Copying or moving an unassociated sender produces another unassociated sender.

The state transitions of a `counting_scope` mean that it can be used to protect asynchronous work from use-after-free errors. Given a resource, `res`, and a `counting_scope`, `scope`, obeying the following policy is enough to ensure that there are no attempts to use `res` after its lifetime ends:

— all senders that refer to `res` are nested within `scope`; and
— `scope` is destroyed (and therefore joined) before `res` is destroyed.

Under the standard assumption that the arguments to `nest()` are and remain valid while evaluating `nest()`, it is always safe to invoke any supported operation on the returned *`nest-sender`*. Furthermore, if all senders returned from `nest()` are eventually started or discarded then the `join()` operation always eventually finishes because the number of outstanding senders nested within the corresponding scope is monotonically decreasing. Conversely, the `join()` operation will never terminate if there are any associated *`nest-senders`* that never become "done with the scope" (i.e. that remain either unconnected or unstarted until after the `join()` is expected to complete). For example:

```
void deadlock() {
    namespace ex = std::execution;

    ex::counting_scope scope;

    ex::sender auto s = ex::nest(ex::just(), scope);

    // never completes because s's continued existence keeps the scope open
    std::this_thread::sync_wait(scope.join());
}
```

The risk of deadlock is explicitly preferred in this design over the risk of use-after-free errors because `counting_scope` is an async scope that is biased towards being used to progressively add structure to generally-unstructured code. We've found that the central problem in progressively structuring unstructured code is determining appropriate bounds for each asynchronous task when those bounds are not clear; it is easier to figure out where to synchronously `join()` a scope than it is to ensure that all `spawn()`ed work is properly scoped within any particular object's lifetime. So, although it is generally easier to diagnose use-after-free errors

than it is to diagnose deadlocks, we've found that it's easier to *avoid* deadlocks with this design than it is to avoid use-after-free errors with other designs.

A `counting_scope` is uncopyable and immovable so its copy and move operators are explicitly deleted. `counting_scope` could be made movable but it would cost an allocation so this is not proposed.

5.6.1 `counting_scope::counting_scope()`

```
counting_scope::counting_scope() noexcept;
```

Initializes a `counting_scope` in the open state with no outstanding senders.

5.6.2 `counting_scope::~counting_scope()`

```
counting_scope::~counting_scope();
```

Checks that the `counting_scope` is in the joined state and invokes `std::terminate()` if not.

5.6.3 `counting_scope::nest()`

```
template <sender S>
struct nest-sender; // exposition-only

template <sender S>
[[nodiscard]] nest-sender<std::remove_cvref_t<S>> nest(S&& s) & noexcept(
        std::is_nothrow_constructible_v<std::remove_cvref_t<S>, S>);
```

Atomically increments the scope's count of outstanding senders if and only if the scope is in the open state and then returns a *`nest-sender`*.

If the atomic increment succeeded then the return value will be an associated *`nest-sender`* that contains a reference to `this` (the associated scope) and a copy of the input sender, `s`, that is copy- or move-constructed from `s`. If this copy or move throws then, before the exception is allowed to escape to the caller, the atomic increment needs to be undone with a decrement to provide the strong exception guarantee.

If the atomic increment failed then the return value will be an unassociated *`nest-sender`* and no exceptions are possible. In this case, the return value does not store a reference to `this` and the given sender, `s`, is discarded.

An associated *`nest-sender`* is a kind of RAII handle to the scope; it is responsible for decrementing the scope's count of outstanding senders in its destructor unless that responsibility is first given to some other object. Move-construction and move-assignment transfer the decrement responsibility to the destination instance. Connecting an instance to a receiver transfers the decrement responsibility to the resulting *`operation-state`*, which must meet the responsibility when the operation completes or is destroyed, whichever comes first (note: if the *`operation-state`* is started, then the decrement should happen *after* invoking a completion method on its receiver to ensure that any reference produced by the nested sender is not dangling at the time of invocation). Whenever the balancing decrement happens (including if it happens as a side effect of allowing an exception to escape from `nest()`), it's possible that the scope has transitioned to the closed/joining state since the *`nest-sender`* was constructed, which means that there is a *`join-sender`* waiting to complete so, if the decrement brings the count of outstanding senders to zero then the waiting *`join-sender`* needs to be notified that the scope is now joined and the sender can complete.

A call to `nest()` does not start the given sender. A call to `nest()` is not expected to incur allocations other than whatever might be required to move or copy `s`.

Similar to `spawn_future()`, `nest()` doesn't constrain the input sender to any specific shape. Any type of sender is accepted.

