async_scope – Creating scopes for non-sequential concurrency

Draft Proposal

Document #:	D2519R0
Date:	2023-04-09
Project:	Programming Language C++
Audience:	SG1 Parallelism and Concurrency LEWG Library Evolution

Reply-to:	Kirk Shoop <kirk.shoop@gmail.com> Lee Howes <lwh@fb.com> Lucian Radu Teodorescu <lucteo@lucteo.ro>

Contents

1 Changes

1.1 R0

— first revision

2 Introduction

A major precept of [P2300R5] is structured concurrency. The `start_detached` and `ensure_started` algorithms are motivated by some important scenarios. Not every asynchronous operation has a clear chain of work to consume or block on the result. The problem with these algorithms is that they provide unstructured concurrency.

This is an unnecessary and unwelcome and undesirable property for concurrency. It leads to problems with lifetimes, and it requires execution contexts to conflate task lifetime management with execution management.

This paper describes an object that would be used to create a scope that will contain all senders spawned within its lifetime. These senders can be running on any execution context. The scope object has only one concern, which is to contain the spawned senders to a lifetime that is nested within any other resources that they depend on. In order to be useful within other asynchronous scopes, the object must not have any blocking functions. In practice, this means the scope serves three purposes. It:

— maintains state for launched work so that all in-flight senders have a well-defined location in which to store an *operation-state*
— manages lifetimes for launched work so that in-flight tasks may be tracked, independent of any particular execution context
— offers a join operation that may be used to continue more work, or block and wait for work, after some set of senders is complete, independent of the context on which they run.

This object would be used to spawn senders without waiting for each sender to complete.

2.1 Implementation experience

The general concept of an async scope to manage work has been deployed broadly in folly to safely launch awaitables in folly's coroutine library and in libunifex where it is designed to be used with the sender/receiver pattern.

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` is destroyed
}
```

In this example we are creating parallel work based on the given input vector. All the work will be spawned in the context of a 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 [P2300R5] (or something equivalent) to dynamically spawn work. [P2300R5] 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 the termination of the work being started. We don't have control over the lifetime of the operation being started.

At the end of the function, we are destroying the work context and the thread pool. But at that point, we don't know whether all the operations have completed. If there are still operations that are not yet complete, this might lead to crashes.

[P2300R5] doesn't give us out-of-the-box facilities to use in solving these types of problems.

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

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

    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); });
        my_work_scope.spawn(std::move(snd));           // MODIFIED!
    }
    this_thread::sync_wait(my_work_scope.on_empty());  // NEW!
    // `ctx` and `my_pool` can now safely be destroyed
}
```

The newly introduced `async_scope` object allows us to scope the lifetime of the dynamic work we are spawning. We can wait for all the work that we spawn to be complete before we destruct the objects used by the parallel work.

Please see below for more examples.

3.2 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 concurrency code. [P2300R5] 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, thus will make the program harder to understand.

To properly structure our concurrency, we need an abstraction that ensures that all the work being started has a proper lifetime guarantee. This is the goal of `async_scope`.

3.3 `async_scope` may increase consensus for P2300

Although [P2300R5] is generally considered a strong improvement on concurrency in C++, various people voted against introducing this into the C++ standard.

This paper is intended to increase consensus for [P2300R5].

4 Examples of use

4.1 Spawning work from within a task

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

```
using namespace std::execution;

system_context ctx;
int result = 0;

int main() {
  async_scope scope;
  scheduler auto sch = ctx.scheduler();

  sender auto val = on(
    sch, just() | then([sch, &scope](auto sched) {

        int val = 13;

        auto print_sender = just() | then([val]{
          std::cout << "Hello world! Have an int with value: " << val << "\n";
        });
        // spawn the print sender on sched to make sure it
        // completes before shutdown
        scope.spawn(on(sch, std::move(print_sender)));

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

  scope.spawn(std::move(val));


  // Safely wait for all nested work
  this_thread::sync_wait(scope.on_empty());

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

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

// and destruction of the context is now safe
```

4.2 Starting work nested within a framework

In this example we use the `async_scope` within a class to start work when the object receives a message and to wait for that work to complete before closing. `my_window::start()` starts the sender using storage reserved in `my_window` for this purpose.

```
using namespace std::execution;

class my_window {
  //..
```

```
  system_context ctx;
  scheduler auto sch{ctx.scheduler()};
  async_scope scope{};
};

sender auto some_work(int id);

void my_window::onMyMessage(int i) {
  this->scope.spawn(on(this->sch, some_work(i)));
}

void my_window::onClickClose() {
  this->start(this->scope.on_empty() | then([&]{this->post(close_message{});}));
}
```

