Asynchronous API

Below, different variations of asynchronous APIs built on top of C-style callbacks with examples of doing 2 GET requests - both sequentially and concurrently are showcased.

NO threads are involved intentionally, to disconnect any associations of coroutines or fibers with multithreading. Some parts are deliberately simple, while still presenting as much details as possible.

Jump to examples for blocking requests, callbacks, tasks .then(), std::future (polling), coroutines, fibers, senders.

Work In Progress.

View: HTML, PDF, source code.

Contents

---

1 #introduction

We start with a simple C-style API on top of libcurl C API and have a code that may look like this:

```
1 // our CURL API
2 std::string CURL_get(const std::string& url);
3
```

```
int main()
{
    const std::string r1 = CURL_get("localhost:5001/file1.txt");
    const std::string r2 = CURL_get("localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
}
```

The code above performs two GET requests sequentially. Everything executes synchronously.

Next, lets have a simple C-style callbacks API (**not** the C++ one, see the note), to run requests concurrently:

```
// libcurl bookkeeping
using CURL_Async = void*;
CURL_Async CURL_async_create();
void CURL_async_destroy(CURL_Async curl_async);
void CURL_async_tick(CURL_Async curl_async);

// main async callback API
void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response))
        noexcept;
```

(Note, CURL_async_get() is noexcept intentionally, for simplicity, see error handling).

Doing 2 GET requests is more involved now:

```
int main()
{
    struct State
    {
        int count = 0;
        std::string r1;
        std::string r2;
    };

    CURL_Async curl_async = CURL_async_create();
    State state;
    CURL_async_get(curl_async, "localhost:5001/file1.txt", &state
        , [](void* user_data, std::string response)
    {
        State& state = *static_cast<State*>(user_data);
        state.count += 1;
```

```
        state.r1 = std::move(response);
    });
    CURL_async_get(curl_async, "localhost:5001/file2.txt", &state
        , [](void* user_data, std::string response)
    {
        State& state = *static_cast<State*>(user_data);
        state.count += 1;
        state.r2 = std::move(response);
    });
    while (state.count != 2) // wait for 2 requests to finish
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
    std::println("{}", state.r1);
    std::println("{}", state.r2);
}
```

There is a need to have a `State` for bookkeeping, pass it as a `void*` user data to access later and, finally, run an event loop to give libcurl a chance to process requests. Note, however, requests execute concurrently now, as in - 2 requests are active at the same time.

After this, lets build other variations of asynchronous API on top of C-style callbacks above:

```
std::string CURL_get(const std::string& url);

void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response))
        noexcept;

Co_CurlAsync CURL_await_get(
    CURL_Async curl_async, const std::string& url);
Co_Task<std::string> CURL_coro_get(
    CURL_Async curl_async, std::string url);

std::string CURL_fiber_get(
    CURL_Async curl_async, const std::string& url);
FiberTask<std::string> CURL_fiber_get(
      CURL_Async curl_async
    , const std::string& url
    , FiberTaskScheduler& fiber_scheduler);

```

```
std::future<std::string> CURL_future_get(
    CURL_Async curl_async, const std::string& url);
Task<std::string> CURL_task_get(
    CURL_Async curl_async, const std::string& url);

CURL_Get_Sender CURL_sender_get(
    CURL_Async curl_async, const std::string& url);
```

But before that, lets wrap libcurl C API for our needs.

---

2 #setup with CMake + libcurl

source code.

For vcpkg, there is an extensive documentation available. In short:

```
git clone https://github.com/microsoft/vcpkg
cd vcpkg
bootstrap-vcpkg.bat
```

Assuming vcpkg is installed at `K:\vcpkg`, we also set VCPKG_ROOT and add this path to `PATH` environment variable so build scripts can use `VCPKG_ROOT` and `vcpkg` without a need to know the exact location.

```
set VCPKG_ROOT=K:\vcpkg
set PATH=%VCPKG_ROOT%;%PATH%
```

For the project (lets say in a `K:\async_api` folder), vcpkg manifest mode is used. Together with `curl` setup, all required steps are

```
cd K:\async_api
vcpkg new --application
vcpkg add port curl
```

Note that to find exact `curl` package name, `vcpkg search curl` was used which prints:

> curl 8.13.0#1 A library for transferring data with URLs

CMakeLists.txt now looks like this:

```
cmake_minimum_required(VERSION 3.24 FATAL_ERROR)
project(async_api LANGUAGES CXX)

add_executable(CH00_cmake main.cc)
target_compile_features(CH00_cmake PUBLIC cxx_std_23)
```

```
find_package(CURL REQUIRED)
target_link_libraries(CH00_cmake PRIVATE CURL::libcurl)
```

`find_package(CURL REQUIRED)` syntax together with `CURL::libcurl` target name is found from the output log of `vcpkg install curl` (or during CMake configuration run) which prints:

```
curl is compatible with built-in CMake targets:

    find_package(CURL REQUIRED)
    target_link_libraries(main PRIVATE CURL::libcurl)
```

To test that everything compiles and links, save main.cc:

```
#include <curl/curl.h>

int main()
{
    CURL* curl = curl_easy_init();
    assert(curl);
    curl_easy_cleanup(curl);
}
```

Finally, to invoke CMake configure, build and run (with vcpkg):

```
cd K:\async_api
cmake -S . -B __build ^
  -DCMAKE_TOOLCHAIN_FILE=%VCPKG_ROOT%\scripts\buildsystems\vcpkg.cmake
cmake --build __build --config Debug
:: run a test
.\__build\Debug\CH00_cmake.exe
```

This assumes `cmake.exe` is in your `PATH`, see `build.cmd`.

3 #building blocking API

source code, sample app.

Blocking, synchronous API for a GET request is straightforward. We go with a function that looks like this:

```
std::string CURL_get(const std::string& url);
```

libcurl comes with two different APIs, "easy" and "multi". Lets use an easy interface; libcurl examples available online, including official simple.c example for a start.

Everything together leads to the implementation below, where `curl_easy_perform()` call is the main one that blocks the execution until request completes; once

complete, we can return results:

```cpp
#include <string>

#include <curl/curl.h>

#if defined(NDEBUG)
#  undef NDEBUG
#endif
#include <cassert>

static size_t CURL_OnWriteCallback(
    void* ptr, size_t size, size_t nmemb, void* data)
{
    std::string& response = *static_cast<std::string*>(data);
    response.append(static_cast<const char*>(ptr), size * nmemb);
    return (size * nmemb);
}

std::string CURL_get(const std::string& url)
{
    CURL* curl = curl_easy_init();
    assert(curl);

    CURLcode status = curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
    assert(status == CURLE_OK);
    status = curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
    assert(status == CURLE_OK);

    std::string response;
    status = curl_easy_setopt(curl
        , CURLOPT_WRITEFUNCTION, CURL_OnWriteCallback);
    assert(status == CURLE_OK);
    status = curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response);
    assert(status == CURLE_OK);

    status = curl_easy_perform(curl);
    assert(status == CURLE_OK);

    long response_code = -1;
    status = curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &response_code);
    assert(status == CURLE_OK);
    assert(response_code == 200L);

    curl_easy_cleanup(curl);
    return response;
```

```
}
```

Note on error handling: for now, we crash on any unexpected error - as in "crash the whole application". `assert()` is enabled always **intentionally** to simplify both, the sample code **and** debugging:

```
// after all includes, main.cc
#if defined(NDEBUG)
#  undef NDEBUG
#endif
#include <cassert>
```

This is "bad" for generic, low-level library API/code, but could be fine sometimes. We'll discuss error handling later.

To see the code in action, lets run our program:

```
#include <print>

int main()
{
    const std::string r = CURL_get("localhost:5001/file1.txt");
    std::println("{}", r);
}
```

that.. should crash since we don't have local HTTP server running to serve `localhost:5001/file1.txt`. See the next section on how to make it happen.

Once done, we should see the sample file1.txt content in the console output:

```
Fox
```

3.1 #run simple http server for tests

source code.

To run sample code, lets use Python to have simple HTTP server that hosts files in the current directory, see `serve.cmd`:

```
python -m http.server 5001
```

Given the directory that has file1.txt and file2.txt `CURL_get("localhost:5001/file1.txt")` should work and return the content of the file, see blocking libcurl section.

4 #building C-style callbacks API

source code, sample app.

4.1 #thoughts on the design

Now, lets imagine simplest possible asynchronous API. The difference to blocking API is that we ask the system to start a GET request and the response could arrive some time later. The system invokes a user-provided `callback` to notify us once everything is done:

```
void CURL_async_get(const std::string& url
    , void (*callback)(std::string))
        noexcept;
```

we could use it like this:

```
// start a request:
CURL_async_get("localhost:5001/file2.txt"
    , [](std::string response)
{
    // probably, some time later:
    std::println("got response: {}", response);
});
```

There are multiple issues with the design above:

1. Where is the "system" that starts the request? It could be implicit, hidden global singleton, but we can also ask a user to explicitly create and pass it around.
2. The callback accepts only `response`, there is no way for a user to access other data, without resorting to global singletons again. When starting a request, user should be able to provide opaque pointer to some data that system does not touch and simply gives it back in callback.
3. When and from where the "system" invokes a `callback`? There are multiple answers, but we go with user-controlled event loop that drives everything.

To solve first issue, lets have explicit API to create and destroy the system:

```
using CURL_Async = void*; // system's state

CURL_Async CURL_async_create();
void CURL_async_destroy(CURL_Async curl_async);
```

where `CURL_Async` is the system itself, since user does not care what's that exactly, it's hidden under `void*`. User creates the system, uses it and, once not needed, destroys to clean up resources, if any.

To drive a system with event loop, user must call the next API:

```
void CURL_async_tick(CURL_Async curl_async);
```

This is the chance for a system to actually do some work over time **and** invoke

user-provided callbacks, if needed.

Lastly, to give a user some control over data in the callback, we pass opaque `void*` pointer around:

```
// main async callback API
void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response))
        noexcept;
```

`user_data` could be anything, system gives it back when invoking `callback`. This is user responsibility to ensure that pointer is valid all the time while request is in progress.

There are more nuances, like how frequently/when `CURL_async_tick()` should be invoked by a user; but all this is left out of the scope.

Overall, everything included, we need to implement the next API, see below:

```
// libcurl bookkeeping
using CURL_Async = void*;
CURL_Async CURL_async_create();
void CURL_async_destroy(CURL_Async curl_async);
void CURL_async_tick(CURL_Async curl_async);

// main async callback API
void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response))
        noexcept;
```

4.2 #note on C-style API (vs C++)

For C-style API above, with C++, "the system" could be a class, callback could be `std::function<>` to accept anything, generally making it less verbose, having something like this:

```
// the API:
class CURL_Async
{
public:
    void get(const std::string& url, std::function<void (std::string)>);
    void tick();
};

// the use:
```

```
CURL_Async curl;
curl.get("localhost:5001/file1.txt", [](std::string r)
{
    std::println("{}", r);
});
curl.tick(); // etc
```

However, C-style API we have, is a de-facto standard, familiar and recognized for asynchronous APIs with callbacks (citation needed).

The rest of asynchronous APIs implementations below are built on top of C-style callback API, as a basic building block to cover similar callbacks-based APIs.

4.3 #implementing with libcurl multi

source code, sample app.

For our callbacks API:

```
using CURL_Async = void*;
CURL_Async CURL_async_create();
void CURL_async_destroy(CURL_Async curl_async);
void CURL_async_tick(CURL_Async curl_async);
void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response))
        noexcept;
```

internally, lets have `CURL_AsyncScheduler` class to handle adding requests, updating/ticking libcurl event loop and, in general, to represent our whole `CURL_Async` system state:

```
struct CURL_AsyncScheduler
{
    CURL_AsyncScheduler();
    ~CURL_AsyncScheduler();
    // no copy, no move
    CURL_AsyncScheduler(const CURL_AsyncScheduler&) = delete;

    using Callback = std::function<void (CURL* curl_easy)>;

    void tick();
    void add_request(CURL* curl_easy, Callback on_finish);

    // our state
    CURLM* _multi_curl = nullptr;
    std::unordered_map<CURL*, Callback> _curl_to_callback;
```

```
};
```

This is what we'll return to a user as `CURL_Async` pointer. Lets do it:

```
CURL_Async CURL_async_create()
{
    CURL_AsyncScheduler* scheduler = new(std::nothrow) CURL_AsyncScheduler();
    assert(scheduler);
    return scheduler;
}

void CURL_async_destroy(CURL_Async curl_async)
{
    assert(curl_async);
    CURL_AsyncScheduler* scheduler =
        static_cast<CURL_AsyncScheduler*>(curl_async);
    delete scheduler;
}
```

Now, user just needs to pass `CURL_Async` handle around. Before implementing internals, lets have a helper function that gets actual `CURL_AsyncScheduler` instance from opaque handle:

```
CURL_AsyncScheduler& CURL_scheduler(CURL_Async curl_async)
{
    CURL_AsyncScheduler* scheduler =
        static_cast<CURL_AsyncScheduler*>(curl_async);
    assert(scheduler);
    return *scheduler;
}
```

It's not exposed to the user in any way. Lets implement our main API in terms of our internal scheduler:

```
void CURL_async_get(CURL_Async curl_async
    , const std::string& url
    , void* user_data
    , void (*callback)(void* user_data, std::string response)) noexcept
{
    // 1. setup curl easy handle
    CURL* curl_easy = curl_easy_init();
    assert(curl_easy);
    CURLcode status = curl_easy_setopt(curl_easy, CURLOPT_URL, url.c_str());
    assert(status == CURLE_OK);
    status = curl_easy_setopt(curl_easy, CURLOPT_FOLLOWLOCATION, 1L);
    assert(status == CURLE_OK);

```

```
    // 2. write response data to separate std::string
    std::string* state = new(std::nothrow) std::string{};
    status = curl_easy_setopt(curl_easy
        , CURLOPT_WRITEFUNCTION, CURL_OnWriteCallback);
    assert(status == CURLE_OK);
    status = curl_easy_setopt(curl_easy, CURLOPT_WRITEDATA, state);
    assert(status == CURLE_OK);

    // 3. associate with multi handle/event loop
    CURL_scheduler(curl_async).add_request(curl_easy
        , [state, user_data, callback](CURL* curl_easy)
    {
        long response_code = -1;
        const CURLcode status = curl_easy_getinfo(curl_easy
            , CURLINFO_RESPONSE_CODE, &response_code);
        assert(status == CURLE_OK);
        assert(response_code == 200L);
        curl_easy_cleanup(curl_easy);
        std::string data = std::move(*state);
        delete state;
        callback(user_data, std::move(data));
    });
}
```

There are few moving parts and issues:

1. we create and setup curl easy handle in the same way as for blocking call;
2. we allocate separate `std::string` to write the response data to with the same `CURL_OnWriteCallback` callback as in the blocking implementation;
3. finally, we associate the request with event loop/multi handle
4. new `std::string` will leak the memory if the request is not completed
5. overall, there are more hidden allocations from within `add_request()`:
    a) std::function<> most likely allocates
    b) std::unordered_map allocates

It could be done another way around, eliminating the need for separate `std::string` allocation and few more optimizations, mainly with the help of associating user data with curl easy handle/ CURLOPT_PRIVATE. However, it's good enough for illustrative purposes.

After creation of curl easy handle, we associate it with curl multi handle:

```
CURL_AsyncScheduler::CURL_AsyncScheduler()
{
    const CURLcode status = curl_global_init(CURL_GLOBAL_ALL);
    assert(status == CURLE_OK);
    _multi_curl = curl_multi_init();
```

```
    assert(_multi_curl);
}

CURL_AsyncScheduler::~CURL_AsyncScheduler()
{
    const CURLMcode status = curl_multi_cleanup(_multi_curl);
    assert(status == CURLM_OK);
    curl_global_cleanup();
}

void CURL_AsyncScheduler::add_request(CURL* curl_easy, Callback on_finish)
{
    assert(on_finish);
    assert(curl_easy);
    assert(!_curl_to_callback.contains(curl_easy));

    const CURLMcode status = curl_multi_add_handle(_multi_curl, curl_easy);
    assert(status == CURLM_OK);
    _curl_to_callback[curl_easy] = std::move(on_finish);
}
```

`_curl_to_callback` map is used to be able to retrieve callback later, given curl easy handle (`CURL*`).