As `nest()` does not immediately start the given work, it is ok to pass in blocking senders.

`nest()` is lvalue-ref qualified as it would be inappropriate to nest senders in a temporary—the temporary's destructor would unconditionally invoke `std::terminate()` as there would be no way to move it into the joined state.

Usage example:

```
...
sender auto snd = s.nest(key_work());
for (int i = 0; i < 10; i++)
    spawn(on(sched, other_work(i)), scope);
return on(sched, std::move(snd));
```

5.6.4 `counting_scope::join()`

```
struct join-sender; // exposition-only

[[nodiscard]] join-sender join() noexcept;
```

Returns a *`join-sender`*. When the *`join-sender`* is connected to a receiver, `r`, it produces an *`operation-state`*, `o`. When `o` is started, the scope moves from the open state to the closed/joining state. `o` completes with `set_value()` when the scope moves from the closed/joining state to the closed state, which happens when the scope's count of outstanding senders drops to zero. `o` may complete synchronously if it happens to observe that the count of outstanding senders is already zero when started; otherwise, `o` completes on the execution context it was started on by asking its receiver, `r`, for a scheduler, `sch`, with `get_scheduler(get_env(r))` and then starting the sender returned from `schedule(sch)`. This requirement to complete on the receiver's scheduler restricts which receivers a *`join-sender`* may be connected to in exchange for determinism; the alternative would have the *`join-sender`* completing on the execution context of whichever nested operation happens to be the last one to complete.

5.6.5 `counting_scope::joined()`

```
[[nodiscard]] bool joined() const noexcept;
```

Returns `true` if the scope is in the joined state (i.e. a *`join-sender`* returned from `join()` has been connected and started, and the count of outstanding senders has dropped to zero).

`joined()` returning `true` implies that `join_started()` will also return `true` and `use_count()` will return `0`.

`joined()` must not introduce data races but need not synchronize with anything.

Note: if `joined()` returns `true` then it will never again return `false` however, it's possible for a return of `false` to be stale by the time it is observed since another thread of execution may be racing to complete a waiting *`join-sender`*.

5.6.6 `counting_scope::join_started()`

```
[[nodiscard]] bool join_started() const noexcept;
```

Returns `true` if the scope is in the closed/joining state or the joined state (i.e. returns `true` if a *`join-sender`* has been connected and started) and `false` otherwise.

`join_started()` must not introduce data races but need not synchronize with anything.

Note: if `join_started()` returns `true` then it will never again return `false` however, it's possible for a return of `false` to be stale by the time it is observed since another thread of execution may be racing to start a *`join-sender`*.

5.6.7 `counting_scope::use_count()`

```
[[nodiscard]] size_t use_count() const noexcept;
```

Returns the number of senders that have been associated with this scope that have not yet completed.

`use_count()` must not introduce data races but need not synchronize with anything.

Note: it is likely that the return value is stale by the time it's observed since another thread of execution may be racing to nest a new sender or complete an old one.

6 Design considerations

6.1 Shape of the given sender

6.1.1 Constraints on `set_value()`

It makes sense for `spawn_future()` and `nest()` to accept senders with any type of completion signatures. The caller gets back a sender that can be chained with other senders, and it doesn't make sense to restrict the shape of this sender.

The same reasoning doesn't necessarily follow for `spawn()` as it returns `void` and the result of the spawned sender is dropped. There are two main alternatives:

— do not constrain the shape of the input sender (i.e., dropping the results of the computation)
— constrain the shape of the input sender

The current proposal goes with the second alternative. The main reason is to make it more difficult and explicit to silently drop result. The caller can always transform the input sender before passing it to `spawn()` to drop the values manually.

> **Chosen:** `spawn()` accepts only senders that advertise `set_value()` (without any parameters) in the completion signatures.

6.1.2 Handling errors in `spawn()`

The current proposal does not accept senders that can complete with error given to `spawn()`. This will prevent accidental error scenarios that will terminate the application. The user must deal with all possible errors before passing the sender to `counting_scope`. i.e., error handling must be explicit.

Another alternative considered was to call `std::terminate()` when the sender completes with error.

Another alternative is to silently drop the errors when receiving them. This is considered bad practice, as it will often lead to first spotting bugs in production.

> **Chosen:** `spawn()` accepts only senders that do not call `set_error()`. Explicit error handling is preferred over stopping the application, and over silently ignoring the error.