4.3 Starting parallel work

In this example we use the `async_scope` within lexical scope to construct an algorithm that performs parallel work. This uses the `let_value_with` algorithm implemented in libunifex which simplifies in-place construction of a non-moveable object in the `let_value_with` algorithms operation state. Here foo launches 100 tasks that concurrently run on some scheduler provided to `foo`, through its connected receiver, and then are asynchronously joined. In this case the context the work is run on will be the `system_context`'s scheduler, from [P2079R2]. This structure emulates how we might build a parallel algorithm where each `some_work` might be operating on a fragment of data.

```
using namespace std::execution;

sender auto some_work(int work_index);

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

4.4 Calling `on_empty` multiple times

In this example we showcase how `async_scope` objects can be used as gateways between different operations multiple times. We have tabular data that needs to be processed in a clear sequence. First we need to preprocess the whole data. Then, we need to process the data by rows (each row can be processed in parallel). Then, we need to process the data by columns (each column can be processed in parallel). Finally, we post-process the tabular data.

As we are creating dynamic work for processing the rows and processing the columns, we are putting the spawned work in an `async_scope` object. As we need to process all the rows before processing columns, we will call `on_empty()` on our `async_scope` object to get notified when the processing of all the rows has completed. Similarly, to wait for the processing of the columns to complete, we can use `on_empty()` again.

It is ok if `on_empty()` is called multiple times on the same `async_scope` object. When the sender returned from `on_empty()` is started, it will complete the next time that the `async_scope` is empty.

```
struct tabular_data;
sender auto preprocess(tabular_data&);
sender auto postprocess(tabular_data&);
sender auto process_row(tabular_data&, int row);
sender auto process_col(tabular_data&, int col);

sender auto process(scheduler auto sch, tabular_data& data) {
  return schedule(sch)
       | let_value_with(
           []{ return async_scope{}; },
           [&](async_scope& scope) {
             return just()
                  // first phase: preprocess the tabular data
                  | let_value([&]{ return preprocess(data); })
                  // second phase: process the data by rows, in parallel
                  | let_value([&]{
                      for(int i = 0; i < data.num_rows(); ++i)
                        scope.spawn(on(sch, process_row(data, i)));
                      return scope.on_empty();
                  })
                  // third phase: process the data by columns, in parallel
                  | let_value([&]{
                      for(int i = 0; i < data.num_cols(); ++i)
                        scope.spawn(on(sch, process_col(data, i)));
                      return scope.on_empty();
                  })
                  // fourth phase: postprocess the data
                  | let_value([&]{ return postprocess(data); })
                  ;
           }
       )
       ;
}
```

4.5 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 `async_scope` to limit the lifetime of the request handling without blocking the acceptance of new requests.

```
task<size_t> listener(int port, io_context& ctx, static_thread_pool& pool) {
    listening_socket listen_sock{port};
    async_scope work_scope;
    size_t count{0};
    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};
        sender auto snd
            = just()
            | let_value([data = std::move(data)]() {
                  return handle_connection(data);
              })
            ;
        work_scope.spawn(std::move(snd));
    }
    // Continue only after all requests are handled
    co_await work_scope.on_empty();
    // At this point, all the request handling is complete
    co_return count;
}
```

5 Async Scope, usage guide

The requirements for the async scope are:

- — An `async_scope` must be non-movable and non-copyable.
- — An `async_scope` must be *empty* when the destructor runs.
- — An `async_scope` must introduce a cancellation scope.
- — An `async_scope` must not provide any query CPOs on the receiver passed to the sender, other than `get_stop_token()` (in order to forward cancellation of the async_scope `stop_source` to all nested and spawned senders).
- — An `async_scope` must allow an arbitrary sender to be nested within the scope without eagerly starting the sender (`nest()`).
- — An `async_scope` must constrain `spawn()` to accept only senders that complete with `void`.
- — An `async_scope` must provide an *`on-empty-sender`* that completes when all spawned senders are complete.
- — An `async_scope` must start the given sender before `spawn()` and `spawn_future()` exit.

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

5.1 Definitions

```
struct async_scope {
    async_scope();
    ~async_scope();
    async_scope(const async_scope&) = delete;
    async_scope(async_scope&&) = delete;
    async_scope& operator=(const async_scope&) = delete;
    async_scope& operator=(async_scope&&) = delete;

    template <sender_to<spawn-receiver> S>
    void spawn(S&& snd);

    template <sender S>
    spawn-future-sender<S> spawn_future(S&& snd);

    template <sender S>
    nest-sender<S> nest(S&& snd);
```

```
    [[nodiscard]]
    on-empty-sender on_empty() const noexcept;
    template <sender S>
    [[nodiscard]]
    on-empty-sender<S> when_empty(S&& snd) const noexcept;

    in_place_stop_source& get_stop_source() noexcept;
    in_place_stop_token get_stop_token() const noexcept;
    void request_stop() noexcept;
};
```

5.2 Lifetime

An `async_scope` object must outlive work that is spawned on it. It should be viewed as owning the storage for that work. The `async_scope` may be constructed in a local context, matching the syntactic scope or the lifetime of surrounding algorithms. The destructor of an `async_scope` will `terminate()` if there is outstanding work in the scope at destruction time.

Another way to view the `async_scope` is that it keeps a counter of how many senders were started with it, but have not yet completed (in execution). The destructor can be called only while this counter is zero.

One way to ensure that there are no active senders in the `async_scope` at destruction is to start the sender returned by `on_empty()` and wait for its completion. At this point, if no work has been added to the `async_scope` since the sender returned by `on_empty()` sender was started, then the `async_scope` is safe to destruct.

Note that there is a race between the completion of the sender returned by `on_empty()` and adding new work to the `async_scope` object. If new work is added from a work that is already in the scope, then the implementation guarantees that there is no race. If, however, new work is added from a different source, the implementation cannot prevent the race. For example, one can imagine that the new work is added just after the `on_empty()` sender starts.

Please see Q & A section for more details on reasons why calling `terminate()` is preferred to implicit waiting.

5.3 `spawn()`