Our user-exposed `CURL_async_tick()` API is implemented in terms of scheduler:

```
void CURL_async_tick(CURL_Async curl_async)
{
    CURL_scheduler(curl_async).tick();
}

void CURL_AsyncScheduler::tick()
{
    int running_handles = -1;
    CURLMcode status = curl_multi_perform(_multi_curl, &running_handles);
    assert(status == CURLM_OK);
    int msgs_in_queue = 0;
    while (CURLMsg* m = curl_multi_info_read(_multi_curl, &msgs_in_queue))
    {
        if (m->msg != CURLMSG_DONE)
        {
            continue;
        }
        CURL* curl_easy = m->easy_handle;
        assert(curl_easy);
        status = curl_multi_remove_handle(_multi_curl, curl_easy);
```

```
        assert(status == CURLM_OK);
        auto it = _curl_to_callback.find(curl_easy);
        assert(it != _curl_to_callback.end());
        Callback callback = std::move(it->second);
        assert(callback);
        (void)_curl_to_callback.erase(it);
        callback(curl_easy);
    }
}
```

The main part of event loop is the call to `curl_multi_perform()`. Once done we ask for easy handle requests that were completed, search for an associated callback for each request and invoke it.

Note, there are no threads involved and it's possible to create many GET requests at once with multiple calls to `CURL_async_get()` - libcurl will manage them all together.

Again, it's user responsibility to drive libcurl with a periodic calls to `CURL_async_tick()`. Lets do single request with the API above:

```
#include <print>

int main()
{
    struct State
    {
        std::string response;
        bool done = false;
    };
    CURL_Async curl_async = CURL_async_create();
    State state;
    CURL_async_get(curl_async, "localhost:5001/file1.txt", &state
        , [](void* user_data, std::string response)
    {
        State& state = *static_cast<State*>(user_data);
        state.response = std::move(response);
        state.done = true;
    });
    while (!state.done)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);

    std::println("async response: '{}'", state.response);
}
```

If python HTTP server is running, our program should print:

```
Fox
```

4.4 #on error handling

TBD?.

We assume CURL get, or, more generally, asynchronous operation never fails.

(This includes assumption that NO exception is thrown, including from withing user-defined callback. In general, unless explicitly mentioned, everything is noexcept to avoid dealing with exceptions, error handling).

This simplifies everything, especially the moment when there are several (asynchronous) operations active and all of them could fail or success.

In addition to errors, there is cancellation. Cancellation of asynchronous operation adds another layer of complexity.

Recent std::execution (Senders/Receivers) is explicit about those 3 possibilities and assumes an operation could either success, fail or be cancelled.

Cancellations is (or usually) also asynchronous. You start an operation, you cancel it, but underlying machinery (a) may or may not support cancellation OR (b) it may or may not be too late. On cancel, you may have:

- an operation that actually completed successfully; because it was cancelled too late or cancellation is not supported;
- an operation that errors out because it was cancelled in time, no side effect
- an operation with PARTIAL results already done
- (note: should partial results be considered success or error or discarded?)

This asynchronous nature of cancellation is especially important in the code with callbacks that may arrive after cancellation that assumed to be immediate.

With only two asynchronous operations run concurrently and 3 possible outcomes per operation, there is already 9 states to think about:

1. {success, success}
2. {success, error}
3. {success, cancel}
4. {error, success}
5. {error, error}
6. {error, cancel}
7. {cancel, success}
8. {cancel, error}
9. {cancel, cancel}

Some domains threat cancel as an error. (however, see "Cancellation is not an Error"). That reduces the possible states to only four:

1. {success, success}
2. {success, error}
3. {error, success}
4. {error, error}

STILL, talking about APIs, how do we represent those states?

With just {success, error}, possibilities are:

- assume success always: fine for tools?

```
> std::string CURL_get(...);
```

- implicit, return empty "success" value. Fine for tools? Usually error prone

```
> std::string CURL_get(...);
```

- encode the error with the same type as success. Examples - POSIX, Win32 API

```
> int open(const char* path);
```

- separate status code (with out parameter?)

```
> bool exists(const path& p, std::error_code& ec)
```

- optional: error is signaled, but details are ignored

```
> std::optional<std::string> CURL_get(...);
```

- exceptions

```
> std::string CURL_get(...);
```

- union like/variant/result return

```
> std::expected<std::string, std::error_code> CURL_get(...)
```

Talking about errors and composition, another dimension - usually missed - is error TYPE. How do we compose 2 operations that can fail on its own?

Lets imagine we need to first open a file, then post it to our service:

```
std::expected<File, std::error_code> open_file(path f);
std::expected<void, OnlineError> post(content);
```

What should we return?

```
std::expected<void, ???> process_file(path f)
{
    auto x = open_file(f);
    if (x.has_error())
    {
        return x.error(); // std::error_code
    }
    auto y = post(read_file(x.value()));
```

```
    if (y.has_error())
    {
        return y.error(); // OnlineError
    }
}
```

See, we need to combine std::error_code and OnlineError together. Usually, errors need to be convertible between each other and one type of error is used at the end.

5 #building C++20 coroutines API

sample app.

Coroutines materials:

- How C++ coroutines work.
- All of Asymmetric Transfer, author of cppcoro.

In short, we'd like to be able to write something like this:

```
const std::string response = co_await CURL_await_get(
    curl_async, "localhost:5001/file1.txt");
// use `response` as a usual variable, no callbacks
```

There are several moving and a bit unrelative parts to have working coroutines code. First, coroutine function return type needs to be built, just to be able to write any/empty coroutine:

```
Co_Task coro_work()
{
    co_return;
}
```

Next, there is a need to write coroutine awaitable to be able to `co_await` some work, specifically, GET request:

```
Co_Task coro_work(CURL_Async curl_async)
{
    std::string response = co_await CURL_await_get(curl_async
        , "localhost:5001/file1.txt");
    co_return;
}
```

And, finally, there are some challenges to have a code that has several GET requests on the fly, with coroutines.

Lets start with basics.

5.1 #C++ coroutines, basic task type

source code.

There is a trick to writing some basic C++20 coroutines code - **listen to the compiler**. Lets see what it takes to make the next code "work":

```
Co_Task coro_work()
{
    co_return;
}
```

`Co_Task` is a class, lets have empty one and try to compile:

```
struct Co_Task {};

Co_Task coro_work()
{
    co_return;
}
```

MSVC complains:

```
main.cc(164,5): error C3774: cannot find 'std::coroutine_traits':
                Please include <coroutine> header
```

after including `<coroutine>` header:

```
main.cc(166,5): error C2039: 'promise_type': is not a member of
                'std::coroutine_traits<Co_Task>'
```

Lets add empty `promise_type` class inside `Co_Task`:

```
#include <coroutine>

struct Co_Task
{
    struct promise_type {};
};

Co_Task coro_work()
{
    co_return;
}
```

MSVC complains:

```
main.cc(170,1): error C3789: this function cannot be a coroutine:
                'Co_Task::promise_type' does not declare the member
                'get_return_object()'
main.cc(170,1): error C3789: this function cannot be a coroutine:
```

```
                'Co_Task::promise_type' does not declare the member
                'initial_suspend()'
main.cc(170,1): error C3789: this function cannot be a coroutine:
                'Co_Task::promise_type' does not declare the member
                'final_suspend()'
```

Ah, so `promise_type` should have `get_return_object()`, `initial_suspend()` and `final_suspend()` member functions. Return types are unclear, unfortunately. To speed-up things, we know that `get_return_object()` should return `Co_Task`. For `initial_suspend()` and `final_suspend()` we'll go with `std::suspend_always` awaitables for now. That gives:

```
#include <coroutine>

struct Co_Task
{
    struct promise_type
    {
        Co_Task get_return_object()           { return {}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend()   { return {}; }
    };
};

Co_Task coro_work()
{
    co_return;
}
```

MSVC complains:

```
main.cc(164,12): error C3781: Co_Task::promise_type: a coroutine's
                 promise must declare either
                 'return_value' or 'return_void'
main.cc(176,1):  error C2039: 'unhandled_exception': is not a member
                 of 'Co_Task::promise_type'
```

Since our `coro_work()` coroutine has just `co_return`, we should provide `return_void()` member function. With `unhandled_exception()`, we have:

```
struct promise_type
{
    Co_Task get_return_object()           { return {}; }
    std::suspend_always initial_suspend() { return {}; }
    std::suspend_always final_suspend()   { return {}; }
    void return_void()                    {}
    void unhandled_exception()            {}
```

```
};
```

MSVC complains:

```
main.cc(168,29): error C5231: the expression
                 'co_await promise.final_suspend()' must be non-throwing
```

OK, makes sense. Finally,

```
#include <coroutine>

struct Co_Task
{
    struct promise_type
    {
        Co_Task get_return_object()                  { return {}; }
        std::suspend_always initial_suspend()        { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void()                           {}
        void unhandled_exception()                   {}
    };
};

Co_Task coro_work()
{
    co_return;
}
```

compiles! We just need to fill in details and implement given functions properly.

There are way too many different ways to implement coroutine task/promise types. There are no constraints and, in general, it all depends on your design and needs. We'll go with an owning coroutine task type:

1. `Co_Task` will own coroutine handle (as in free coroutine in the destructor).
2. Because of the above, `final_suspend()` must suspend always.
3. Co_Task will be a "lazy" coroutine, meaning, it's going to be suspended after initial call of `coro_work()`/coroutine function.
4. Because of the above, `initial_suspend()` must suspend.
5. Because coroutine is suspended initially, `Co_Task` needs to expose `resume()` or similar function to run a coroutine.

For now, lets proceed with implementation. Since we own coroutine, our `Co_Task` needs to have destructor, should be move-only:

```
struct Co_Task
{
    struct promise_type;
    using co_handle = std::coroutine_handle<promise_type>;
```

```
5
6       struct promise_type
7       {
8           Co_Task get_return_object()
9           {
10              return Co_Task{co_handle::from_promise(*this)};
11          }
12          // ...
13      };
14
15      Co_Task(co_handle coro)
16          : _coro{coro} {}
17      Co_Task(Co_Task&& rhs) noexcept
18          : _coro{std::exchange(rhs._coro, {})} { }
19      Co_Task(const Co_Task&) = delete;
20      ~Co_Task() noexcept
21      {
22          if (_coro)
23          {
24              _coro.destroy();
25          }
26      }
27
28      co_handle _coro;
29  };
```

In short, when we call `coro_work()`, compiler creates `Co_Task::promise_type` and invokes `get_return_object()` to be able to return an instance of `Co_Task` to the user. Here, in `get_return_object()` there is a way to get an access to `std::coroutine_handle<>` - the only way to interact with just allocated coroutine. Once `Co_Task` is created, we return it to the user. It's **up to the user** to manage `std::coroutine_handle<>`. In our case, we own just created coroutine, hence if `Co_Task` is destroyed, we assume coroutine is in suspended state and destroy it too.

Writing down the rest of functions:

```
1   std::suspend_always promise_type::initial_suspend()
2   {
3       return {};
4   }
5
6   std::suspend_always promise_type::final_suspend() noexcept
7   {
8       return {};
9   }
10
```

```
11  void promise_type::return_void()
12  {
13      // yeah, we return void. Nothing to do
14  }
15
16  void promise_type::unhandled_exception()
17  {
18      // crash, no exceptions handling
19      assert(false);
20  }
```

we can test the basics:

```
1   Co_Task coro_work()
2   {
3       std::println("inside coro_work");
4       co_return;
5   }
6
7   int main()
8   {
9       Co_Task coro = coro_work();
10  }
```

which runs and... prints nothing since our coroutine is created and immediately suspended even before executing first print.

Lets expose `resume()` for our `Co_Task` and use it:

```
1   void Co_Task::resume()
2   {
3       assert(_coro);
4       assert(!_coro.done());
5       _coro.resume();
6   }
7
8   Co_Task coro_work()
9   {
10      std::println("inside coro_work");
11      co_return;
12  }
13
14  int main()
15  {
16      std::println("-- before coro_work()");
17      Co_Task coro = coro_work();
18      std::println("-- after coro_work()");
```

```
    coro.resume();
    std::println("-- after resume()");
}
```

which prints:

```
-- before coro_work()
-- after coro_work()
inside coro_work
-- after resume()
```

5.2 #C++ coroutines, basic await

source code.

Given that we can have simplest coroutine, what does it take to co_await? Lets try to compile:

```
struct Co_CurlAsync {};

Co_Task coro_work()
{
    co_await Co_CurlAsync{};
    co_return;
}
```

MSVC complains:

```
main.cc(73,26): error C2039: 'await_ready': is not a member of 'Co_CurlAsync'
main.cc(73,26): error C2039: 'await_suspend': is not a member of 'Co_CurlAsync'
main.cc(70,26): error C2039: 'await_resume': is not a member of 'Co_CurlAsync'
```

So `co_await` requires "awaiter" to have those 3 functions. We can think about awaiter as something that:

1. knows if some operation is ready or not (`..._ready`)
2. knows how to resume coroutine later (`..._suspend`)
3. knows how to get the result of awaited operation (`..._resume`)

The compiler asks awaiter, specifically, `Co_CurlAsync` with `bool await_ready()` if operation is done/ready or is in progress. If awaiter returns false, the compiler switches current coroutine state to "suspended" and invokes awaiter's `await_suspend(std::coroutine_handle<> coro)` customization point which allows to remember currently suspended coroutine `coro` handle, to call `.resume()` later, once operation is done. Once coroutine is resumed, compiler asks for a value from last awaiter responsible for suspend.

In short, we can have `Co_CurlAsync` awaiter that tells that (1) operation is not ready yet (2) on suspend, resumes coroutine immediately and (3) returns nothing:

```cpp
struct Co_CurlAsync
{
    bool await_ready()
    {
        return false;
    }

    void await_suspend(std::coroutine_handle<> coro)
    {
        std::println("-- inside suspend, resuming immediately");
        coro.resume();
    }

    void await_resume()
    {
        std::println("-- resume");
    }
};

Co_Task coro_work()
{
    std::println("before co_await");
    co_await Co_CurlAsync{};
    std::println("after co_await");
    co_return;
}

int main()
{
    Co_Task coro = coro_work();
    coro.resume();
}
```

which prints:

```
before co_await
-- inside suspend, resuming immediately
-- resume
after co_await
```

Now, on suspend, we did nothing, but immediately resumed coroutine. But we also could start an asynchronous operation and, on finish, resume the coroutine.

5.3 #C++ coroutines, await callback with a crash

source code.

Lets continue implementing `Co_CurlAsync` above, in short:

```
struct Co_CurlAsync
{
    CURL_Async _curl_async{};
    std::string _url;
    std::coroutine_handle<> _coro;
    std::string _response;

    bool await_ready()
    { // 1. CURL_async_get() is not yet started, force coroutine suspend:
        return false;
    }

    void await_suspend(std::coroutine_handle<> coro)
    { // 2. remember coroutine handle, start request, resume on finish:
        _coro = coro;

        CURL_async_get(_curl_async, _url, this
            , [](void* user_data, std::string response)
        {
            Co_CurlAsync& self = *static_cast<Co_CurlAsync*>(user_data);
            self._response = std::move(response);
            self._coro.resume();
        });
    }

    std::string await_resume()
    { // 3. after resume, return response:
        return std::move(_response);
    }
};

Co_CurlAsync CURL_await_get(CURL_Async curl_async, const std::string& url)
{
    return Co_CurlAsync{._curl_async = curl_async, ._url = url};
}
```

So, now `co_await CURL_await_get(..., "url")` should compile and kind-a work. As always, there are few moving part.