6.1.3 Handling stop signals in `spawn()`

Similar to the error case, we have the alternative of allowing or forbidding `set_stopped()` as a completion signal. Because the goal of `counting_scope` is to track the lifetime of the work started through it, it shouldn't matter whether that the work completed with success or by being stopped. As it is assumed that sending the stop signal is the result of an explicit choice, it makes sense to allow senders that can terminate with `set_stopped()`.

The alternative would require transforming the sender before passing it to spawn, something like `s.spawn(std::move(snd) | let_stopped(just))`. This is considered boilerplate and not helpful, as the stopped scenarios should be implicit, and not require handling.

> **Chosen:** `spawn()` accepts senders that complete with `set_stopped()`.

6.1.4 No shape restrictions for the senders passed to `spawn_future()` and `nest()`

Similarly to `spawn()`, we can constrain `spawn_future()` and `nest()` to accept only a limited set of senders. But, because we can attach continuations for these senders, we would be limiting the functionality that can be expressed. For example, the continuation can handle different types of values and errors.

Chosen: `spawn_future()` and `nest()` accept senders with any completion signatures.

6.2 P2300's `start_detached()`

The `spawn()` method in this paper can be used as a replacement for `start_detached` proposed in [P2300R7]. Essentially it does the same thing, but it also provides the given scope the opportunity to apply its bookkeeping policy to the given sender, which, in the case of `counting_scope`, ensures the program can wait for spawned work to complete before destroying any resources references by that work.

6.3 P2300's `ensure_started()`

The `spawn_future()` method in this paper can be used as a replacement for `ensure_started` proposed in [P2300R7]. Essentially it does the same thing, but it also provides the given scope the opportunity to apply its bookkeeping policy to the given sender, which, in the case of `counting_scope`, ensures the program can wait for spawned work to complete before destroying any resources references by that work.

6.4 Supporting the pipe operator

This paper doesn't support the pipe operator to be used in conjunction with `spawn()` and `spawn_future()`. One might think that it is useful to write code like the following:

```
std::move(snd1) | spawn(s); // returns void
sender auto snd3 = std::move(snd2) | spawn_future(s) | then(...);
```

In [P2300R7] sender consumers do not have support for the pipe operator. As `spawn()` works similarly to `start_detached()` from [P2300R7], which is a sender consumer, if we follow the same rationale, it makes sense not to support the pipe operator for `spawn()`.

On the other hand, `spawn_future()` is not a sender consumer, thus we might have considered adding pipe operator to it.

On the third hand, Unifex supports the pipe operator for both of its equivalent algorithms (`unifex::spawn_detached()` and `unifex::spawn_future()`) and Unifex users have not been confused by this choice.

To keep consistency with `spawn()` this paper doesn't support pipe operator for `spawn_future()`.

7 Naming

As is often true, naming is a difficult task.

7.1 `nest()`

This provides a way to build a sender that is associated with a "scope", which is a type that implements and enforces some bookkeeping policy regarding the senders nested within it. `nest()` does not allocate state, call connect, or call start. `nest()` is the basis operation for async scopes. `spawn()` and `spawn_future()` use `nest()` to associate a given sender with a given scope, and then they allocate, connect, and start the given sender.

It would be good for the name to indicate that it is a simple operation (insert, add, embed, extend might communicate allocation, which `nest()` does not do).

alternatives: `wrap()`, `attach()`

7.2 async_scope

This is a concept that is satisfied by types that support nesting senders within themselves. It is primarily useful for constraining the arguments to `spawn()` and `spawn_future()` to give useful error messages for invalid invocations.

Since concepts don't support existential quantifiers and thus can't express "type `T` is an `async_scope` if there exists a sender, `s`, for which `t.nest(s)` is valid", the `async_scope` concept must be parameterized on both the type of the scope and the type of some particular sender and thus describes whether *this* scope type is an `async_scope` in combination with *this* sender type. Given this limitation, perhaps the name should convey something about the fact that it is checking the relationship between two types rather than checking something about the scope's type alone. Nothing satisfying comes to mind.

alternatives: don't name it and leave it as *exposition-only*

7.3 spawn()

This provides a way to start a sender that produces `void` and to associate the resulting async work with an async scope that can implement a bookkeeping policy that may help ensure the async work is complete before destroying any resources it is using. This allocates, connects, and starts the given sender.

It would be good for the name to indicate that it is an expensive operation.

alternatives: `connect_and_start()`, `spawn_detached()`, `fire_and_remember()`

7.4 spawn_future()

This provides a way to start work and later ask for the result. This will allocate, connect, start, and resolve the race (using synchronization primitives) between the completion of the given sender and the start of the returned sender. Since the type of the receiver supplied to the result sender is not known when the given sender starts, the receiver will be type-erased when it is connected.