```
template <sender_to<spawn-receiver> S> void spawn(S&& s);
```

Eagerly launches work on the `async_scope`. This involves an allocation for the *`operation-state`* of the sender until it completes.

This is similar to `start_detached()` from [P2300R5], but we keep track of the lifetime of the given work.

The given sender must complete with `void` or `stopped`. The given sender is not allowed to complete with an error; the user must explicitly handle the errors that might appear before passing the corresponding sender to `spawn()`.

As `spawn()` starts the given sender synchronously, it is important that the user provides non-blocking senders. This matches user expectations that `spawn()` is asynchronous and avoids surprising blocking behavior at runtime. The reason is that `spawn()` needs extra resources, and it's less efficient than just executing the work inline. Using `spawn()` with a sender generated by `on(sched, blocking-sender)` is a very useful pattern in this context.

Usage example:

```
// the 2 variables outlive the code below
scheduler auto sched = ...;
async_scope s;
...
```

```
for (int i=0; i<100; i++)
    s.spawn(on(sched, some_work(i)));
return s.on_empty(); // completes when all work is done
```

5.4 spawn_future()

```
template <sender S> spawn-future-sender<S> spawn_future(S&& s);
```

Eagerly launches work on the `async_scope` but returns a *`spawn-future-sender`* that represents an eagerly running task. This involves an allocation for the *`operation-state`* of the sender, until it completes, and synchronization to resolve the race between the production of the result and the consumption of the result.

This is similar to `ensure_started()` from [P2300R5], but we keep track of the lifetime of the given work.

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.

It is safe to drop the sender returned from `spawn_future()` without starting it, because the `async_scope` safely manages the lifetime of the running operations.

Please note that there is a race between the completion of the given sender and the start of the returned sender. The race will be resolved by the *`spawn-future-sender`*`<>` state.

Cancelling the returned sender, cancels `s` but does not cancel the `async_scope`.

If the given sender `s` completes with an error, but the returned sender is dropped, the error is dropped too.

Usage example:

```
// the 2 variables outlive the code below
scheduler auto sched = ...;
async_scope s;
...
sender auto snd = s.spawn_future(on(sched, key_work()))
                | then(continue_fun);
for ( int i=0; i<10; i++)
    s.spawn(on(sched, other_work(i)));
return when_all(s.on_empty(), std::move(snd));
```

5.5 nest()

```
template <sender S> nest-sender<S> nest(S&& s);
```

Returns a *`nest-sender`* that, when started, extends the lifetime of the `async_scope` that produced it to include the lifetime of the *`nest-sender`* object and the lifetime of the given sender operation.

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

The sender returned by a call to `nest()` holds a reference to the `async_scope`. Connecting and starting the sender returned from `nest()` will connect and start the input sender and will extend the `async_scope`'s lifetime to include the *`nest-sender`* and given sender operation.

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

Unlike `spawn_future()` the returned sender does not prevent the scope from ending. It is safe to drop the returned sender without starting it. It is not safe to start the sender after the `async_scope` has been destroyed.

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

One can say that `nest()` is more fundamental than `spawn()` and `spawn_future()` as the latter two can be implemented in terms of `nest()`. In terms of performance, `nest()` does not introduce any penalty. `spawn()` is more expensive than `nest()` as it needs to allocate memory for the operation. `spawn_future()` is even more expensive than `spawn()`; the receiver needs to be type-erased and a possible race condition needs to be avoided. `nest()` does not require allocations, so it can be used in a free-standing environment.

Cancelling the returned sender, once it is connected and started, cancels `s` but does not cancel the `async_scope`.

Usage example:

```
// the 2 variables outlive the code below
scheduler auto sched = ...;
async_scope s;
...
sender auto snd = s.nest(key_work());
for ( int i=0; i<10; i++)
    s.spawn(on(sched, other_work(i)));
// will deadlock: this_thread::sync_wait(s.on_empty());
return when_all(s.on_empty(), std::move(snd)); // OK, completing snd will also complete s.on_empty()
```

5.6 Empty detection