When coroutine function (represented as `std::coroutine_handle<>`) `co_await`s our CURL awaiter - Co_CurlAsync, we:

1. force whole coroutine to suspend, since we return false from `await_ready()`
2. this is needed so compiler invokes `await_suspend()` and gives us a handle to currently awaiting coroutine, so we can (a) start request and (b) resume

coroutine with a call to `coro.resume()`
3. finally, once request is complete, we can return the `_response` from `await_resume()`

There is one big issue there: what if we start a request with `CURL_async_get()`, coroutine suspends, BUT user discards `Co_Task` value that destroys coroutine, making `std::coroutine_handle<>` we remembered - dangling? There are several possible solutions, but lets see the current code in action by writing our main() function:

```
Co_Task coro_main(CURL_Async curl_async)
{
    const std::string response = co_await CURL_await_get(
        curl_async, "localhost:5001/file1.txt");

    std::println("{}", response);
    co_return;
}

int main()
{
    CURL_Async curl_async = CURL_async_create();
    Co_Task task = coro_main(curl_async);
    task.resume();
    while (task.is_in_progress())
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

Here, we setup `CURL_Async`, as usual, and drive the loop until coroutine is in progress:

```
bool Co_Task::is_in_progress() const
{
    assert(_coro);
    return !_coro.done();
}
```

Running the sample should print:

```
Fox
```

However, that works because we wait for coroutine until full complete. If we discard Co_Task too early, there is going to be a crash:

```
int main()
{
```

```
3       CURL_Async curl_async = CURL_async_create();
4
5       {
6           Co_Task task = coro_main(curl_async);
7           task.resume(); // run
8       }   // **destroy**
9
10      while (true)
11      {
12          CURL_async_tick(curl_async); // resume coroutine from there
13      }
14      CURL_async_destroy(curl_async);
15  }
```

It happens because `CURL_async_get()` callback remembers 2 pointers:

1. `this` pointer to Co_CurlAsync/awaiter which is owned by coroutine frame
2. and `coroutine_handle<>` itself, which we destroy BEFORE `CURL_async_get()` finish.

In short, we start request, then `.destroy()` coroutine, then try to resume dangling coroutine inside a callback with a call to `.resume()` even using stale pointer to awaiter (user data in the callback).

There are several solutions, few of them:

1. Don't own and don't destroy coroutine inside Co_Task destructor (.. in a multiple ways).
2. Delay coroutine destroy if there are live references to it.
3. Be able to cancel `CURL_async_get()` request if coroutine/awaiter is destroyed.
4. Ensure that callback has a safe way to detect dead coroutine and do nothing.

1st solution could be the best but changes completely the semantics of `Co_Task`, does not allow to easily have `Co_Task<T>` that return some value and requires to be able to change `Co_Task` internals.

2nd solution is similar in the sense that it also requires `Co_Task` changes and the code around.

3rd solution requires changes to our basic C-style callback API which we assume we can't do (since, otherwise, the interface is more advanced).

4th solution is the most inefficient and requires no changes neither in Co_Task nor in callback API.

5.4 #C++ coroutines, await callback

source code.

Lets fix the problematic part in a simple way:

```
// struct Co_CurlAsync ...
void await_suspend(std::coroutine_handle<> coro)
{ // 2. remember coroutine handle, start request, resume on finish:
    _coro = coro;

    CURL_async_get(_curl_async, _url
        , this // ** HERE
        , [](void* user_data, std::string response)
    {
        Co_CurlAsync& self = *static_cast<Co_CurlAsync*>(user_data);
        self._response = std::move(response);
        self._coro.resume();
    });
}
```

For a "happy" path, when `CURL_async_get()` completes before coroutine destruction, the flow is:

1) create Co_CurlAsync, invoke CURL_async_get
2) invoke callback (access this/coroutine)
3) destroy Co_CurlAsync

For a "bad" path, when coroutine is destroyed while we have CURL_async_get in progress, the flow is:

1) create Co_CurlAsync, invoke CURL_async_get
2) destroy Co_CurlAsync
3) invoke callback (access this/dead coroutine)

Lets allocate a separate object that can outlive the coroutine/await. There is an assumption that `CURL_async_get()` callback is always going to be invoked. Given this, we:

A) allocate `WaitState` object - right before CURL_async_get()
B) pass it to the callback instead of `this`
C) destroy allocated object inside callback

This way we guarantee that the object is always alive while request is in progress and its lifetime is bound the the request itself and nothing else:

```
_wait_state = new(std::nothrow) WaitState{._self = this};
assert(_wait_state);

CURL_async_get(_curl_async, _url
    , _wait_state
    , [](void* user_data, std::string response)
{
    WaitState* wait_state = static_cast<WaitState*>(user_data);
```

```
    assert(wait_state);
    if (wait_state->_self)
    {
        Co_CurlAsync& self = *wait_state->_self;
        self._wait_state = nullptr;
        self._response = std::move(response);
        self._coro.resume();
    }
    // else: Co_CurlAsync/coroutine is dead
    delete wait_state;
});
```

`WaitState` is just a struct that has a reference to `Co_CurlAsync`:

```
struct Co_CurlAsync
{
    struct WaitState
    {
        Co_CurlAsync* _self = nullptr;
    };
    WaitState* _wait_state = nullptr;
```

When coroutine/Co_CurlAsync is destroyed, we need to mark `WaitState` reference to it as null so `CURL_async_get()` knows it's not alive:

```
~Co_CurlAsync()
{
    if (_wait_state)
    { // CURL_async_get() is still in progress
        assert(_wait_state->_self == this);
        _wait_state->_self = nullptr; // dead
    }
    // else: CURL_async_get() is already completed
}
```

That's how you write inefficient coroutine types for a systems that know nothing about coroutines. When doing simple call:

```
const std::string response = co_await CURL_await_get(
    curl_async, "localhost:5001/file1.txt");
```

we:

1. allocate coroutine frame itself
2. CURL_await_get allocates `WaitState`
3. CURL_async_get allocates `std::string` to write a response
4. CURL_async_get allocates `std::function` for a generic callback
5. CURL_async_get allocates `std::unordered_map` node to remember what

to call when
6. .. and probably something else (CURL internals, etc)

"simple" CURL_async_get() implementation alone brings 3 allocations. "simple" co_await CURL_async_get() brings 2 more separate allocations.

With CURL scheduler that **knows** about coroutines and few more optimizations and limitations, this number of allocations can go down to amortized 0:

- CURL scheduler preallocates up to N max requests
- request itself knows how to store the response and coroutine-callback inline
- coroutine itself re-uses memory pool for up to N max active coroutines.

Anyway, we now can co-await requests with a nice syntax:

```
static Co_Task coro_main(CURL_Async curl_async)
{
    const std::string r1 = co_await CURL_await_get(
        curl_async, "localhost:5001/file1.txt");
    const std::string r2 = co_await CURL_await_get(
        curl_async, "localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
    co_return;
}

int main()
{
    CURL_Async curl_async = CURL_async_create();
    Co_Task task = coro_main(curl_async);
    task.resume();
    while (task.is_in_progress())
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

Note, however, we run 2 requests one after another: second request starts when first co_await ends. There is no concurrency..

5.5 #coroutine task that holds a return value

source code.

To be more useful, we'd like to be able return something out of coroutine:

```
static Co_Task<int> coro_main()
{
```

```
    co_return 3;
}

int main()
{
    Co_Task<int> task = coro_main();
    task.resume();
    std::println("{}", task.get_once());
}
```

It requires implementing promise_type `return_value()` and `return_void()` customization points. The only complication is that we need to handle `Co_Task<void>`. To do so, lets start with `promise_return<T>`:

```
template<typename T>
struct promise_return
{
    std::variant<std::monostate, T> _value;
    template<typename U>
    void return_value(U&& v) noexcept
    {
        _value.template emplace<1>(std::forward<U>(v));
    }
    const T& get() const noexcept
    {
        assert(_value.index() == 1);
        return std::get<1>(_value);
    }
    T&& get_once() noexcept
    {
        const T& v = static_cast<const promise_return&>(*this).get();
        return std::move(const_cast<T&>(v));
    }
};

template<>
struct promise_return<void>
{
    void return_void() noexcept
    {
    }
    void get() const noexcept
    {
    }
    void get_once() noexcept
    {
```

```
    }
};

template<typename T>
struct promise_return<T&>
{
    static_assert(sizeof(T) == 0, "we do not support Co_Task<T&>");
};
```

promise_return for void injects `return_void()`, get() returns nothing. promise_return for any type T injects `return_value()` and remembers the value. std::variant is used to handle non-default-constructible types and, in general, construct a value only at the point of the actual return/co_return.

With the help of promise_return, promise_type remains the same as our initial version:

```
template<typename T>
struct Co_Task
{
    struct promise_type;
    using co_handle = std::coroutine_handle<promise_type>;

    struct promise_type : promise_return<T>
    {
        Co_Task get_return_object() noexcept
        {
            return Co_Task{co_handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() noexcept
        {
            return {};
        }
        std::suspend_always final_suspend() noexcept
        {
            return {};
        }
        void unhandled_exception() noexcept
        {
            // crash, no exceptions handling
            assert(false);
        }
    };

    // ...
    decltype(auto) get() const
```

```
    {
        assert(_coro);
        return _coro.promise().get();
    }
    decltype(auto) get_once() const
    {
        assert(_coro);
        return _coro.promise().get_once();
    }

    co_handle _coro;
};
```

Now `Co_Task<T>` works:

```
static Co_Task<int> coro_main()
{
    co_return 3;
}
```

5.6 #awaiting coroutine task

source code.

Now, given `Co_Task<T>`, we need to await for it somehow:

```
static Co_Task<int> coro_get()
{
    co_return 3;
}

static Co_Task<int> coro_main()
{
    const int v = co_await coro_get();
    co_return v;
}
```

There are 2 important implementation bits to know:

- Symmetric Transfer
- final_suspend() can return **custom** awaitable

`final_suspend()` allows to inject custom code for the moment of coroutine END, so we can do something at that point in time:

```
struct promise_type : promise_return<T>
{
    auto final_suspend() noexcept
    {
```

```
        struct Final_Await : std::suspend_always
        {
            void await_suspend(co_handle self_coro) noexcept
            {
                // **HERE**
            }
        };
        return Final_Await{};
    }
```

Given a simple coroutine:

```
static Co_Task<int> coro_get()
{
    co_return 3;
} // <-- final_suspend()
```

our Final_Await, await_suspend() gets invoked after co_return, passing a coroutine that is about to end. This is exactly what we need to notify some other waiting code that we are done. We could invoke a callback... BUT with a symmetric transfer, `await_suspend()` can return a **next** coroutine that must be executed:

```
struct promise_type : promise_return<T>
{
    std::coroutine_handle<> _waiting_coro = std::noop_coroutine();

    auto final_suspend() noexcept
    {
        struct Final_Await : std::suspend_always
        {
            std::coroutine_handle<> await_suspend(co_handle self_coro) noexcept
            {
                return self_coro.promise()._waiting_coro;
            }
        };
        return Final_Await{};
    }
```

Here, if no one waits for us, we return `std::noop_coroutine()` that does nothing. Otherwise, if `_waiting_coro` was set, we end the execution of our coroutine and resume any other coroutine that was waiting for us. To setup `_waiting_coro`, we implement awaitable interface for Co_Task itself:

```
template<typename T>
struct Co_Task
{
    bool await_ready()
```

```
    {
        assert(is_in_progress());
        return false;
    }
    // intentionally auto, not decltype(auto)
    auto await_resume()
    {
        return get_once();
    }
    std::coroutine_handle<> await_suspend(std::coroutine_handle<> waiting_coro)
    {
        _coro.promise()._waiting_coro = waiting_coro;
        return _coro; // resume us
    }

    co_handle _coro;
};
```

It's important to understand what is `_coro` and what is `waiting_coro`. We have 3 moving parts (1) Final_Await::await_suspend() with self_coro, (2) Co_Task::await_suspend() waiting_coro and (3) Co_Task `_coro` member itself:

```
struct Co_Task
{
    [1] ... promise_type::Final_Await::await_suspend(co_handle self_coro) noexcept
    {
        return self_coro.promise()._waiting_coro;
    }
    [2] ... await_suspend(std::coroutine_handle<> waiting_coro)
    {
        _coro.promise()._waiting_coro = waiting_coro;
        return _coro; // resume us
    }
    [3] co_handle _coro;
};
```

Given our awaiting code sample:

```
static Co_Task<int> coro_get()
{
    co_return 3;
}

static Co_Task<int> coro_main()
{
    co_return co_await coro_get();
}
```

We have 2 coroutines created, where:

1. `_coro` from part [3] is `coro_get()`
2. `waiting_coro` from part [2] await_suspend() is our `coro_main()`; and
3. `self_coro` from part [1] is `coro_get()` again that gets finished.

So, we

1. create coro_main()
2. create coro_get()
3. then await_suspend() on coro_get() from within coro_main()
4. that remembers that `_waiting_coro` is coro_main()
5. resumes coro_get() execution (by returning it, symmetric transfer); and then
6. we finish coro_get() and its final_suspend() resumes `_waiting_coro` which is coro_main().

That way coro_get() was scheduled to be executed because we co_await it from within coro_main() and coro_main() was executed 2nd time because coro_get() was completed.

One note on the co_await semantic and ownership: see how for await_resume() we use `get_once()` to move the value out awaited task. In general, when co_awaiting, we consume the task and it can't be used after co_await. So the next code should be disallowed:

```
static Co_Task<int> coro_main()
{
    Co_Task<int> t = coro_get(1);
    co_await t; // t is lvalue
    t.get(); // error
}
```

To enforce that, we use `operator co_await() &&` with `&&` ref-qualifier; see also C++ Coroutines: Understanding operator co_await:

```
struct Co_Await
{
    Co_Task<T> _task;

    bool await_ready()
    {
        assert(_task.is_in_progress());
        return false;
    }
    // intentionally auto, not decltype(auto)
    auto await_resume()
    {
        return _task.get_once();
```

```
    }
    std::coroutine_handle<> await_suspend(std::coroutine_handle<> waiting_coro)
    {
        _task._coro.promise()._waiting_coro = waiting_coro;
        return _task._coro; // resume us
    }
};

Co_Await operator co_await() &&
{
    // consume this.
    return Co_Await{._task{std::move(*this)}};
}
```

Now we can co_await our tasks multiple times:

```
static Co_Task<int> coro_get(int v)
{
    co_return v;
}

static Co_Task<int> coro_main()
{
    const int v1 = co_await coro_get(2);
    const int v2 = co_await coro_get(3);
    co_return (v1 + v2);
}

int main()
{
    Co_Task<int> task = coro_main();
    task.resume();
    std::println("{}", task.get_once());
}
```

Note, how we still execute `coro_get(2)` first then `coro_get(3)` second. There is still no way to launch those 2 coroutines concurrently and wait for both of them:

```
static Co_Task<int> coro_main()
{
    auto [v1, v2] = co_await CO_await_all(coro_get(2), coro_get(3));
    co_return (v1 + v2);
}
```

5.7 #waiting for multiple coroutines

source code.

Lets implement awaiting for multiple Co_Tasks:

```
static Co_Task<int> coro_main()
{
    auto [v1, v2] = co_await CO_await_all(coro_get(2), coro_get(3));
    co_return (v1 + v2);
}
```

Note how we return a tuple of all awaited results (since our coroutine task can never fail). Also, note how this is different to sequential co_await since all children coroutine tasks are started at the point of await and executed concurrently:

```
static Co_Task<void> coro_main()
{
    int v1 = co_await coro_get(2);
    int v2 = co_await coro_get(3); // runs strictly AFTER first task
}
```

To await N tasks, our parent coroutine/task needs to be resumed only when last of that N tasks completes. To achieve this, we introduce a counter that signals how many tasks are still in progress. So our Final_Await can resume the parent only when the wait count is zero. See the old implementation:

```
struct promise_type : promise_return<T>
{
    std::coroutine_handle<> _waiting_coro = std::noop_coroutine();

    auto final_suspend() noexcept
    {
        struct Final_Await : std::suspend_always
        {
            std::coroutine_handle<> await_suspend(co_handle self_coro) noexcept
            {
                return self_coro.promise()._waiting_coro;
            }
        };
        return Final_Await{};
    }
```

and compare to a new one:

```
struct promise_type : promise_return<T>
{
    std::int32_t* _wait_count = nullptr;
    co_handle _waiting_coro;
```

```
5
6       std::coroutine_handle<> Final_Await::await_suspend(co_handle self_coro) noexcept
7       {
8           promise_type& self = self_coro.promise();
9           if (self._waiting_coro)
10          {
11              assert(self._wait_count);
12              std::int32_t& wait_count = *self._wait_count;
13              wait_count -= 1;
14              assert(wait_count >= 0);
15              if (wait_count == 0)
16              {
17                  return self._waiting_coro;
18              }
19              // else: we are not the last task, do nothing.
20          }
21          // else: no one was awaiting us.
22          return std::noop_coroutine();
23      }
```

When Co_Task ends, we:

1. check if someone waits for us
2. decrement wait counter by 1
3. see if we are the last one and resume awaiting coroutine

With this change, awaiting a single Co_Task requires setting the counter:

```
struct Co_Await
{
    Co_Task<T> _task;
    std::int32_t _wait_count = 1;
    std::coroutine_handle<> Co_Await::await_suspend(
        std::coroutine_handle<> waiting_coro) noexcept
    {
        promise_type& promise = _task._coro.promise();
        promise._wait_count = &_wait_count; // 1
        promise._waiting_coro = waiting_coro;
        return _task._coro; // resume us
    }
};
```

For awaiting of N coroutines/tasks, we start with `CO_await_all()`:

```
template<typename... Ts>
static auto CO_await_all(Co_Task<Ts>&&... tasks)
{
    static_assert(sizeof...(Ts) > 0);
```

```
    using Is = std::index_sequence_for<Ts...>;
    return Co_Await_All<Is, Ts...>{._tasks{std::move(tasks)...}};
}
```

It accepts variadic number of any tasks, consumes all of them and returns an awaiter - Co_Await_All. We store all of the Co_Tasks and when wait completes, return a tuple of all of the results:

```
template<auto... Is, typename... Ts>
struct Co_Await_All<std::index_sequence<Is...>, Ts...>
{
    std::tuple<Co_Task<Ts>...> _tasks;
    std::int32_t _wait_count = sizeof...(Ts);
    bool await_ready()
    {
        // assert(_tasks[I].is_in_progress()...);
        return false;
    }
    auto await_resume()
    {
        return std::tuple<Ts...>{std::get<Is>(_tasks).get_once()...};
    }
};
```

When awaiting starts, we (a) setup wait_count=N and (b) resume all of the tasks, also linking awaiting coroutine - as in the regular single-task await:

```
void Co_Await_All::await_suspend(std::coroutine_handle<> waiting_coro)
{
    auto handle = [&](auto& Task)
    {
        auto& promise = Task._coro.promise();
        promise._wait_count = &_wait_count;
        promise._waiting_coro = waiting_coro;
        Task._coro.resume();
    };

    (handle(std::get<Is>(_tasks)), ...);
}
```

Finally, we can await several concurrently running Co_Tasks:

```
static Co_Task<int> coro_get(int v)
{
    co_return v;
}

static Co_Task<int> coro_main()
```

```
{
    auto [v1, v2] = co_await CO_await_all(coro_get(2), coro_get(3));
    co_return (v1 + v2);
}

int main()
{
    Co_Task<int> task = coro_main();
    task.resume();
    std::println("{}", task.get_once());
}
```

5.8 #waiting for multiple CURL requests with tasks

source code.