It would be good for the name to be ugly, to indicate that it is a more expensive operation than `spawn()`.

alternatives: `spawn_with_result()`

7.5 counting_scope

A `counting_scope` represents the root of a set of nested lifetimes.

One mental model for this is a semaphore. It tracks a count of lifetimes and fires an event when the count reaches 0.

Another mental model for this is block syntax. `{}` represents the root of a set of lifetimes of locals and temporaries and nested blocks.

Another mental model for this is a container. This is the least accurate model. This container is a value that does not contain values. This container contains a set of active senders (an active sender is not a value, it is an operation).

alternatives: `async_scope`

7.5.1 counting_scope::join()

This method returns a sender that, when started, prevents new senders from being nested within the scope and then waits for the scope's count of outstanding senders to drop to zero before completing. It is somewhat analogous to `std::thread::join()` but does not block.

`join()` must be invoked, and the returned sender must be connected, started, and completed, before the scope may be destroyed so it may be useful to convey some of this importance in the name, although `std::thread` has similar requirements for its `join()`.

`join()` is the biggest wart in this design; the need to manually manage the end of a scope's lifetime stands out as less-than-ideal in C++, and there is some real risk that users will write deadlocks with `join()` so perhaps `join()` should have a name that conveys danger.

alternatives: `complete()`, `close()`

7.5.2 `counting_scope::joined()`

This method starts returning `true` once the *`join-sender`* completes and it means "`join()` was called and its work has finished". The name should be the past participle of whichever verb is chosen for `join()` (e.g. `completed()` or `closed()`).

The result of `joined()` may be stale before it can be observed, like `std::weak_ptr<>::expired()`, so users may find the name more obvious if it communicated this staleness.

alternatives: `completed()`, `closed()`

7.5.3 `counting_scope::join_started()`

This method starts returning `true` once the *`join-sender`* has been started and it means that the scope will no longer permit new senders to be associated with it via calls to `nest()`. The name was chosen for its directness: `join_started()` is true when the `join()` operation has started.

Like `join()`, the result may be stale before it can be observed.

alternatives: `joining()`, `closing()`, `completing()`

7.5.4 `counting_scope::use_count()`

This method returns the scope's count of outstanding senders (i.e. the number of senders that have been associated with the scope via calls to `nest()` and that haven't been discarded or completed, yet).

Like `join()` and `join_started()`, the result may be stale before it can be observed. Unlike those two methods, there isn't a simple rule like "once it hits zero it stays there" since new work may always be added to the scope so long as the *`join-sender`* hasn't been started.

The name was chosen by analogy with `std::shared_ptr<>::use_count()`; both represent reference counts that may be immediately stale and for which the 1 -> 0 transition is significant.

alternatives: `outstanding_senders()`

8 Acknowledgements

Thanks to Andrew Royes for unwavering support for the development and deployment of Unifex at Meta and for recognizing the importance of contributing this paper to the C++ Standard.

Thanks to Eric Niebler for the encouragement and support it took to get this paper published.

9 References

[Dahl72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press Ltd., 1972.

[`folly::coro`] folly::coro.
https://github.com/facebook/folly/tree/main/folly/experimental/coro

[`folly::coro::AsyncScope`] folly::coro::AsyncScope.
https://github.com/facebook/folly/blob/main/folly/experimental/coro/AsyncScope.h

[io_uring HTTP server] io_uring HTTP server.
https://github.com/facebookexperimental/libunifex/blob/main/examples/linux/http_server_io_uring_test.cpp

[letvwthunifex] let_value_with.
https://github.com/facebookexperimental/libunifex/blob/main/doc/api_reference.md#let_value_withinvocable-state_factory-invocable-func—sender

[libunifex] libunifex.
https://github.com/facebookexperimental/libunifex/

[P2079R2] Lee Howes, Ruslan Arutyunyan, Michael Voss. 2022-01-15. System execution context.
https://wg21.link/p2079r2

[P2300R7] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Bryce Adelstein Lelbach. 2023-04-21. ‘std::execution‘.
https://wg21.link/p2300r7

[rsys] A smaller, faster video calling library for our apps.
https://engineering.fb.com/2020/12/21/video-engineering/rsys/

[`unifex::v1::async_scope`] unifex::v1::async_scope.
https://github.com/facebookexperimental/libunifex/blob/main/include/unifex/v1/async_scope.hpp

[`unifex::v2::async_scope`] unifex::v2::async_scope.
https://github.com/facebookexperimental/libunifex/blob/main/include/unifex/v2/async_scope.hpp