```
template <sender S> on-empty-sender<S> when_empty(S&& s);
```

An `async_scope` object is considered to be *non-empty* when there are spawned senders that haven't completed yet, or there are senders created with `async_scope` that are in flight. The object is considered *empty* otherwise. An `async_scope` can be *empty* more than once.

`on-empty-sender` starts the given sender when the `async_scope` object becomes *empty* and completes when the given sender completes. This can be used to run async cleanup for the resources used by the spawned senders.

The intended usage is to spawn all the senders and then start the *`on-empty-sender`* to know when all spawned senders have completed.

If the `async_scope` object is requested to stop, the returned *`on-empty-sender`* is not cancelled. This ensures that the `async_scope` is not reported as empty until all active senders complete after a stop is requested to the `async_scope` object.

To safely destroy the `async_scope` object it's recommended to use *`on-empty-sender`* to get notified when the scope object finished executing all the work. Moreover, after starting *`on-empty-sender`* for the purpose of detecting when it is safe to destroy an `async_scope`, all calls `nest()`, `spawn()` and `spawn_future()` must *happen_before* the *`on-empty-sender`* was started.

That is to say that the following is safe:

```
{
  async_scope s;
  s.spawn(snd);
  sync_wait(s.on_empty());
}
```

Usage example:

```
sender auto run_in_parallel(int num_jobs, async_scope& scope, scheduler auto& sched) {
  // Create parallel work
  for ( int i=0; i<num_jobs; i++ )
    scope.spawn(on(sched, some_work(i)));
  // Join the work with the help of the scope
```

```
  return scope.on_empty();
}
```

```
on-empty-sender on_empty() const noexcept;
```

Equivalent to calling `when_empty(just())`

5.7 Stopping `async_scope`

```
in_place_stop_source& get_stop_source() noexcept;
```

Returns a `in_place_stop_source` associated with the `async_scope`'s `stop_token`. This `in_place_stop_source` will trigger the `in_place_stop_token`, and will cause future calls to `nest()`, `spawn()` and `spawn_future()` to start with a `in_place_stop_token` that is already in the `stop_requested()` state.

Calling `request_stop` on the returned `stop_source` will forward that request to all the nested and spawned senders.

```
in_place_stop_token get_stop_token() const noexcept;
```

Equivalent to calling `get_stop_source().get_token()`.

Returns the `in_place_stop_token` associated with the `async_scope`. This will report stopped when the `stop_source` is stopped or `request_stop()` is called. The `in_place_stop_token` is provided to all nested and spawned senders so that they are able to respond to a stop request.

```
void request_stop() noexcept;
```

Equivalent to calling `get_stop_source().request_stop()`.

Usage example:

```
int main() {
    async_scope scope;
    sender auto program = program_work(scope);
    sender auto program_with_termination
        = std::move(program)
        // Ensure we don't run any dynamic tasks before exiting
        // and wait for ongoing work to complete
        | let_value([&]{
            scope.request_stop();
            return scope.on_empty();
        })
        ;
    this_thread::sync_wait(program_with_termination);
    // all work, static or dynamic is completed at this point
    return 0;
}
```

6 Design considerations

6.1 Shape of `async_scope`

6.1.1 Concept vs type

One option is to have an `async_scope` concept that has many implementations.

Another option is to have a type that has one implementation per library vendor.

Chosen: Due to time constraints, this paper proposes a type.

6.1.2 One vs many

One option would be for `async_scope` to have:

— `template <sender S>nest-sender<S> nest(S&&)`

and not

— `template <sender S> void spawn(S&&)`
— `template <sender S> spawn-future-sender<S> spawn_future(S&&)`

This would remove questions of when and how the state is allocated and the operation started from the scope.

The single concern of the `async_scope` that only had `nest()` would be to combine the lifetimes of many senders within one async scope.

`spawn()` and `spawn_future()` would still exist, in some form, and would use an `async_scope` parameter or member or base class to place the sender within an `async_scope`.

Another option is to add `spawn()` and `spawn_future()` methods to `async_scope`.

Chosen: Due to time constraints, this paper proposes to add methods for `spawn` and `spawn_future` in addition to `nest`.

6.1.3 Customization point object vs method

One option is to define Customization Point Objects for `nest`, `spawn`, `spawn_future`, and `when_empty` that operate on anything that customizes those objects.

Another option is to define a type with `nest`, `spawn`, `spawn_future` and `when_empty` methods.

Chosen: methods on a type.

6.1.4 Phased types vs Mono type

One option would be for `async_scope` to have:

— `template <sender S>nest-sender<S> nest(S&&)`

and add `async_scope_token`. `async_scope_token` would consume an `async_scope` in its constructor. Transitioning an `async_scope` to an `async_scope_token` would end the nesting phase and begin the completion phase. The `async_scope_token` would have:

— `template <sender S>empty-sender<S> when_empty(S&&)`

`when_empty()` will connect and start the given sender when all the nested senders complete.

Chosen: Due to time constraints, this paper proposes a Mono type.

6.2 Shape of input senders

6.2.1 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.

6.2.2 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 `async_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 spotting bugs too late.

> **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.

6.2.3 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 `async_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([]{ return 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()`.

6.2.4 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.

6.3 Stop handling

The paper requires that if the caller requests stop to an `async_scope` object, then this request is forwarded to the nested and spawned senders.

6.3.1 Alternative 1: `request_stop()` on the `async_scope` is forwarded

When stop is requested to `async_scope`, then stop is also requested to operations that are not yet complete. While this can be a good thing in many contexts, it is not the best strategy in all cases.

Consider an `async_scope` that is used to keep track of the work needed to handle requests. When trying to gracefully shut down the application, one might need to drain the active senders without stopping their processing. The way to do that is to use the *`on-empty-sender`* without stopping the `async_scope`.

Consider `spawn()` in isolation. Forwarding the cancellation of the `async_scope` to the spawned senders would be natural.

Consider `nest()` and `spawn_future()`. They must combine two potential stop tokens. One from the `async_scope` and the other from the receiver passed to the returned *`nest-sender`* and *`spawn-future-sender`*.

The semantics would be that either stop token would cancel the sender and would not stop the `async_scopes` `stop_source`.

Consider the use case where a reference to an `async_scope` is provided to many nested operations and functions to attach senders that they produce. Some of those senders may restore an invariant in a file-system or some other system. The way for a nested operation and function to make sure that the invariant is not corrupted by a forwarded stop request from the `async_scope`, is to apply a `never_stoppable_token` to their sender to hide the token provided by the `async_scope`.

6.3.2 Alternative 2: `request_stop()` on the `async_scope` is not forwarded

A motivation for not forwarding a stop request was that a `stop_callback` is not a destructor, it is a signal requesting running work to stop. If `request_stop()` was called within the `async_scope` destructor, or any other destructor, then those destructors would be expected to block until an *`on-empty-sender`* completed. As falling off a scope or having a shared_ptr count reach 0 is implicit, it is very difficult to ensure that a `request_stop()` followed by starting an *`on-empty-sender`* would not have a race with concurrent calls to `nest()`, `spawn()` and `spawn_future()`.

6.3.3 Inverting the forwarding default

Either of the two cases can be simulated with the help of the other case.

Example: When cancellation is not forwarded and forwarding is wanted, inject the same `stop_token` into all the spawned senders that need to be cancelled.

Example: When cancellation is forwarded and forwarding is not wanted, mask the receiver provided `stop_token` by injecting a `never_stoppable_token` into all the spawned senders that need to complete even when cancelled.

6.3.4 Result

Chosen: `request_stop()` on the `async_scope` is forwarded.

6.4 Uses in other concurrent abstractions

In its most basic form the interface `async_scope` applies to other concurrent abstractions. This implies that it is useful to think of this interface in a larger context. If the interface is fit for the other purposes, it may be an indication that we have the right interface and that we should add a concept for that interface.

Let us consider a concurrent abstraction that will serialize dynamic work provided to it. That is, if try to start multiple operations at the same time, only one is executed at a given time; the other ones are queued and will be executed whenever the previous operations complete.

An interface to this abstraction might look like the following:

```
struct async_mutex {
    async_mutex();
    ~async_mutex();
    async_mutex(const async_mutex&) = delete;
    async_mutex(async_mutex&&) = delete;
    async_mutex& operator=(const async_mutex&) = delete;
    async_mutex& operator=(async_mutex&&) = delete;