We already built CURL_await_get() that awaits CURL request within a coroutine. With CO_await_all() API, we can await for multiple concurrent CURL requests by wrapping every request into a separate coroutine task:

```
Co_Task<std::string> CURL_coro_get(CURL_Async curl_async, std::string url)
{
    co_return co_await CURL_await_get(curl_async, url);
}

Co_Task<void> coro_main(CURL_Async curl_async)
{
    auto [r1, r2] = co_await CO_await_all(
        CURL_coro_get(curl_async, "localhost:5001/file1.txt"),
        CURL_coro_get(curl_async, "localhost:5001/file2.txt"),
        );
    std::println("{}", r1);
    std::println("{}", r2);
}
```

That's.. all. Note, however, 3 separate coroutines are allocated (one for coro_main and two for CURL_coro_get). We can do less generic coroutine awaiter that skips intermediate coroutines.

5.9 #waiting for multiple CURL request with custom awaitable

source code.

Instead of CURL_coro_get() + CO_await_all():

```
Co_Task<std::string> CURL_coro_get(CURL_Async curl_async, std::string url)
{
    co_return co_await CURL_await_get(curl_async, url);
}
// ...
auto [r1, r2] = co_await CO_await_all(
    CURL_coro_get(curl_async, "localhost:5001/file1.txt"),
    CURL_coro_get(curl_async, "localhost:5001/file2.txt"),
    );
```

we can make a separate, specialized awaiter, like CURL_await_get_many():

```
auto [r1, r2] = co_await CURL_await_get_many(curl_async,
    "localhost:5001/file1.txt",
    "localhost:5001/file2.txt"
    );
```

that skips intermediate CURL_coro_get() coroutine.

CURL_await_get_many() is almost the same as CURL_await_get() implementation:

```
template<typename... URLs>
    requires std::conjunction_v<
        std::is_constructible<std::string, URLs&&>...>
auto CURL_await_get_many(CURL_Async curl_async, URLs... urls)
{
    Co_CurlAsyncMany<std::index_sequence_for<URLs...>> awaiter;
    awaiter._curl_async = curl_async;
    awaiter.setup_urls(std::move(urls)...);
    return awaiter;
}
```

Ignoring templates (to be able to accept variadic number of URLs), we create Co_CurlAsyncMany awaitable that knows how to wait for N requests:

```
template<typename Is>
struct Co_CurlAsyncMany;

template<auto... Is>
struct Co_CurlAsyncMany<std::index_sequence<Is...>>
{
    struct WaitState
    {
        std::int32_t _active_count = 0;
        Co_CurlAsyncMany* _self = nullptr;
    };

```

```
    static constexpr std::size_t Count = sizeof...(Is);
    CURL_Async _curl_async{};
    std::coroutine_handle<> _coro;

    WaitState* _wait_state = nullptr;
    std::array<std::string, Count> _urls;
    std::array<std::string, Count> _responses;

    template<typename... URLs>
    void setup_urls(URLs... urls)
    {
        ((_urls[Is] = std::move(urls)), ...);
    }

    // ...
};
```

Here, we remember all of URLs and have an `std::array<std::string, Count>` for all of responses. Our WaitState passed to CURL_async_get() callback also has a counter for active requests since we want to complete only when last request is finished:

```
bool Co_CurlAsyncMany::await_ready() noexcept
{
    return false;
}

void Co_CurlAsyncMany::await_suspend(std::coroutine_handle<> coro) noexcept
{
    _coro = coro;
    _wait_state = new(std::nothrow) WaitState;
    assert(_wait_state);
    _wait_state->_active_count = Count;
    _wait_state->_self = this;

    auto handle = [&]<auto I>(std::integral_constant<std::size_t, I>)
    {
        CURL_async_get(_curl_async, _urls[I]
            , _wait_state
            , [](void* user_data, std::string response)
        {
            WaitState* wait_state = static_cast<WaitState*>(user_data);
            assert(wait_state);
            if (on_response<I>(*wait_state, std::move(response)))
            {
                delete wait_state;
```

```
            }
        });
    };

    (handle(std::integral_constant<std::size_t, Is>{}), ...);
}

Co_CurlAsyncMany::~Co_CurlAsyncMany() noexcept
{
    if (_wait_state)
    {
        assert(_wait_state->_self == this);
        _wait_state->_self = nullptr; // dead
    }
}

std::array<std::string, Count> Co_CurlAsyncMany::await_resume() noexcept
{
    return std::move(_responses);
}
```

Basically, we:

1. start N requests; and
2. clean-up `wait_state` only for a last response
3. wait_state logic/lifetime handling is the same as is for Co_CurlAsync

Finally, `on_response<I>()` is also largely the same as Co_CurlAsync: (a) we decrement active requests count, (b) see if coroutine is still alive, (c) write a response to proper place and (c) resume awaiting coroutine if we are the last response:

```
template<auto I>
static bool Co_CurlAsyncMany::on_response(
    WaitState& wait_state, std::string&& response)
{
    wait_state._active_count -= 1;
    assert(wait_state._active_count >= 0);

    if (wait_state._self)
    {
        Co_CurlAsyncMany& self = *wait_state._self;
        self._responses[I] = std::move(response);
        if (wait_state._active_count == 0)
        {
            self._wait_state = nullptr;
            self._coro.resume();
```

```
        }
    }
    // else: Co_CurlAsyncMany/coroutine is dead

    if (wait_state._active_count == 0)
    {
        return true; // done
    }
    return false;
}
```

With that in mind, we can do:

```
Co_Task<void> coro_main(CURL_Async curl_async)
{
    auto [r1, r2] = co_await CURL_await_get_many(curl_async,
        "localhost:5001/file1.txt",
        "localhost:5001/file2.txt"
        );
    std::println("{}", r1);
    std::println("{}", r2);
}
```

which is slightly different to a more generic version:

```
Co_Task<void> coro_main(CURL_Async curl_async)
{
    auto [r1, r2] = co_await CO_await_all(
        CURL_coro_get(curl_async, "localhost:5001/file1.txt"),
        CURL_coro_get(curl_async, "localhost:5001/file2.txt"),
        );
    std::println("{}", r1);
    std::println("{}", r2);
}
```

For the rest of the sample code, we'll go using `CO_await_all()` version.

6 #building Fibers API

source code, sample app.

(Win32) Fibers materials:

- Using Fibers
- Fibers: the Most Elegant Windows API

In short, we'd like to be able to write something like this:

```
const std::string response = FF_await_get(
    curl_async, "localhost:5001/file1.txt");
// use `response` as a usual variable, no callbacks
```

to make an asynchronous request. No callbacks, no special keywords.

While "Using Fibers" example above is nice, it's still overly complicated to get basic idea in a simpler form.

Going with Win32 Fibers, short intro is:

- fibers allow to suspend and resume execution at any given point inside a function
- they are stackful coroutines, as opposed to C++20 coroutines that are stackless
- they implement symmetric coroutines (same as C++20 coroutines); we'll build asymmetric coroutines on top of Fibers
- it should be trivial to ifdef POSIX implementation using deprecated `ucontext.h`

and the general idea is:

- you create a fiber with `::CreateFiber()` API; it's suspended
- you switch to/activate/run a fiber with `::SwitchToFiber()` call
- you can switch only between fibers; so everything must be a fiber

Last point is more specific to Win32 API:

- from within a `main()` entry point we are in a thread context
- once `::CreateFiber()` gets you a fiber to switch to, (main) thread needs to be converted to a fiber with a call to `::ConvertThreadToFiber()`

But, ignoring thread-to-fiber conversion, we just (a) create N fibers and (b) switch an execution between them. Worth mentioning: fibers still execute withing a thread context, meaning - context switches (between threads) still happen while fiber is executing.

Below, we go with a `Fiber` class that encapsulates all the system APIs above. We'd like to have a working code that may look like this:

```
void Fiber::run()
{
    std::println("fiber1");
    suspend();
    std::println("fiber2");
}

int main()
{
    Fiber fiber;
```

```
    std::println("main1");
    fiber.resume();
    std::println("main2");
    fiber.resume();
    std::println("main3");
}
```

and prints:

```
main1
fiber1
main2
fiber2
main3
```

6.1 #basic Fiber

source code.

We start with a `Fiber` class that allocates a fiber:

```
struct Fiber
{
    void* _fiber = nullptr;
    void* _parent_fiber = nullptr;

    Fiber(const Fiber&) = delete;
    Fiber()
    {
        _fiber = ::CreateFiber(
            0 // default stack size
            , &FiberProc
            , this); // parameter to FiberProc
        assert(_fiber);
    }
    ~Fiber()
    {
        ::DeleteFiber(_fiber);
        _fiber = nullptr;
    }
```

To make our lives easier, there are few assumptions and simplifications:

- we are on x64 system, so there is no need to use extended Fibers API
- we assume `LPVOID` is `void*` so no Win32 API types are used (great for headers)
- and, given x64, `WINAPI`/`__stdcall` can be omitted, so callbacks passed to Win32 API can have simple C++ declarations

That gives us next include:

```
#include <Windows.h>
// Note: the floating-point state on x86 systems is not preserved.
// If there is need to support x86, Fiber's Ex-tended API must be used.
#if !defined(_WIN64)
#  error Fiber implementation does not support x86 systems.
#endif
#include <type_traits>
static_assert(std::is_same_v<LPVOID, void*>);
```

Next, while Win32 Fibers can switch from any one Fiber to any other Fiber, we simplify and implement a Fiber that can be resumed, but when suspends - goes to the point of last resume (its parent). Hence, `void* _parent_fiber`.

`FiberProc()` we pass to `::CreateFiber()` gets a reference to a given `Fiber` instance and runs it. FiberProc must never end, hence a while loop and a suspend if a call to run() ends:

```
static void FiberProc(void* parameter)
{
    assert(parameter);
    Fiber& self = *static_cast<Fiber*>(parameter);
    while (true)
    {
        self.run();
        self.suspend();
    }
}
```

To suspend a Fiber is to switch to our parent fiber who did resume us:

```
void suspend()
{
    assert(_parent_fiber);
    void* switch_to_fiber = _parent_fiber;
    _parent_fiber = nullptr;
    ::SwitchToFiber(switch_to_fiber);
}
```

To resume a Fiber, we simply call `::SwitchToFiber()` for a `_fiber` we created. However, since when suspending, we need to know how to switch back, we remember current fiber as our parent:

```
void resume()
{
    assert(_parent_fiber == nullptr);
    _parent_fiber = ::GetCurrentFiber();
    ::SwitchToFiber(_fiber);
```

```
}
```

Remember, `.resume()` is, basically, a first call to a Fiber from within a main function/thread. That means that `::GetCurrentFiber()` is invoked in a context of `main()`:

```
int main()
{
    Fiber fiber;
    fiber.resume(); // call to ::GetCurrentFiber()??
```

To make that work, specifically for Win32 API, main thread needs to become a Fiber. This is what we do by having a simple RAII class:

```
struct Fiber::Boot
{
    Boot(const Boot&) = delete;
    Boot()
    {
        const void* fiber = ::ConvertThreadToFiber(nullptr);
        assert(fiber);
    }
    ~Boot()
    {
        const auto ok = ::ConvertFiberToThread();
        assert(ok);
    }
};
```

Hence, main and/or any other thread that may use Fibers, needs to scope `Fiber::Boot` variable on top:

```
int main()
{
    Fiber::Boot _;

    Fiber fiber;
    std::println("main1");
    fiber.resume();
    std::println("main2");
    fiber.resume();
    std::println("main3");
}
```

It's possible to avoid that by calling `::ConvertThreadToFiber()` on each and every call to `Fiber::resume()`; choose what you like more.

Given a main() above, we:

- create a fiber, which is suspended initially
- print “main1”
- first call to `.resume()` switches us back to `FiberProc` that invokes `Fiber::run()`:

```
void Fiber::run()
{
    std::println("fiber1");
    suspend();
    std::println("fiber2");
}
```

that:

- prints “fiber1”
- suspends itself, which switches back to main (our parent fiber)
- main prints “main2”, resumes fiber
- fiber prints “fiber2”, exits run(), but immediately suspends itself
- main prints “main3”

All at once:

```
main1
fiber1
main2
fiber2
main3
```

6.2 #switching between Fibers

source code.

Section above shows how we can switch from a main to a different Fiber. Fiber on it’s own, when suspended, switches back to its invoker/resumer. However, instead of suspend, Fiber can switch execution to a different Fiber. While not quite used anywhere else, lets show the possibility. Our `Fiber::run()` must be able to run different code; lets inject any user-defined callback and run it:

```
struct Fiber
{
    // For an example:
    std::function<void ()> _callback;
    // ...
};

void Fiber::run()
{
    assert(_callback);
    _callback();
```

```
}
```

Where main can now drive 2 Fibers at once:

```
void Fiber::run()
{
    assert(_callback);
    _callback();
}

int main()
{
    Fiber::Boot _;

    Fiber fiber1;
    Fiber fiber2;
    fiber1._callback = [&fiber2, self = &fiber1]()
    {
        std::println("fiber1: start");
        fiber2.resume(); // switch to fiber2
        std::println("fiber1: END");
        self->suspend(); // switch to main (=our parent)
    };
    fiber2._callback = [self = &fiber2]()
    {
        std::println("fiber2: start");
        self->suspend(); // switch back to fiber1 (=our parent)
        std::println("fiber2: END");
        self->suspend(); // switch to main (=our parent)
    };
    std::println("main1");
    fiber1.resume();
    std::println("main2");
    fiber2.resume();
    std::println("main3");
}
```

we:

- create 2 fibers; they are suspended
- print “main1”
- resume fiber1, that starts run(), that prints “fiber1: start”
- instead of suspend/switch to main, we then resume/switch to fiber2
- fiber2 resume prints “fiber2: start” initially; then
- we suspend fiber2
- that brings us back to our parent = fiber1
- print “fiber1: END”

- fiber1 suspends itself, that switches back to main
- print “main2”
- main resumes fiber2 which executes last print
- fiber2 prints “fiber2: END”, ends execution by suspending itself
- fiber2 suspend brings back to main
- main prints “main3”

which prints:

```
main1
fiber1: start
fiber2: start
fiber1: END
main2
fiber2: END
main3
```

so, while we used Win32 Fibers to implement asymmetric coroutines:

- we can still freely switch between Fiber(s)
- it does not matter if Fiber was resumed/activated from a thread (main) or the other Fiber
- the act of switching to/resuming is to give a possibility to execute
- this is similar to C++20 coroutines with its `.resume()`
- nothing magically “runs” in the background; there should be a scheduler that resumes or switches between fibers; same is true for C++20 coroutines
- we were able to interlieve execution of 3 fibers: main, fiber1, fiber2 - all within one system thread; there are no multiple other threads
- fibers are executed concurrently within main thread

Note, how while executing `fiber2` we (a) were resumed from 2 different contexts and (b) changed the parent in meantime, switching to different contexts:

```
1 fiber2._callback = [self = &fiber2]()
2 {
3     // ... from fiber1
4     self->suspend(); // switch back to fiber1 (=our parent)
5     // ... from main
6     self->suspend(); // switch to main (=our parent)
7 };
```

6.3 #tasks for Fibers

C++20 coroutines allow for a function to return a value:

```
1 co::Tast<int> MyCoroutine()
2 {
3     co_await Request();
4     co_return 1;
```

```
}

int main()
{
    co::Tast<int> task = MyCoroutine();
    // use a task
}
```

We need to build something similar on top of `Fiber` class:

```
int MyFiber()
{
    this_fiber::suspend();
    return 1;
}

int main()
{
    FiberTask<int> task = FF_async([] { return MyFiber(); });
    // use a task
}
```

Note how:

- any mention of a `Fiber` goes away
- there is nice interface to launch a new Fiber
- we can suspend itself within an execution context; and
- `MyFiber()` simply returns naked `int`; there is no need to mark every coroutine with task-like return type.

In addition, under the hood:

- we use Fibers pool to preallocate N fibers and reuse them
- there is simple Fibers scheduler to drive our Fibers execution
- use exceptions to allow cancellable fiber tasks

Last one is presented just to showcase one of the possibilities; not required for CURL_async_get() Fiber wrapper.

6.4 #cancellable Fiber task with a generic callback

source code.

First, we start with replacing hardcoded `Fiber::run()` with a generic version that uses next interface:

```
// To separate `FiberTask` implementation from an actual `Fiber`.
struct FiberCallbackBase
{
```

```
    void run()    { return do_run();    }
    void resume() { return do_resume(); }

    FiberCallbackBase() noexcept = default;
    FiberCallbackBase(const FiberCallbackBase& rhs) = delete;
protected:
    ~FiberCallbackBase() noexcept = default;
    virtual void do_run() = 0;
    virtual void do_resume() {} // optional
};
```

Where we allow a user to `run()` anything and also allow to be notified that Fiber is resumed. This is needed to implement a task cancelation: when canceled, user's defined `resume()` throws an exception to unwind everything in a context of fiber execution. Hence, our `run()` implementation becomes:

```
static void FiberProc(void* parameter)
{
    Fiber& self = *static_cast<Fiber*>(parameter);
    while (true)
    {
        self.run();
        self.suspend();
    }
}

void Fiber::run()
{
    try
    {
        _callback->resume();
        _callback->run();
    }
    catch (...)
    {
        _exception = std::current_exception();
    }
    _callback = nullptr;
}
```

where `_callback` and `_exception` are:

```
struct Fiber
{
    void* _fiber = nullptr;
    void* _parent_fiber = nullptr;
    FiberCallbackBase* _callback = nullptr;
```

```
    std::exception_ptr _exception;
    // ...
};
```

so, when `run()` enters we (a) notify a user it got resumed (first time) and (b) run everything. If exception is thrown, we just remember it and clean-up our current `_callback` - the execution is done and we go to suspend.

What is `_callback`? This is something user can set on a `Fiber` instance allocated from a `FiberPool` (not shown yet). This is going to be done by a `FiberTask` under the hood. For now, we do everything manually:

```
void Fiber::set_callback(FiberCallbackBase& callback)
{
    assert(_callback == nullptr);
    _callback = &callback;
    _exception = {};
}

struct MyCallback : FiberCallbackBase
{
    virtual void do_run() override
    {
        std::println("fiber1");
    }
};

int main()
{
    Fiber::Boot _;

    Fiber fiber;
    MyCallback callback;
    fiber.set_callback(callback);
    std::println("main1");
    fiber.resume();
    std::println("main2");
}
```

For now, we just made everything we had before more complicated. But the difference is that `Fiber::run()` now is generic and can run anything user-defined.

To support exceptions (cancellation) - the rest of `FiberCallbackBase` interface - our old implementation for suspend() needs to be tweak:

```
void suspend()
{
    assert(_parent_fiber);
```

```
    void* switch_to_fiber = _parent_fiber;
    _parent_fiber = nullptr;
    ::SwitchToFiber(switch_to_fiber);
    assert(_callback);
    _callback->resume();
}
```

So once `::SwitchToFiber()` returns - meaning other Fiber was running and we are resumed, we notify a user on a new `resume()`.

Finally, to check that Fiber is doing something (either running or suspended from within a user code), we expose next function:

```
bool is_busy() const
{
    if (_exception)
    {
        std::rethrow_exception(_exception);
    }
    return !!_callback;
}
```

It does cover 2 things: (1) allows to see Fiber has valid user callback and (2) allows to throw any Fiber exception to a user. A bit weird, but does the job.

Overall, the `Fiber` use for a single Task - becomes:

1. allocate a new Fiber from a pool
2. set a new callback
3. resume/run the fiber to an end (is_busy() == false)
4. return Fiber to the pool
5. repeat for a new Task

Note, that from within a Task or FiberCallbackBase, there is no access to a Fiber. How can we suspend a Task then? Following `std::this_thread` convention, we have:

```
namespace this_fiber
{
void suspend()
{
    assert(::IsThreadAFiber());
    void* fiber_data = ::GetFiberData();
    assert(fiber_data);
    Fiber& self = *static_cast<Fiber*>(fiber_data);
    self.suspend();
}
} // namespace this_fiber
```

That allows to simply do `this_fiber::suspend()` to give up task execution and be re-scheduled later.

Linking all the pieces together, low-level Fiber Task may look like this:

```
struct MyFiberTask : FiberCallbackBase
{
    bool _cancel = false;

    virtual void do_run() override
    {
        std::println("fiber1");
        this_fiber::suspend();
        std::println("fiber2");
    }

    virtual void do_resume() override
    {
        if (_cancel)
        {
            _cancel = false;
            throw std::exception("canceled");
        }
    }
};

int main()
{
    Fiber::Boot _;

    Fiber fiber;
    MyFiberTask task;
    fiber.set_callback(task);
    std::println("main1");
    fiber.resume();
    std::println("main2");
    task._cancel = true;
    fiber.resume();
    std::println("main3");
}
```

This is going to be wrapped into nice `FF_async()` interface later. Interesting bit here is that we cancel fiber task in the middle of its execution and "fiber2" console line is not printed:

```
main1
fiber1
```

```
main2
main3
```

6.5 #basic FiberPool

source code.

The idea is simple: fiber task can be small and short-lived, there is no point allocating (and dealocating) a system Fiber every time task is created. Hence, we preallocate a pool of N (=128) system Fibers that are always alive and reuse them. That means:

- there can be up to N Fiber(s) alive; and
- all system resources are allocated once and never released

`FiberPool` interface may look like this:

```
struct FiberPool
{
    using Handle = std::size_t;
    explicit FiberPool(std::size_t size) noexcept

    Handle allocate(FiberCallbackBase& callback);
    void free(Handle handle);

    void suspend(Handle handle);
    void resume(Handle handle);
    bool is_busy(Handle handle) const;

private:
    std::unique_ptr<Fiber[]> _fibers;
    std::size_t _size = 0;
};
```

where `Fiber` instance is hidden from a user and exposed only with a `Handle`. So we can manage an array of fixed size internally:

```
explicit FiberPool::FiberPool(std::size_t size) noexcept
{
    assert(size > 0);
    _fibers.reset(new(std::nothrow) Fiber[size]);
    assert(_fibers.get());
    _size = size;
}
```

To create a Fiber is to invoke an `allocate()`. Note, we go as dumb and as simple as possible - doing linear search to find first free Fiber. It all could be made better, having basic free list allocator for indices as one way to go about it:

```
Handle FiberPool::allocate(FiberCallbackBase& callback)
{
    auto find_first_free_handle = [this]()
    {
        for (std::size_t i = 0; i < _size; ++i)
        {
            if (_fibers[i]._callback == nullptr)
            {
                return Handle(i);
            }
        }
        assert(false);
        return Handle(-1);
    };
    const Handle handle = find_first_free_handle();
    Fiber& fiber = _fibers[handle];
    fiber.set_callback(callback);
    return handle;
}
```

`FiberPool::free()` is, basically, no-op. We asset `Fiber` returned is in "complete" state - finished execution:

```
void FiberPool::free(Handle handle)
{
    assert(handle < _size);
    assert(_fibers[handle]._callback == nullptr);
}
```

Next, `suspend()`, resume() and is_busy() are just a wrappers around Fiber API, but for a `Handle`:

```
void FiberPool::suspend(Handle handle)
{
    assert(handle < _size);
    _fibers[handle].suspend();
}
```

With all this, our previous example running fiber task becomes:

```
int main()
{
    Fiber::Boot _;
    FiberPool fiber_pool{128};

    MyFiberTask task;
    FiberPool::Handle fiber = fiber_pool.allocate(task);
```

```
    fiber_pool.resume(fiber);
    fiber_pool.free(fiber);
}
```

Now, lets get rid of manually created MyFiberTask and make `FiberTask<T>` possible.

6.6 #FiberTask

source code.

We'd like to be able to write something like this:

```
int MyFiber()
{
    this_fiber::suspend();
    return 1;
}

int main()
{
    FiberTaskScheduler scheduler;
    FiberTask<int> task = FF_async(scheduler, [] { return MyFiber(); });
    // ...
    // scheduler.schedule();
}
```

There are several moving parts:

1. we need type-erased implementation of FiberTask that can consume any user-defined lambda/callback, work with any return type `T`; and
2. we need a scheduler that knows how to drive tasks/fibers execution

`FiberTask<T>` allocates a Fiber, runs a given callable/lambda and stores the result. Given FiberTask can be destroyed by a user any time, the way we handle this is to ensure that actual FiberTask state is owned by `FiberTaskScheduler`: if FiberTask is destroyed while still running, we cancel `Fiber` and free it later, once the execution is done.

Interestingly, there could be up to N active Fibers (FiberPool capacity), but FiberTasks count owned by a user can be larger. Consider next code:

```
FiberTask<void> m_tasks[N];
for (...) { m_tasks[i] = FF_async(...) }
// N Fiber tasks completes
//
// User still owns N m_tasks and may query
// the status and result - any time later.
```

Going with more optimized implementation, we could:

- preallocate N system Fibers
- preallocate a memory for up to M (>=N) fiber Tasks
- each fiber Task could be limited in size (let 256 bytes); and
- each Task could free Fiber as soon as task ends

i.e., Fiber and Task lifetime could be different and everything is preallocated; meaning no allocations at runtime.

For simpler implementation, we (a) bind Fiber lifetime to a Task lifetime: Fiber is released only when Task is destroyed; and (b) Task allocates on creation, going with more standard code for type-erased implementation.

With this in mind, we start with FiberTask_Any:

```
struct FiberTask_Any : public FiberCallbackBase
{
    explicit FiberTask_Any(FiberPool& fiber_pool);
    virtual ~FiberTask_Any() noexcept;

    bool is_completed() const;
    void execute();
    void cancel();

private:
    virtual void do_resume() override;

protected:
    FiberPool& _fiber_pool;
    FiberPool::Handle _fiber;
    bool _cancelled = false;
};
```

that implements FiberCallbackBase interface and can be cancelled, but knows nothing on how to store the result of execution:

```
FiberTask_Any::FiberTask_Any(FiberPool& fiber_pool)
    : _fiber_pool(fiber_pool)
    , _fiber(fiber_pool.allocate(*this))
{
}
FiberTask_Any::~FiberTask_Any() noexcept
{
    assert(is_completed());
    _fiber_pool.free(_fiber);
}
```

For cancellation, we throw an exception when Fiber is resumed:

```
class Exception_FiberTaskCancelled : public std::exception
{
};

void FiberTask_Any::cancel()
{
    assert(_cancelled == false);
    _cancelled = true;
}

void FiberTask_Any::do_resume()
{
    if (_cancelled)
    {
        _cancelled = false;
        throw Exception_FiberTaskCancelled{};
    }
}
```

To execute task is to resume a Fiber:

```
void FiberTask_Any::execute()
{
    assert(is_completed() == false);
    _fiber_pool.resume(_fiber);
}
```

Finally, `is_completed()` check looks for a Fiber that ended the execution and/or has an exception thrown:

```
bool FiberTask_Any::is_completed() const
{
    try
    {
        return (_fiber_pool.is_busy(_fiber) == false);
    }
    catch (...)
    {
    }
    return true;
}
```

To handle void and non-void return types, we introduce FiberTask_WithResult:

```
template<typename R>
struct FiberTask_WithResult : public FiberTask_Any
{
public:
```

```
    using FiberTask_Any::FiberTask_Any;
    const R& get() const;
    R&& get_once()
    {
        const R& r = static_cast<const FiberTask_WithResult&>(*this).get();
        return std::move(const_cast<R&>(r));
    }
protected:
    // In case return type is not DefaultConstructiable.
    std::variant<std::monostate, R> _storage;
};

template<>
struct FiberTask_WithResult<void> : public FiberTask_Any
{
public:
    using FiberTask_Any::FiberTask_Any;
    void get() const;
    void get_once()
    {
        return static_cast<const FiberTask_WithResult&>(*this).get();
    }
};
```

where non-void FiberTask places the result in the `_storage` data member. `get_once()` is just a `get()` that give rvalue reference back so it could be moved from. `get()` implementation is just:

```
const R& get() const
{
    // Populate exception, if any.
    const bool running = _fiber_pool.is_busy(_fiber);
    assert(running == false);
    assert(_storage.index() == 1);
    return std::get<1>(_storage);
}
```

Note how we must check Fiber's `is_busy()` to populate any exception to the user if something was throws during Fiber execution.

Finally, to run a lambda, we go with FiberTask_Callable:

```
template<typename R, typename C>
struct FiberTask_Callable : public FiberTask_WithResult<R>
{
    using Base = FiberTask_WithResult<R>;
public:
```

```
    explicit FiberTask_Callable(C&& callable, FiberPool& fiber_pool)
        : Base(fiber_pool)
        , _callable(std::move(callable))
    {
    }
private:
    virtual void do_run() override
    {
        if constexpr (std::is_same_v<void, R>)
        {
            _callable();
        }
        else
        {
            this->_storage.template emplace<1>(_callable());
        }
    }
private:
    C _callable;
};
```

FiberTask_Callable is what's going to be allocated on every `FF_async()` call that creates a `FiberTask<T>`:

```
template<typename R>
struct FiberTask
{
    using Task = FiberTask_WithResult<R>;

    template<typename C>
    explicit FiberTask(C&& callable, FiberTaskScheduler& scheduler) noexcept
        : _scheduler(&scheduler)
    {
        using TaskCallable = FiberTask_Callable<R, std::remove_cvref_t<C>>;
        TaskCallable* task = new(std::nothrow) TaskCallable(
            std::forward<C>(callable), scheduler._fiber_pool);
        assert(task);
        scheduler.add_fiber_task(std::unique_ptr<FiberTask_Any>(task));
        _task = task;
    }
    ~FiberTask() noexcept
    {
        destroy_once();
    }
    void destroy_once() noexcept
    {
```

```
        if (_scheduler)
        {
            assert(_task);
            _scheduler->remove_fiber_task(*_task);
            _scheduler = nullptr;
            _task = nullptr;
        }
    }
    decltype(auto) get() const;
    decltype(auto) get_once();
// ...
    FiberTaskScheduler* _scheduler = nullptr;
    Task* _task = nullptr;
};

template<typename C>
auto FF_async(FiberTaskScheduler& scheduler, C&& callable)
{
    using Task = FiberTask<std::invoke_result_t<C>>;
    return Task{std::forward<C>(callable), scheduler};
}
```

`_task` itself is owned by a FiberTaskScheduler:

```
struct FiberTaskScheduler
{
    FiberPool& _fiber_pool;
    std::vector<std::unique_ptr<FiberTask_Any>> _tasks;
    std::vector<FiberTask_Any*> _tasks_to_remove;

    void add_fiber_task(std::unique_ptr<FiberTask_Any>&& task)
    {
        assert(task.get());
        _tasks.push_back(std::move(task));
    }
    void remove_fiber_task(FiberTask_Any& task)
    {
        if (task.is_completed() == false)
        {
            task.cancel();
        }
        _tasks_to_remove.push_back(&task);
    }
```

Note how removing a task is also a cancel if needed.

We are left with `FiberTaskScheduler` execution, which is its `schedule()`:

```cpp
void FiberTaskScheduler::schedule()
{
    while (schedule_once()) {}
}

bool FiberTaskScheduler::schedule_once()
{
    bool repeat = false;
    auto tasks = std::move(_tasks);
    for (auto it = tasks.rbegin(); it != tasks.rend(); ++it)
    {
        std::unique_ptr<FiberTask_Any>& task = *it;
        assert(task);
        if (task->is_completed() == false)
        {
            task->execute();
            repeat |= task->is_completed();
            continue;
        }
        auto it_remove = std::find(_tasks_to_remove.begin()
            , _tasks_to_remove.end(), task.get());
        if (it_remove == _tasks_to_remove.end())
        {
            continue;
        }
        task.reset();
    }
    for (std::unique_ptr<FiberTask_Any>& task : tasks)
    {
        if (task)
        {
            _tasks.push_back(std::move(task));
        }
    }
    return repeat;
}
```

There are 2 moments worth mentioning:

1. tasks are executed from the end - this is needed to execute child tasks first so parent tasks that wait can have progress
2. if any task is completed, we schedule a loop again - again so any parent tasks can complete if child tasks complete

Other then that, we just `.execute()` every task and remove it when completed and was removed.