    template <sender S>
    lock-sender<S> lock(S&& snd);
};
```

One can add a sender in the context of the `async_mutex` by passing it to `lock()`. Starting the sender returned from `lock()` will add the given sender to the queue waiting for the lock. One might want to add some work that needs to be executed in the `async_mutex`, then continue with some other work outside the `async_mutex`. One might want to wait until the `async_mutex` is drained, or might want to stop processing any work in the `async_mutex`. All of these can be fulfilled by composing `async_mutex` and `async_scope`. `async_scope::nest()` is the same basis operation as `async_mutex::lock`. If a name that would work for both `nest()` and `lock()` was used, then this would be the basis function for a new concept.

Similar to this abstraction, one might imagine abstractions that can execute maximum N concurrent work items, or abstractions that execute work based on given labels, or abstractions that execute work based on dynamic priorities, etc. All of these can be obtained by using an interface similar to the one we have for `async_scope`, maybe with some extra arguments.

This provides a strong indication that the API for `async_scope` is appropriate.

6.5 P2300's `start_detached()`

The `spawn()` method in this paper can be used as a replacement for `start_detached` proposed in [P2300R5]. Essentially it does the same thing, but it can also scope the lifetime of the spawned work.

6.6 P2300's `ensure_started()`

The `spawn_future()` method in this paper can be used as a replacement for `ensure_started` proposed in [P2300R5]. Essentially it does the same thing, but it can also scope the lifetime of the spawned work.

6.7 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:

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

In [P2300R5] sender consumers do not have support for the pipe operator. As `spawn()` works similarly to `start_detached()` from [P2300R5], 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. To keep consistency with `spawn()`, at this point the paper doesn't support pipe operator for `spawn_future()`.

If `spawn_future()` was an algorithm and the `spawn_future()` method was removed from `async_scope`, then the pipe operator would be a natural and obvious fit.

7 Q & A

7.1 Why does `async_scope` terminate in the destructor instead of blocking like `jthread`?

- `jthread` blocking in the destructor is bad for composition.
- `jthread` and `thread` should `terminate()` if the destructor runs before the thread exits.

Imagine `make_shared<jthread>(...)`. Where will the destructor run? In what context will the destructor run?

We can require users to know whether the destructor blocks for every type, and require users to carefully control the lifetime of all those objects – with the only indication of failure being a deadlock. Or we can teach that destructors will not block and indicate lifetime failures with `terminate()`.

One authors philosophy is that software is less likely to ship with crashes and more likely to be fixed when there are crashes. Deadlocks result in users forcefully terminating the app and forced terminations are rarely reported to the developer as a bug and even if reported, tend to have no debug data (stacks, dumps, etc…). If there is a lifetime bug that you want fixed – it had better crash.

Principles that lead to avoid blocking in the destructor:

- Blocking must be explicit (exiting a sync scope is implicit – and `shared_ptr` makes it even more scary as the destructor will potentially run at a different point each time).
- Blocking must be grepable.
- Blocking must be rare.
- Blocking must be composable.
- Blocking is like `reinterpret_cast<>` – the name should be long and scary.
- `join()` is grepable and explicit, it is not rare, it is not composable (There is a separate blocking wait for each. One blocking wait for many different things to complete would be better)– this is why async_scope has `when_empty()` instead.

Every asynchronous operation must join with non-blocking primitives and only `sync_wait()` is used to block some composition of those primitives.

7.2 Why doesn't the `async_scope` destructor stop all the nested and spawned senders?

- `stop_callback` is not a destructor because:
  - `request_stop()` is **asking** for early completion.
  - `request_stop()` does not end the lifetime of the operation, `set_value()`, `set_error()` and `set_stopped()` end the lifetime – those are the destructors for an operation.
  - `request_stop()` might result in completion with `set_stopped()`, but `set_value()` and `set_error()` are equally valid.

`request_stop()` should not be called from a destructor because: If a sync context intends to ask for early completion of an async operation, then it needs to wait for that operation to actually complete before continuing (`set_value()`, `set_error()` and `set_stopped()` are the destructors for the async operation), and sync destructors must not block. *See* Why does `async_scope` terminate in the destructor instead of blocking like `jthread`?.

> NOTE: async RAII could be used to signal early completion because it would be composed with other async operation lifetimes. The operation being stopped would complete before the async RAII operation completed – without any blocking.

8 Naming

As is often true, naming is a difficult task.

8.1 `async_scope`

This 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 a state).

alternatives: `sender_scope`, `sender_anchor`, `sender_nursery`

rejected: `dynamic_scope`, `dynamic_lifetime`, `scope`, `lifetime`

8.2 `nest()`

This provides a way to build a sender that, when started, extends the lifetime of the `async_scope` to include the given sender. This does not allocate state, call connect or call start. This is the basis operation for `async_scope`. `spawn()` and `spawn_future()` use `nest()` to extend the scope and then they allocate, connect and start the returned [**__nest-sender__?**]@.

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

If this becomes a basis operation for a new concept, it might be good to use a name that would also work for things like `async_mutex`. *See* Uses in other concurrent abstractions

alternatives: `add()`, `extend_with()`, `adopt()`, `attach()`, `enter()`

rejected: `embed()`, `include()`, `constrain()`, `apply()`

8.3 `spawn()`

This provides a way to start a sender that produces `void` and extend the lifetime of the `async_scope` to exceed the lifetime of the operation. This allocates, connects and starts the given sender.

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

alternatives: `start_sender()`, `connect_and_start()`

rejected: `start()`, `submit()`, `enqueue()`, `run()`

8.4 `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 very expensive operation.

alternatives: `spawn_continue()`, `spawn_result()`, `spawn_with_result()`, `spawn_buffered()`, `spawn_virtual()`, `spawn_dynamic()`

> *Note*: "`spawn`" in these alternatives would be replaced by the alternative selected for `spawn()`

8.5 `when_empty()` (and `on_empty()`)

`when_empty()` provides a way to start a given sender when all the activity nested inside the `async_scope` is complete.

The alternative `empty` falls out of the poor mental model of `async_scope` being a container. The alternatives `ended`, `complete` etc.. are problematic because additional senders might be used to extend the lifetime after the sender returned has completed.

`on_empty()` is the async version of a 'get' member function. A pattern was established a long time ago to not prefix 'get' methods on an object in `std` with `get_`. What is the current guidance? Do we want a prefix for async queries on objects in std?

alternatives: `empty`, `ready`, `inactive`, `when_ready`, `upon_empty`, `upon_ready`

8.6 table of how some alternatives might be combined

id	comments	nest	spawn void	spawn w/result	empty
a:	status quo	nest	spawn	spawn_future	when_empty
b:	removes confusion around “future”, “empty” and “nest”	add	spawn	spawn_continue	when_empty
c:	tries to match start_detached() in [P2300R5]	add	start	start_continue	when_empty
d:	tries an alternative to using “continue”	add	start	start_chain	when_empty
e:	tries an alternative “result” and “extend” and “ready”	extend	start	start_result	upon_ready
f:	verbose sender_scope	extend_with	connect_and_start	spawn_with_result_synchronized	when_empty
g:	sender_anchor	attach	launch	launch_with_result_synchronized	when_empty
h:	sender_nursery	enter	spawn	spawn_with_result_synchronized	when_empty

9 Specification

9.1 Synopsis