Finally, running a FiberTask is:

```
int MyFiber()
{
    this_fiber::suspend();
    return 1;
}

int main()
{
    Fiber::Boot _;
    FiberPool fiber_pool{128};
    FiberTaskScheduler fibers_scheduler{fiber_pool};

    FiberTask<int> task = FF_async(fibers_scheduler, &MyFiber);
    while (task.is_completed() == false)
    {
        fibers_scheduler.schedule();
    }
    std::println("{}", task.get()); // prints "1"
}
```

6.7 #waiting for a CURL request with a Fiber

source code.

Assuming we are withing a Fiber context already:

```
void Fiber_Main(CURL_Async curl_async)
{
    const std::string r = CURL_fiber_get_V1(curl_async, "localhost:5001/file1.txt");
    std::println("{}", r);
}
```

Lets write our CURL_fiber_get_V1():

```
std::string CURL_fiber_get_V1(CURL_Async curl_async, const std::string& url)
{
    std::optional<std::string> response;
    CURL_async_get(curl_async, url, &response
        , [](void* user_data, std::string response)
    {
        auto& state = *static_cast<std::optional<std::string>*>(user_data);
        state.emplace(std::move(response));
    });
    while (response.has_value() == false)
    {
        this_fiber::suspend();
```

```
    }
    return std::move(response.value());
}
```

In a way - it's trivial:

1) we start a GET request
2) we wait for a data in a while loop
3) while loop suspends itself if no data yet
4) we use a pointer to local `response` variable since it's valid until Fiber end

Note: compiler can't see or prove that local variable `response` is NOT modified externally, hence the loop is not an infinite loop. There is no need to use volatile/atomic to sidestep the compiler.

Given this - all good... Until someone CANCELS our Fiber Task. Remember, user has a reference to a FiberTask:

```
FiberTask<void> task = FF_async(fibers_scheduler, &Fiber_Main, curl_async);
```

What if user drops/destroys a task while GET request is still in progress?

```
FiberTask::~FiberTask()
{
    assert(_task);
    _scheduler->remove_fiber_task(*_task);

}
```

And remove_fiber_task() just cancels the Task in progress:

```
void FiberTaskScheduler::remove_fiber_task(FiberTask_Any& task)
{
    if (task.is_completed() == false)
    {
        task.cancel();
    }
    _tasks_to_remove.push_back(&task);
}
```

What happens to cancelled Fiber? There are 2 options:

1) we destroy a Fiber, deallocating a memory/system Fiber; and
2) we keep a Fiber in a pool, leaving the memory alive.

Since we have a FiberPool, cancelling a Task/Fiber does not deallocate the memory since system Fiber is still alive. So, here:

```
std::optional<std::string> response;
CURL_async_get(curl_async, url, &response
    , [](void* user_data, std::string response)
```

```
{
    auto& state = *static_cast<std::optional<std::string>*>(user_data);
    state.emplace(std::move(response));
});
```

when callback is invoked and Fiber was/is cancelled, our `response` local variable is still alive and there is no access error to deallocated memory. So.. no issue?

Still, while the memory/Fiber is not deallocated, logically, it's an access to released resource and in practice, in our case - someone else could start another Fiber task that so happens gets the same Fiber. Hence, our CURL_async_get() callback overrides and corrupts the memory of some other task!

Sadly, cancellation is a problem at the end. To handle that - since CURL_async_get() is not cancellable, we need to allocate a flag/state on a heap so it can outlive cancelled Fiber:

```
std::string CURL_fiber_get(CURL_Async curl_async, const std::string& url)
{
    std::optional<std::string> response;

    CancelToken* cancel_token = CancelToken::Make(&response);
    cancel_token->add_ref();

    CURL_async_get(curl_async, url, cancel_token
        , [](void* user_data, std::string response)
    {
        CancelToken& cancel_token = *static_cast<CancelToken*>(user_data);
        if (auto* state = cancel_token.as<std::optional<std::string>>())
        {
            state->emplace(std::move(response));
        }
        cancel_token.release();
    });

    try
    {
        while (response.has_value() == false)
        {
            this_fiber::suspend();
        }
        cancel_token->release();
    }
    catch (...) // including Exception_FiberTaskCancelled
    {
        cancel_token->reset();
        cancel_token->release();
```

```
        throw;
    }

    return std::move(response.value());
}
```

Logically, we:

1) allocate CancelToken, which is ref-counted and there are 2 users: callback and the task itself
2) remember a pointer to a local variable `response`
3) when task is cancelled (exception thrown) - we reset the reference to `response`
4) when callback is invoked, we check if `response` is still alive; CancelToken is guaranteed to be valid since it's ref-counted.

CancelToken implementation is just:

```
struct CancelToken
{
    std::int64_t _ref_count = 0;
    void* _user_data = nullptr;
    static CancelToken* Make(void* user_data)
    {
        CancelToken* token = new(std::nothrow) CancelToken;
        assert(token);
        token->_ref_count = 1;
        token->_user_data = user_data;
        return token;
    }
    void add_ref()
    {
        _ref_count += 1;
    }
    void release()
    {
        _ref_count -= 1;
        assert(_ref_count >= 0);
        if (_ref_count == 0)
        {
            Destroy(this);
        }
    }
    static void Destroy(CancelToken* token)
    {
        assert(token);
        delete token;
```

```
    }
    void reset()
    {
        _user_data = nullptr;
    }
    template<typename T>
    T* as() const
    {
        return static_cast<T*>(_user_data);
    }
};
```

With that in mind, complete CURL_fiber_get() within a FiberTask is:

```
void Fiber_Main(CURL_Async curl_async)
{
    const std::string r1 = CURL_fiber_get(curl_async, "localhost:5001/file1.txt");
    const std::string r2 = CURL_fiber_get(curl_async, "localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
}

int main()
{
    Fiber::Boot _;
    FiberPool fiber_pool{8};
    FiberTaskScheduler fibers_scheduler{fiber_pool};
    CURL_Async curl_async = CURL_async_create();
    FiberTask<void> task = FF_async(fibers_scheduler, &Fiber_Main, curl_async);
    while (task.is_completed() == false)
    {
        CURL_async_tick(curl_async);
        fibers_scheduler.schedule();
    }
    CURL_async_destroy(curl_async);
}
```

Note, FF_async() was extended to accept extra arguments.

6.8 #waiting for multiple a FiberTasks

source code.

Given a list of FiberTasks, we just wait for completion in the loop:

```
template<typename... Ts>
std::tuple<Ts...> FF_await_all(FiberTask<Ts>... tasks)
```

```
{
    auto wait_task = [](auto task) -> auto
    {
        while (task.is_completed() == false)
        {
            this_fiber::suspend();
        }
        return task.get_once();
    };
    return {wait_task(std::move(tasks))...};
}
```

Here, we intentionally accept `tasks` by value so that all the tasks are consumed and, more importantly, if await is cancelled, all of the tasks are also discarded due to stack unwinding.

To be able to wait for multiple CURL requests, we need to have FiberTask. We do:

```
FiberTask<std::string> CURL_fiber_get(
      CURL_Async curl_async
    , const std::string& url
    , FiberTaskScheduler& fiber_scheduler)
{
    return FF_async(fiber_scheduler, [=]()
    {
        return CURL_fiber_get(curl_async, url);
    });
}
```

Finally, doing 2 concurrent requests with Fibers is:

```
void Fiber_Main(FiberTaskScheduler* fiber_scheduler, CURL_Async curl_async)
{
    auto [r1, r2] = FF_await_all(
        CURL_fiber_get(curl_async, "localhost:5001/file1.txt", *fiber_scheduler),
        CURL_fiber_get(curl_async, "localhost:5001/file2.txt", *fiber_scheduler)
        );
    std::println("{}", r1);
    std::println("{}", r2);
}
```

We need `FiberTaskScheduler` to create children Fiber tasks, so it's a bit more lengthy to type all that compared with the same code for C++20 coroutines.

It's possible to get rid of `fiber_scheduler` by having it implicitly available as a global thread local variable, so CURL_fiber_get()/or FF_async() becomes:

```
FiberTask<std::string> CURL_fiber_get(
      CURL_Async curl_async
    , const std::string& url
    , FiberTaskScheduler& fiber_scheduler = FiberTaskScheduler::get_current())
```

However, that's mostly irrelevant in our context.

7 #building std::future API

source code, sample app.

Having CURL_async_get() leads to the next implementation that wraps everything into a std::future:

```
std::future<std::string> CURL_future_get(
    CURL_Async curl_async, const std::string& url)
{
    using Promise = std::promise<std::string>;
    Promise* promise = new(std::nothrow) Promise;
    assert(promise);
    std::future<std::string> future = promise->get_future();
    CURL_async_get(curl_async, url, promise
        , [](void* user_data, std::string response)
    {
        Promise* promise = static_cast<Promise*>(user_data);
        promise->set_value(std::move(response));
        delete promise;
    });
    return future;
}
```

Everything is just what std:: exposes. Note how we need to allocate our promise so it can be alive until the end of the request.

All in all, we:

1. allocate `std::promise`
2. `std::promise` allocates shared state, used by std::future
3. CURL_async_get allocates `std::string` to write a response
4. CURL_async_get allocates `std::function` for a generic callback
5. CURL_async_get allocates `std::unordered_map` node to remember what to call when
6. .. and probably something else (CURL internals, etc)

Anyway, std::future is missing useful bits to work with it in a non-blocking manner:

- no built-in `.is_ready()` check

- no built-in `.then()` continuation, see Design and evolution of C++ future continuations

Faking `is_future_ready()` with:

```
template<typename T>
bool is_future_ready(const std::future<T>& future)
{
    const std::future_status status = future.wait_for(std::chrono::seconds::zero());
    return (status == std::future_status::ready);
}
```

allows to finally write something among the lines:

```
int main()
{
    CURL_Async curl_async = CURL_async_create();
    std::future<std::string> result = CURL_future_get(
        curl_async, "localhost:5001/file1.txt");
    while (is_future_ready(result) == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
    std::println("{}", result.get());
}
```

Missing std::future features makes it not composable; waiting 2 tasks to finish is the same as checking 2 flags to become true; giving not much of a win compared to direct use of CURL_async_get().

8 #building task API with .then() support

source code, sample app.

Lets build std::future-like Task type that supports `.then()` continuations.

Everything is the same as std::future, but there is a support to chain operations, see std::experimental::future::then() and boost::future::then():

```
Task<std::string> CURL_task_get(CURL_Async curl_async, const std::string& url);

int main()
{
    Task<void> task = CURL_task_get(curl_async, "localhost:5001/file1.txt")
        .then([](std::string response)
    {
        std::println("{}", response);
    });
```

```
    // ...
}
```

.then() returns new Task that holds the value of whatever our callable returned (`Task<void>` in the example above), but it can be a value:

```
Task<std::string> t;
Task<int> next = t.then([](std::string)
{
    return 573;
});
```

With such an API, .then() needs to return a Task to the caller while also holding same task data to set the value to it once previous task completes. That requires to have a separate reference-counted block under the hood. Lets start with a simple Task_RefCount then:

```
struct Task_RefCount
{
    std::int64_t _ref_count = 1;
    Task_RefCount() noexcept = default;
    Task_RefCount(const Task_RefCount&) = delete;

    void add_ref()
    {
        assert(_ref_count >= 0);
        _ref_count += 1;
    }

    bool release()
    {
        assert(_ref_count >= 1);
        _ref_count -= 1;
        if (_ref_count == 0)
        {
            Deallocate(this);
            return true;
        }
        return false;
    }

    static void Deallocate(Task_RefCount* ptr)
    {
        assert(ptr);
        delete ptr;
    }

```

```
    virtual ~Task_RefCount() = default;
};
```

We also re-use std::unique_ptr to hold the value of Task_RefCount, so there is no need to manually implement move operations:

```
struct Task_Release
{
    void operator()(Task_RefCount* ptr) noexcept
    {
        assert(ptr);
        ptr->release();
    }
};

template<typename T>
using Task_Ptr = std::unique_ptr<T, Task_Release>;
```

Now, our Task holds any value T. Lets implement Task_Storage that knows how to handle that since there are at least 2 complications:

1) T can be void and T can be a reference; those can't be directly stored and
2) T can be not default constructible

```
template<typename T>
struct Task_Storage : Task_RefCount
{
    std::variant<std::monostate, T> _value;
    template<typename U>
    void set_value(U&& v) noexcept
    {
        assert(has_value() == false);
        _value.template emplace<1>(std::forward<U>(v));
        finish();
    }
    bool has_value() const
    {
        return (_value.index() == 1);
    }
    const T& get() const noexcept
    {
        assert(has_value());
        return std::get<1>(_value);
    }
    T consume() noexcept
    {
        assert(has_value());
```

```
        T v{std::move(std::get<1>(_value))};
        _value.template emplace<0>();
        return v;
    }
    template<typename F>
    decltype(auto) dispatch_once(F&& f)
    {
        return std::invoke(std::forward<F>(f), consume());
    }
    virtual void finish() = 0;
};
```

`std::variant<std::monostate, T>` is there to solve non-default-constructible case (could be `std::optional<T>`, I prefer variant) and `T&` and `void` cases are Task_Storage specializations:

```
template<typename T>
struct Task_Storage<T&> : Task_RefCount
{
    T* _value = nullptr;
};

template<>
struct Task_Storage<void> : Task_RefCount
{
    bool _has_value = false;
};
```

The code for those is largely the same as our base case. While `set_value()`, `has_value()` and `get()` are self-explanatory, we also have:

- `T consume()`: just a `get()` that move the value out of the storage;
- `dispatch_once(F)`: calls the callable F with our value; mostly needed to specialize void case so F could be invoked with no arguments; and
- pure virtual `finish()` that is invoked when we `set_value()`: so we can handle .then() implementation/completion/continuation.

Strictly speaking, that's all we need to implement just `Task<T>`. Still, for `.then(F)`, we need to store any user-defined callable F. Hence, we extend Task_Storage with Task_Callback:

```
template<typename T>
struct Task_Callback : Task_Storage<T>
{
    std::move_only_function<void ()> _callback;
    virtual void finish() override
    {
        if (_callback)
```

```
        {
            std::move_only_function<void ()> call = std::move(_callback);
            call();
        }
    }
};
```

where the "trick" is to use std::function to handle all of/any user-defined callable. So, at the end, when we `set_value()` on our Task, this callback is invoked once (if set). We'll use that for .then implementation, but, for now, lets see all of Task details where we simply allocate Task_Callback:

```
template<typename T>
struct Task
{
    using type = T;
    Task_Ptr<Task_Callback<T>> _ptr;
    explicit Task() noexcept
        : _ptr(new(std::nothrow) Task_Callback<T>{})
    {
        assert(_ptr.get());
    }
    ~Task() noexcept = default;
    Task(Task&& rhs) noexcept = default;
    Task& operator=(Task&& rhs) noexcept = default;
    Task(const Task&) = delete;
    Task& operator=(const Task&) = delete;
```

Then implement the usual getters/setters (set_value() and get() handles void case too):

```
    decltype(auto) get() const
    {
        assert(is_valid());
        return _ptr->get();
    }

    decltype(auto) get_once()
    {
        assert(is_valid());
        return _ptr->consume();
    }

    template<typename... Ts>
        requires ((sizeof...(Ts)) == 0)
    void set_value(Ts&&... vs)
    {
```

```
        assert(is_valid());
        _ptr->set_value();
    }
    template<typename... Ts>
        requires ((sizeof...(Ts)) == 1)
    void set_value(Ts&&... vs)
    {
        assert(is_valid());
        (_ptr->set_value(std::forward<Ts>(vs)), ...);
    }

    bool has_value() const
    {
        assert(is_valid());
        return _ptr->has_value();
    }

    bool is_valid() const
    {
        return (_ptr.get() != nullptr);
    }
```

std::future does not expose set_value on a future directly to limit the number of API misuse, std::promise needs to be used; but we go with a simpler code. Finally, to expose reference-counter task under the hood, we add `share()` API:

```
explicit Task(Task_Callback<T>* ptr) noexcept // explicit share
    : _ptr(ptr)
{
    assert(ptr);
    ptr->add_ref();
}
Task share()
{
    assert(is_valid());
    return Task(_ptr.get());
}
```

which allows to give a Task to the user, but also have a reference to it to set the value later, so we can implement .then(), which roughly does:

```
auto Task<T>::then(F&& f)
{
    using Return = invoke_result_t<F, T>;
    Task<Return> task;
    _ptr->_callback = [target = task.share()]() // **HERE: remember task
    {
```

```
        target.set_value(...);
    }
    return task;
}
```

There is one complication to implement .then() that easily: implicit task unwrap. Going back to .then() example:

```
Task<std::string> t;
Task<int> next = t.then([](std::string)
{
    return 573;
});
```

when our callable returns `int`, .then() returns `Task<int>`. What if we want to start another task within the callback?:

```
Task<std::string> t;
Task<???> next = t.then([](std::string)
{
    return CURL_task_get("...");
});
```

CURL_task_get() returns `Task<std::string>` on its own. This chaining is so common that instead of just returning `Task<Task<std::string>>` to the user, we want to return `Task<std::string>` directly that would represent the completed result of the inner `CURL_task_get()` operation. Hence, our .then() implementation checks the return type of the callable:

```
template<typename T>
template<typename F>
auto Task<T>::then(F&& f)
{
    using Return = invoke_result_t<F, T>;
    if constexpr (is_task_type_v<Return>)
    {
        Task<typename Return::type> task;
        attach_callback(task, std::forward<F>(f));
        return task;
    }
    else
    {
        Task<Return> task;
        attach_callback(task, std::forward<F>(f));
        return task;
    }
}
```

and if the callable `Return=Task<U>`, we get U out of it and return `Task<U>` to the user; otherwise, it's just `Task<Return>` as it is. The helper `this->attach_callback(target, callback)`:

- waits for task finish
- then invokes callback
- then sets the value of that callback to the target task

Lets see the simple case:

```
void Task<T>::attach_callback(Task<U>& target, F&& callback)
{
    _ptr->_callback = [
          f = std::forward<F>(callback)
        , self_ptr = _ptr.get()
        , target = target.share()
        ]() mutable
    {
        target.set_value(
            self_ptr->dispatch_once(std::move(f))); // calls f
    };

    if (_ptr->has_value())
    {
        _ptr->finish(); // invokes _callback
    }
    // else: to be invoked later, on a first call to set_value()
};
```

where we:

1) remember the `target` task for which the value needs to be set once we are done;
2) setup `_callback` that's going to be executed on our `finish()`;
3) once on finish(), we invoke the callback with dispatch_once() wrapper; and
4) set a value for our target task.

Note that `target` task IS what user gets (here `target` is our `next`):

```
Task<std::string> t;
Task<int> next = t.then([](std::string)
{
    return 573;
});
```

On the other hand, if user callable returns `Task<T>`, we need to attach callback to that Task instead:

```cpp
void Task<T>::attach_callback(Task<U>& target, F&& callback)
{
    using Result = invoke_result_t<F, T>;

    _ptr->_callback = [
          f = std::forward<F>(callback)
        , self_ptr = _ptr.get()
        , target = target.share()
        ]() mutable
    {
        if constexpr (is_task_type_v<Result> == false)
        {
            target.set_value(
                self_ptr->dispatch_once(std::move(f)));
        }
        else // unwrap inner Task
        {
            auto inner_task = self_ptr->dispatch_once(std::move(f));
            inner_task.attach_callback(target, Identity_Callback<U>{});
        }
    };

    if (_ptr->has_value())
    {
        _ptr->finish();
    }
};
```

There are nuances of handling the case when callable returns `void` (see source code).

With that in mind, we can use Tasks like this:

```cpp
Task<int> task;
task.then([](int v)
{
    std::println("got {}", v);
});
std::println("-- set task to 5");
task.set_value(5); // invokes callback
```

or unwrap the inner task like so:

```cpp
Task<void> task;
Task<int> inner;
Task<int> end = task.then([&inner]()
{
```

```
    std::println("task end");
    return inner.share();
});
std::println("-- set task");
task.set_value();
assert(end.has_value() == false); // not yet, inner is in progress
std::println("-- set INNER task to 9");
inner.set_value(9);
std::println("got {}", end.get()); // prints 9
```

That task chaining is less unintuitive once real function that returns Task is invoked instead of dummy inner task. Lets wrap CURL_async_get() to return a Task.

8.1 #wrapping CURL get into a Task

source code.

Implementing `CURL_task_get()` is trivial:

```
Task<std::string> CURL_task_get(CURL_Async curl_async, const std::string& url)
{
    Task<std::string> task;
    void* ptr = task.share_address();
    CURL_async_get(curl_async, url, ptr
        , [](void* user_data, std::string response)
    {
        Task<std::string> task = Task<std::string>::from_address(user_data);
        task.set_value(std::move(response));
    });
    return task;
}
```

Here, to pass void* pointer to our CURL_async_get(), we implement `share_address()`:

```
void* share_address()
{
    assert(_ptr);
    _ptr->add_ref();
    return _ptr.get();
}

static Task from_address(void* ptr)
{
    assert(ptr);
    Task_Callback<T>* task_ptr = static_cast<Task_Callback<T>*>(ptr);
```

```
    Task task{task_ptr};
    task_ptr->release();
    return task;
}
```

That's possible since our internal pointer is reference counted and we have full access to the details.

Doing 2 sequential CURL get requests is

```
static Task<void> Main_Task(CURL_Async curl_async)
{
    return CURL_task_get(curl_async, "localhost:5001/file1.txt")
        .then([curl_async](std::string r1)
    {
        std::println("{}", r1);
        return CURL_task_get(curl_async, "localhost:5001/file2.txt")
            .then([](std::string r2)
        {
            std::println("{}", r2);
        });
    });
}

int main()
{
    CURL_Async curl_async = CURL_async_create();
    Task<void> task = Main_Task(curl_async);
    while (task.has_value() == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

8.2 #waiting for multiple Task requests

source code.

Lets wait for N Tasks - Tasks_WhenAll():

```
template<typename... Ts>
Task<std::tuple<Ts...>> Tasks_WhenAll(Task<Ts>&&... tasks);

int main()
{
    Task<int> t1;
```

```
    Task<char> t2;
    Tasks_WhenAll(t1.share(), t2.share())
        .then([](std::tuple<int, char> v)
    {
        auto [x, y] = v;
        std::println("{} {}", x, y);
    });
    t1.set_value(1);
    t2.set_value('a');
}
```

Given 2 tasks, `Task<int>` and `Task<char>`, waiting for them give us `std::tuple<int, char>` back. Since tasks may be not ready yet, we return `Task<std::tuple<int, char>>`. Conceptually, we need to do something like this:

```
Task<std::tuple<int, char>> Tasks_WhenAll(Task<int> t1, Task<char> t2)
{
    Task<std::tuple<int, char>> out;
    t1.then([](int v)
    {
        // set v to slot 0 of out tuple
        // complete out task if we are last
    });
    t2.then([](char v)
    {
        // set v to slot 1 of out tuple
        // complete out task if we are last
    });
    return out;
}
```

Again, since t1 and t2 may complete out of order, we need to resort to reference counting again and have some internal state that completes when last task completes.

Lets re-use Task_Storage that is reference-counted already. For a case of waiting for `Task<int>` and `Task<char>`, we have:

```
auto Tasks_WhenAll0(Task<int>&& task0, Task<char>&& task1)
{
    auto* state = new(std::nothrow) WhenAll_State0{};
    assert(state);

    state->start_all(std::move(task0), std::move(task1));

    auto task = state->_task.share();
```

```
    state->release(); // already owned by start_all()
    return task;
}
```

where `state` manages all the logic. WhenAll_State0 example is:

```
struct WhenAll_State0 : Task_Storage<std::tuple<int, char>>
{
    Task<std::tuple<int, char>> _task;
    WhenAll_State0() noexcept
    {
        this->_value.template emplace<1>();
    }
    ~WhenAll_State0() noexcept
    {
        finish();
    }
    std::tuple<int, char>& data()
    {
        return std::get<1>(this->_value);
    }
    virtual void finish() override
    {
        _task.set_value(this->consume());
    }
```

where we just use Task_Storage as our implementation detail and on `finish()` - move the data from out internal storage to the task itself. `start_all()` starts waiting for both of the tasks:

```
void WhenAll_State0::start_all(Task<int>&& task0, Task<char>&& task1)
{
    start0(std::move(task0));
    start1(std::move(task1));
}
void WhenAll_State0::start0(Task<int>&& task)
{
    this->add_ref();
    task.then([this, self = Task_Ptr<>{this}](int v)
    {
        std::get<0>(data()) = std::move(v);
    });
}
void WhenAll_State0::start1(Task<char>&& task)
{
    this->add_ref();
    task.then([this, self = Task_Ptr<>{this}](char v)
```

```
    {
        std::get<1>(data()) = std::move(v);
    });
}
```

start0() does:

1. increment our reference count so this state is alive until the end of the task
2. remember itself as `self = Task_Ptr<>{this}` to properly decrement the counter
3. on complete, save the value AND decrement `self` (implicitly destroyed).

More generic implementation of Tasks_WhenAll() does the same:

```
template<typename... Ts>
auto Tasks_WhenAll(Task<Ts>&&... tasks)
{
    static_assert(sizeof...(Ts) >= 1);
    static_assert(std::conjunction_v<is_value_type<Ts>...>
        , "Task<void> or Task<T&> is not implemented");
    static_assert(std::conjunction_v<std::is_default_constructible<Ts> ...>
        , "Waiting a Task<T> with non-default-constructible T is not implemented");

    using Tuple = std::tuple<Ts...>;

    auto* state = new(std::nothrow) WhenAll_State<Task<Tuple>>{};
    assert(state);

    state->start_all(std::tuple<Task<Ts>...>{std::move(tasks)...}
        , std::index_sequence_for<Ts...>{});

    auto task = state->_task.share();
    state->release();
    return task;
}
```

The limitations (no `Task<void>` or `Task<T&>`) could be implemented, but do no change the point.

9 #building C++26 senders

source code, sample app.

Required read: What are Senders Good For, Anyway?.

std::execution, accepted in C++26 P2300R10 provides a framework for managing asynchronous execution. Eric Niebler has a nice intro above among few other

videos on the topic. Some examples of senders and receivers are provided in P2300R10. In addition, stdexec has nice Developer's Guide.

Ultimately, we'd like to write a Sender that wraps our CURL_async_get() and produces a response:

```
SENDER CURL_sender_get(CURL_Async curl_async, const std::string& url);

auto App_Senders(CURL_Async curl_async)
{
    return CURL_sender_get(curl_async, "localhost:5001/file1.txt")
        | stdexec::then([](std::string r)
    {
        std::println("{}", r);
    });
}
```

or, with Sender's coroutines support:

```
stdexec::task<void> App_Senders(CURL_Async curl_async)
{
    const std::string r = co_await CURL_sender_get(
        curl_async, "localhost:5001/file1.txt");
    std::println("{}", r);
}
```

We'll use stdexec for a start. Start from senders basics for a simplified senders implementation.

As of 2026/09/13, stdexec requires at least Visual Studio 2022 version 17.13.0 (MSVC 14.43).

We start by building simplest sender that does nothing:

```
struct Sender
{
    // ...
};

Sender CURL_get()
{
    return Sender{};
}

int main()
{
    stdexec::sync_wait(CURL_get());
}
```

where `sync_wait()` starts our Sender and blocks the execution until the end.

To make it compile, `Sender` is missing several required bits:

```
struct Sender
{
    using sender_concept = stdexec::sender_tag;
    using completion_signatures = stdexec::completion_signatures<
        stdexec::set_value_t ()>;

    template<typename Receiver>
    auto connect(Receiver&& receiver)
    {
        using Receiver_ = std::remove_cvref_t<Receiver>;
        return State<Receiver_>{std::forward<Receiver>(receiver)};
    }
};
```

where we, basically, say that:

1) Sender is... a Sender concept (sender_tag).
2) Our Sender will only return void/nothing - that completion_signatures. And
3) The Sender requires State when connected to any compatible Receiver.

(If nothing clicks, read What are Senders Good For, Anyway?).

Our State is:

```
template<typename Receiver>
struct State
{
    using operation_state_concept = stdexec::operation_state_tag;

    void start() noexcept
    {
        stdexec::set_value(std::move(_receiver));
    }

    Receiver _receiver;
};
```

where we:

1) Say that State is operation_state concept.
2) When operation starts, we immediately complete it by invoking set_value() on our receiver.

With this, we can use our Sender:

```
Sender CURL_get()
{
```

```
    return Sender{};
}

int main()
{
    std::optional<std::tuple<>> x = stdexec::sync_wait(CURL_get());
    assert(x.has_value());
}
```

Note: sync_wait() returns an optional since, in general, sender/operation can fail and a tuple, since sender can return multiple values.

Since Sender concept is compatible with C++20 coroutines awaitable, just implementing a Sender allows to use it in coroutines. So, next coroutine works just fine too:

```
Sender CURL_get()
{
    return Sender{};
}

stdexec::task<void> CURL_get_coro()
{
    co_await CURL_get();
}

int main()
{
    std::optional<std::tuple<>> x = stdexec::sync_wait(CURL_get_coro());
    assert(x.has_value());
}
```

9.1 #sender for a CURL get

source code.

Lets adapt our simple Sender to return ("send") a std::string - i.e., what CURL_async_get() returns:

```
CURL_Get_Sender CURL_sender_get(CURL_Async curl_async, const std::string& url)
{
    return CURL_Get_Sender{}; // TBD
}
```

where CURL_Get_Sender is:

```
template<typename Receiver>
struct CURL_Get_State
```

```
{
    using operation_state_concept = stdexec::operation_state_tag;

    void start() noexcept
    {
        stdexec::set_value(std::move(_receiver), std::string{});
    }

    Receiver _receiver;
};

struct CURL_Get_Sender
{
    using sender_concept = stdexec::sender_tag;
    using completion_signatures = stdexec::completion_signatures<
        stdexec::set_value_t (std::string)>;

    template<typename Receiver>
    auto connect(Receiver&& receiver)
    {
        using Receiver_ = std::remove_cvref_t<Receiver>;
        return CURL_Get_State<Receiver_>{std::forward<Receiver>(receiver)};
    }
};
```

See, completion_signatures now indicate that we set_value(std::string) and inside operation `start()` we now pass an empty std::string.

Can we use our dummy CURL_sender_get() already?

```
int main()
{
    CURL_Async curl_async = CURL_async_create();
    std::optional<std::tuple<std::string>> x =
        stdexec::sync_wait(CURL_sender_get(curl_async, "localhost:5001/file1.txt"));
    assert(x.has_value());
    while (???)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

Yes. But note there are 2 issues:

1) sync_wait() will block execution and we'll never tick our CURL loop; (note: we probably could inject our tick logic into sync_wait run_loop);

2) we don't know when to stop.