```
namespace std::execution {

namespace { // exposition-only
    struct spawn-receiver { // exposition-only
        friend void set_value(spawn-receiver) noexcept;
        friend void set_stopped(spawn-receiver) noexcept;
    };
    template <typename S>
    struct nest-sender; // exposition-only
    template <typename S>
    struct spawn-future-sender; // exposition-only
    template <typename S>
    struct on-empty-sender; // exposition-only
}

struct async_scope {
    async_scope();
    ~async_scope();
    async_scope(const async_scope&) = delete;
    async_scope(async_scope&&) = delete;
    async_scope& operator=(const async_scope&) = delete;
    async_scope& operator=(async_scope&&) = delete;

    template <sender_to<spawn-receiver> S>
    void spawn(S&& snd);

    template <sender S>
    spawn-future-sender<S> spawn_future(S&& snd);
```

```
    template <sender S>
    nest-sender<S> nest(S&& snd);

    template <sender S>
    [[nodiscard]]
    on-empty-sender<S> when_empty(S&& snd);
    [[nodiscard]]
    on-empty-sender on_empty() const noexcept;

    in_place_stop_source& get_stop_source() noexcept;
    in_place_stop_token get_stop_token() const noexcept;
    void request_stop() noexcept;
};

}
```

9.2 `async_scope::async_scope`

1. `async_scope::async_scope` constructs the `async_scope` object, in the empty state.
2. *Note*: It is always safe to call the destructor immediately after the constructor, without adding any work to the `async_scope` object.

9.3 `async_scope::~async_scope`

1. `async_scope::~async_scope` destructs the `async_scope` object, freeing all resources
2. The destructor will call `terminate()` if there is outstanding work in the `async_scope` object (i.e., work created by `nest()`, `spawn()` and `spawn_future()` did not complete).
3. *Note*: It is always safe to call the destructor after the sender returned by `on_empty()` sent the completion signal, provided that there were no calls to `nest()`, `spawn()` and `spawn_future()` since the *`on-empty-sender`* was started.

9.4 `async_scope::spawn`

1. `async_scope::spawn` is used to eagerly start a sender while keeping the execution in the lifetime of the `async_scope` object.
2. *Effects*:
   - An *`operation-state`* object `op` will be created by connecting the given sender to a receiver `recv` of type *`spawn-receiver`*.
   - If an exception occurs while trying to create `op` in its proper storage space, the exception will be passed to the caller.
   - If no exception is thrown while creating `op` and stop was not requested on our stop source, then:
     - `start(op)` is called (before `spawn()` returns).
     - The lifetime of `op` extends at least until `recv` is called with a completion notification.
   - `recv` supports the `get_stop_token()` query customization point object; this will return the stop token associated with `async_scope` object.
   - The `async_scope` will not be *empty* until `recv` is notified about the completion of the given sender.
3. *Note*: the receiver will help the `async_scope` object to keep track of how many operations are running at a given time.

9.5 `async_scope::spawn_future`

1. `async_scope::spawn_future` is used to eagerly start a sender in the context of the `async_scope` object, and returning a sender that will be triggered after the completion of the given sender. The lifetime of the

returned sender is not associated with `async_scope`.

2. The returned sender has the same completion signatures as the input sender.

3. *Effects*:

   - An *`operation-state`* object `op` will be created by connecting the given sender to a receiver `recv`.
   - If an exception occurs while trying to create `op` in its proper storage space, the exception will be passed to the caller.
   - If no exception is thrown while creating `op` and stop was not requested on our stop source, then:
     - `start(op)` is called (before `spawn_future` returns).
     - The lifetime of `op` extends at least until `recv` is called with a completion notification.
     - If `rsnd` is the returned sender, then using it has the following effects:
       - Let `ext_op` be the *`operation-state`* object returned by connecting `rsnd` to a receiver `ext_recv`.
       - If `ext_op` is started, the completion notifications received by `recv` will be forwarded to `ext_recv`, regardless whether the completion notification happened before starting `ext_op` or not.
       - It is safe not to connect `rsnd` or not to start `ext_op`.
     - The `async_scope` will not be *empty* until one of the following is true:
       - `rsnd` is destroyed without being connected
       - `rsnd` is connected but `ext_op` is destroyed without being started
       - If `rsnd` is connected to a receiver to return `ext_op`, `ext_op` is started, and `recv` is notified about the completion of the given sender
   - `recv` supports the `get_stop_token()` query customization point object; this will return a stop token object that will be stopped when:
     - the `async_scope` object is stopped (i.e., by using `async_scope::request_stop()`;
     - if `rsnd` supports `get_stop_token()` query customization point object, when stop is requested to the object `get_stop_token(rsnd)`.

4. *Note*: the receiver `recv` will help the `async_scope` object to keep track of how many operations are running at a given time.

5. *Note*: the type of completion signal that `op` will use does not influence the behavior of `async_scope` (i.e., `async_scope` object behaves the same way if the sender describes a work that ends with success, error or cancellation).

6. *Note*: cancelling the sender returned by this function will not have an effect about the `async_scope` object.

9.6 `async_scope::nest`

1. `async_scope::nest` is used to produce a *`nest-sender`* that, when started, nests the sender within the lifetime of the `async_scope` object. The given sender will be started when the *`nest-sender`* is started.

2. The returned sender has the same completion signatures as the input sender.

3. *Effects*:

   - If `rsnd` is the returned *`nest-sender`*, then using it has the following effects:
     - Let `op` be the *`operation-state`* object returned by connecting the given sender to a receiver `recv`.
     - Let `ext_op` be the *`operation-state`* object returned by connecting `rsnd` to a receiver `ext_recv`.
     - Let `op` be stored in `ext_op`.
     - If `ext_op` is started, then `op` is started and the completion notifications received by `recv` will be forwarded to `ext_recv`.
     - *Note*: as `op` is stored in `ext_op`, calling `nest()` cannot start the given sender.
     - Once `rsnd` is connected and `ext_op` started the `async_scope` will not be empty until `recv` is notified about the completion of the given sender.

- `recv` supports the `get_stop_token()` query customization point object; this will return a stop token object that will be stopped when:
  - the `async_scope` object is stopped (i.e., by using `async_scope::request_stop()`;
  - if `rsnd` supports `get_stop_token()` query customization point object, when stop is requested to the object `get_stop_token(rsnd)`.

4. *Note*: the type of completion signal that `op` will use does not influence the behavior of `async_scope` (i.e., `async_scope` object behaves the same way if the sender completes with success, error or cancellation).

5. *Note*: cancelling the sender returned by this function will not cancel the `async_scope` object.

9.7 `async_scope::when_empty`

1. `async_scope::when_empty` is used to produce a *`on-empty-sender`* that can be used to get notifications when all the work belonging to the `async_scope` object is completed. The given sender will be started when the `async_scope` object becomes empty after the `on-empty-sender_` is started. `on-empty-sender_` will complete when the given sender completes.

2. *Effects*:

   - If `rsnd` is the returned `on-empty-sender_`, then using it has the following effects:
     - Let `op` be the *`operation-state`* object returned by connecting the given sender to a receiver `recv`.
     - Let `ext_op` be the *`operation-state`* object returned by connecting `rsnd` to a receiver `ext_recv`.
     - Let `op` be stored in `ext_op`.
     - If `ext_op` is started, then `op` will be started and the completion notifications received by `recv` will be forwarded to `ext_recv` whenever all the work started in the context of the `async_scope` object (by using `spawn()` and `spawn_future()` or by using connecting and starting the sender returned from a call to `nest()`) is completed, and no senders are active.
     - *Note*: as `op` is stored in `ext_op`, calling `when_empty()` cannot start the given sender.
   - `recv` supports the `get_stop_token()` query customization point object; this will return a stop token object that will be stopped, when stop is requested to the object `get_stop_token(rsnd)`.
   - It is safe not to connect `rsnd` or not to start `ext_op`.

3. *Note*: it is safe to call `when_empty()` multiple times on the same object and use the returned sender; it is also safe to use the returned senders in parallel.

4. *Note*: it is safe to call `when_empty()` and use the returned sender in parallel to calling `nest()`, `spawn()` and `spawn_future()` on the same `async_scope` object.

5. *Note*: there is a race between the start of the returned `on-empty-sender_` and adding new work into the scope (from senders that are not active in the `async_scope` object). The returned sender might indicate that the `async_scope` is empty at the same time, or immediately after new work is added to it.

9.8 `async_scope::on_empty`

1. This is equivalent to calling `when_empty(just())`.

9.9 `async_scope::get_stop_source`

1. Returns an `in_place_stop_source` object associated with `async_scope`.
2. Requesting stop on the returned stop source will have the following effects:
   - work added to the `async_scope` object by using `nest()`, `spawn()` and `spawn_future()` is given a stop token that already has `stop_requested() == true`.
   - stop is requested for all the ongoing work added to `async_scope` by means of `nest()`, `spawn()` and `spawn_future()`.

9.10 `async_scope::get_stop_token`

1. This is equivalent to calling `get_stop_source().get_token()`.

9.11 `async_scope::request_stop`

1. This is equivalent to calling `get_stop_source().request_stop()`.

10 References

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

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

[P2300R5] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2022-04-22. ‘std::execution‘.
https://wg21.link/p2300r5