To continue main() execution, we must remove sync_wait and manually start the Sender. To do that, we wrap all and every senders logic in Senders_Main():

```
auto Senders_Main(CURL_Async curl_async)
{
    return CURL_sender_get(curl_async, "localhost:5001/file1.txt")
        | stdexec::then([](std::string r)
    {
        std::println("{}", r);
    });
}
```

This is where anything/everything related to senders should be done. Next, we can manually connect() and start() our Senders_Main():

```
int main()
{
    CURL_Async curl_async = CURL_async_create();
    bool done = false;
    auto state = stdexec::connect(Senders_Main(curl_async), AnyReceiver{&done});
    stdexec::start(state);
    while (done == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

where AnyReceiver is a generic receiver that accepts any and all values and just sets true to a bool flag:

```
struct AnyReceiver
{
    using receiver_concept = stdexec::receiver_tag;

    template<typename... Args>
    void set_value(Args&&...) noexcept { finish(); }
    template<typename... Args>
    void set_error(Args&&...) noexcept { finish(); }
    void set_stopped() noexcept        { finish(); }

    void finish() noexcept
    {
        assert(_done);
        assert(*_done == false);
        *_done = true;
```

```
    }

    bool* _done = nullptr;
};
```

Having a main() that properly runs CURL main loop, we can go back to CURL_sender_get() implementation:

```
struct CURL_Get_Sender
{
    using sender_concept = stdexec::sender_tag;
    using completion_signatures = stdexec::completion_signatures<
        stdexec::set_value_t (std::string)>;

    CURL_Async _curl_async = nullptr;
    std::string _url;
    // ...
};

CURL_Get_Sender CURL_sender_get(CURL_Async curl_async, const std::string& url)
{
    return CURL_Get_Sender
    {
        ._curl_async = curl_async,
        ._url = url
    };
}
```

See how CURL_sender_get() does not do any work and passes any required input to CURL_Get_Sender itself. CURL_Get_Sender remembers everything until `connect()` is invoked. This is where operation state is finally constructed:

```
struct CURL_Get_Sender
{
    CURL_Async _curl_async = nullptr;
    std::string _url;

    template<typename Receiver>
    auto connect(Receiver&& receiver)
    {
        using Receiver_ = std::remove_cvref_t<Receiver>;
        return CURL_Get_State<Receiver_>
        {
            ._receiver = std::forward<Receiver>(receiver),
            ._curl_async = _curl_async,
            ._url = std::move(_url)
        };
```

```
    }
};

template<typename Receiver>
struct CURL_Get_State
{
    void start() noexcept;

    Receiver _receiver;
    CURL_Async _curl_async = nullptr;
    std::string _url;
};
```

CURL_Get_State also waits for start() to be invoked. At that point, we:

1) can start the actual operation/work; and
2) our state is guaranteed to be alive until we end it on our side

meaning, that we can pass a pointer to state around and it's guaranteed to be alive:

```
void CURL_Get_State::start() noexcept
{
    assert(_curl_async);
    CURL_async_get(_curl_async, _url
        , this // HERE, pointer to our state
        , [](void* user_data, std::string response)
    {
        CURL_Get_State& state = *static_cast<CURL_Get_State*>(user_data);
        stdexec::set_value(std::move(state._receiver), std::move(response));
    });
}
```

Once CURL_async_get() callback is invoked, we simply invoke the receiver and pass the retrieved response as a result.

One more time, full CURL get sender implementation is:

```
template<typename Receiver>
struct CURL_Get_State
{
    using operation_state_concept = stdexec::operation_state_tag;

    void start() noexcept
    {
        assert(_curl_async);
        CURL_async_get(_curl_async, _url
            , this
```

```
            , [](void* user_data, std::string response)
        {
            CURL_Get_State& state = *static_cast<CURL_Get_State*>(user_data);
            stdexec::set_value(std::move(state._receiver), std::move(response));
        });
    }

    Receiver _receiver;
    CURL_Async _curl_async = nullptr;
    std::string _url;
};

struct CURL_Get_Sender
{
    using sender_concept = stdexec::sender_tag;
    using completion_signatures = stdexec::completion_signatures<
        stdexec::set_value_t (std::string)>;

    CURL_Async _curl_async = nullptr;
    std::string _url;

    template<typename Receiver>
    auto connect(Receiver&& receiver)
    {
        using Receiver_ = std::remove_cvref_t<Receiver>;
        return CURL_Get_State<Receiver_>
        {
            ._receiver = std::forward<Receiver>(receiver),
            ._curl_async = _curl_async,
            ._url = std::move(_url)
        };
    }
};

CURL_Get_Sender CURL_sender_get(CURL_Async curl_async, const std::string& url)
{
    return CURL_Get_Sender
    {
        ._curl_async = curl_async,
        ._url = url
    };
}
```

That allows to use existing, composable senders algorithms. Executing 2 GET requests in sequence now becomes:

```
auto Senders_Main(CURL_Async curl_async)
{
    return exec::sequence(
        CURL_sender_get(curl_async, "localhost:5001/file1.txt")
            | stdexec::then([](std::string r1)
        {
            std::println("{}", r1);
        }),
        CURL_sender_get(curl_async, "localhost:5001/file2.txt")
            | stdexec::then([](std::string r2)
        {
            std::println("{}", r2);
        }));
}
```

9.2 #senders basics: implementing then() and sync_wait()

source code.

std::execution (with stdexec) is complex, generic and handles wide range of cases.

Going with few assumptions and restrictions allows to show the core idea behind and implement basic version of senders and receivers. We'll assume:

- sender can only send one value T; so we have only set_value(T)
- sender can only fail with one value E; so we have only set_error(E)
- value and error is non-void type; so we don't need to branch void case
- no customization points
- no exceptions, everything is noexcept (including user-defined lambdas)
- we skip advanced concepts, like domains, environments and cancellation (see P2300R10)

With that, we can start with coding the basic ideas:

```
// Receiver.
template<typename Receiver, typename T>
void set_value(Receiver&& r, T&& v) noexcept
{
    FWD(r).set_value(FWD(v));
}

template<typename Receiver, typename E>
void set_error(Receiver&& r, E&& e) noexcept
{
    FWD(r).set_error(FWD(e));
}

```

```
template<typename Receiver>
void set_stopped(Receiver&& r) noexcept
{
    FWD(r).set_stopped();
}
```

See, everything we can do with Receiver is those 3 things: set_value(T), set_error(E) and set_stopped().

A note on FWD: typing `std::forward<Receiver>(r)` clutters the details; we simplify:

```
#define FWD(...) ::std::forward<decltype(__VA_ARGS__)>(__VA_ARGS__)
#define MOV(...) ::std::move(__VA_ARGS__)
#define REMOVE_CVR(...) std::remove_cvref_t<__VA_ARGS__>
using void_t = std::monostate;
```

For Sender, we define next API:

```
// Sender.
template<typename Sender, typename Receiver>
auto connect(Sender&& s, Receiver&& r) noexcept
{
    return FWD(s).connect(FWD(r));
}

template<typename Sender>
using sender_value_t = typename REMOVE_CVR(Sender)::value_t;

template<typename Sender>
using sender_error_t = typename REMOVE_CVR(Sender)::error_t;
```

Again, for Sender, we can only (a) connect() it to a Receiver and (b) query the types we would eventually send.

Finishing with Operation API, we can only start():

```
// Operation.
template<typename Operation>
void start(Operation& o) noexcept
{
    o.start();
}
```

With only the pieces above, we can implement `just(v)` Sender:

```
template<typename T>
auto just(T&& v) noexcept
{
```

```
4       return Sender_Just<REMOVE_CVR(T)>{._v = FWD(v)};
5   }
```

so... just() only returns a wrapper (Sender_Just) that remembers a value v we would set later. No actual work done or started. Sender_Just is:

```
1   template<typename T>
2   struct Sender_Just
3   {
4       using value_t = T;
5       using error_t = void_t;
6       T _v;
7       template<typename Receiver>
8       auto connect(Receiver&& r) noexcept
9       {
10          return State_Just<REMOVE_CVR(Receiver), T>{._r = FWD(r), ._v = MOV(_v)};
11      }
12  };
```

where we signal that our operation would set_value() of a type T; there is no error; and when just Sender is connected to a Receiver, we... return the proper state we need; again, no actual work is done:

```
1   template<typename Receiver, typename T>
2   struct State_Just
3   {
4       Receiver _r;
5       T _v;
6       void start() noexcept
7       {
8           set_value(MOV(_r), MOV(_v));
9       }
10  };
```

Operation state (State_Just) is the non-movable, always alive (during operation) state that knows how to start the operation. For a just(1) case, we complete the operation immediately - by invoking set_value() on a Receiver and passing the value we have.

One more time - Receiver could be thought as as callable/lambda. Saying:

> invoking set_value() on a Receiver [...]

is just a fancy way to say that we call a lambda and pass a result value to it.

For now, we can't use a lambda as a Receiver directly (then() needs to be implemented). For a test, lets have a simple one:

```cpp
struct LogReceiver
{
    void set_value(auto&& v) noexcept
    {
        std::println("set_value({})", v);
    }
};

int main()
{
    auto operation_state = connect(just(399), LogReceiver{});
    start(operation_state); // *
}
```

So:

- we have a just Sender
- we create a just Sender instance by calling `just(399)`
- just(399) describes a work that would be done when operation starts
- we connect the Sender to a specific instance of a Receiver (read "callback")
- connecting the Sender and the Receiver gives us Operation state back
- Operation state encodes all that work and data that are needed to execute everything; finally
- we start an operation by invoking a .start() on an operation state

We do not wait for operation completion in the code above because we know everything completes immediately, inline and we see the output:

```
set_value(399)
```

sync_wait does mostly the same under the hood. Lets implement it. sync_wait() accepts any Receiver, starts it, waits for it and returns the result:

```cpp
int main()
{
    auto x = sync_wait(just(1));
}
```

Since any Sender, in general, can return either value T on success, error E on error or cancel signal (.set_stopped()), lets first write a type that can represent that:

```cpp
template<typename T, typename E>
struct sync_wait_result : std::variant<std::monostate, T, E>
{
    bool was_stopped() const { return (this->index() == 0); }
    bool has_value() const { return (this->index() == 1); }
    bool has_error() const { return (this->index() == 2); }
    T& value() { return std::get<1>(*this); }
```

```
    E& error() { return std::get<2>(*this); }
};
```

(Real stdexec::sync_wait() does it differently, mostly because errors are handled differently).

Now, we can:

```
template<typename Sender>
auto sync_wait(Sender&& s) noexcept
{
    using T = sender_value_t<Sender>;
    using E = sender_error_t<Sender>;
    State_SyncWait<T, E> state;
    auto o = connect(FWD(s), Receiver_SyncWait<T, E>{._state = &state});
    start(o);
    state.wait();
    return MOV(state.get());
}
```

See, given a Sender, we:

- query its value type (for success) and error type
- create our internal wait state (State_SyncWait)
- (note it's all local variable on the stack since we implement blocking wait); then
- connect a Sender with our Receiver (that knows how to notify operation end); then
- start an operation; finally
- do a blocking wait

Lets check what Receiver_SyncWait does:

```
template<typename T, typename E>
struct Receiver_SyncWait
{
    State_SyncWait<T, E>* _state = nullptr;
    template<typename U>
    void set_value(U&& v) noexcept
    {
        _state->template emplace<1>(FWD(v));
        _state->_done.set_value();
    }
    template<typename U>
    void set_error(U&& e) noexcept
    {
        _state->template emplace<2>(FWD(e));
        _state->_done.set_value();
```

```
    }
    void set_stopped() noexcept
    {
        _state->template emplace<0>();
        _state->_done.set_value();
    }
};
```

It's a generic Receiver that remembers the values (by forwarding to State_SyncWait) and signalling done event. State_SyncWait is:

```
template<typename T, typename E>
struct State_SyncWait : sync_wait_result<T, E>
{
    std::promise<void> _done;
    sync_wait_result<T, E>& get()
    {
        return *this;
    }
    void wait()
    {
        _done.get_future().wait();
    }
};
```

which holds `sync_wait_result<T, E>` that we use to save the values/errors and `std::promise<void>` which we (ab)use to implement that blocking waiting.

That allows to wait for any Sender:

```
int main()
{
    auto r = sync_wait(just(4));
    assert(r.has_value());
    assert(r.value() == 4);
}
```

Lets continue and implement then() - which allows to attach a lambda to a Sender complete event:

```
int main()
{
    auto r = sync_wait(then(just(3)
        , [](int v) -> void_t
    {
        std::println("{}", v); // prints 3
        return {};
    }));
```

```
    assert(r.has_value());
}
```

then() accepts any Sender and invokes a given lambda when that Sender completes. Lambda accepts the result of the Sender and returns a new value. That new value is what a then-sender would return. Basically, then() transforms another Sender's value.

```
template<typename Sender, typename Lambda>
auto then(Sender&& s, Lambda&& f) noexcept
{
    return Sender_Then<REMOVE_CVR(Sender), REMOVE_CVR(Lambda)>
        {._s = FWD(s), ._f = FWD(f)};
}
```

See, we return Sender_Then that remembers inner sender that we would invoke and a lambda that we would call:

```
template<typename Sender, typename Lambda>
struct Sender_Then
{
    using inner_value_t = sender_value_t<Sender>;
    using value_t = std::invoke_result_t<Lambda, inner_value_t>;
    using error_t = sender_error_t<Sender>;
    Sender _s;
    Lambda _f;
    template<typename Receiver>
    auto connect(Receiver&& r) noexcept
    {
        using Receiver_ = Receiver_Then<REMOVE_CVR(Receiver), Lambda>;
        return ::connect(MOV(_s), Receiver_{._r = FWD(r), ._f = MOV(_f)});
    }
};
```

Sender_Then:

- leaves error value (error_t) unchanged - the same as the original Sender since we do not touch that
- signals that a value type we would return is the result of invoking a lambda; and
- on connect(), we just return original state because its so happens that our then implementation does not need to store anything extra; and
- everything goes to our then-receiver

Receiver_Then is:

```
template<typename Receiver, typename Lambda>
struct Receiver_Then
{
```

```
    Receiver _r;
    Lambda _f;
    template<typename U>
    void set_value(U&& v) noexcept
    {
        ::set_value(MOV(_r), MOV(_f)(FWD(v)));
    }
    template<typename U>
    void set_error(U&& e) noexcept
    {
        ::set_error(MOV(_r), FWD(e));
    }
    void set_stopped() noexcept
    {
        ::set_stopped(MOV(_r));
    }
};
```

See, set_value() just gets a value, calls a lambda and sends that to a next receiver.

That's all. We are forced to return void_t since we do not support void values. But other then that, it's a complete implementation:

```
sync_wait(then(just(3)
    , [](int v) -> void_t
{
    std::println("{}", v); // prints 3
    return {};
}));
```

Finally, to show some asynchronous work, lets implement simple `async()` sender that completes the work on thread pool (using std::async):

```
template<typename Receiver>
struct State_Async
{
    Receiver _r;
    std::future<void> _f;
    void start() noexcept
    {
        _f = std::async(std::launch::async
            , [r = MOV(_r)]() mutable
        {
            ::set_value(MOV(r), void_t());
        });
    }
```

```
};

struct Sender_Async
{
    using value_t = void_t;
    using error_t = void_t;
    template<typename Receiver>
    auto connect(Receiver&& r) noexcept
    {
        return State_Async<REMOVE_CVR(Receiver)>{._r = FWD(r)};
    }
};

auto async()
{
    return Sender_Async{};
}
```

`std::async()` there is just for illustrative purpose, has nothing to do with senders and unused everywhere else.

Connecting async() with then() allows to execute a lambda on a worker thread:

```
int main()
{
    const auto main_thread_id = std::this_thread::get_id();
    std::thread::id work_thread_id;
    auto x = then(async()
        , [&](void_t) -> void_t
    {
        work_thread_id = std::this_thread::get_id();
        return {};
    });
    auto r = sync_wait(MOV(x));
    assert(r.has_value());
    assert(main_thread_id != work_thread_id);
}
```

9.3 #senders basics: implementing when_all()

source code.

Lets implement when_all() senders algorithm (sender adaptor, per p2300r10):

```
int main()
{
    auto op = when_all(
```

```
            just(6)
          , just('v')
          , just(7.2)
          );
    sync_wait(then(MOV(op), [](auto vs) -> void_t
    {
        auto [a, b, c] = vs;
        std::println("{} {} {}", a, b, c);
        return {};
    }));
}
```

Conceptually, we are given a set of N senders and we:

1. start all of them at once;
2. wait for completion of every sender/operation; and
3. finish once everything is done.

Given N senders, we are going to have N values at the end, so we return a tuple of all of the values in a successful case - `std::tuple<int, char, double>` for an example above.

What happens when one of the senders fails? We just return this first error and discard all of the results. Since we store one error value, but there are N error types, we return `std::variant<E1, E2, ...>`.

Similarly, when one of the senders is cancelled and we receive set_stopped(), we finish with set_stopped() too, discarding/ignoring all of the values.

Real stdexec implementation cancels all of the senders yet-in-progress when first error or cancel arrives. For our simplified senders implementation, cancellation is not implemented so we do nothing and simply ensure all of the senders/operations complete (as if cancelled, but none of the senders support cancellation).

Handling variadic set of Senders, each of which could send different types for values and errors is a bit noisy, but lets start with when_all():

```
template<typename... Senders>
auto when_all(Senders&&... ss)
{
    static_assert(sizeof...(Senders) >= 1);
    return Sender_When_All<REMOVE_CVR(Senders)...>{FWD(ss)...};
}
```

where we accept one or more Senders, construct our Sender wrapper - Sender_When_All - which remembers all the Senders, since we need to start them later:

```
template<typename... Senders>
struct Sender_When_All
```

```
{
    using value_t = std::tuple<sender_value_t<Senders>...>;
    using error_t = std::variant<sender_error_t<Senders>...>;

    std::tuple<Senders...> _ss;

    template<typename... Ss>
    Sender_When_All(Ss&&... ss)
        : _ss{FWD(ss)...}
    {
    }

    template<typename Receiver>
    auto connect(Receiver&& r) noexcept
    {
        using State = State_When_All<
              REMOVE_CVR(Receiver)
            , std::tuple<Senders...>
            , std::index_sequence_for<Senders...>
            >;
        return State{._results{._r = FWD(r)}, ._ss{MOV(_ss)}};
    }
};
```

Sender_When_All does:

- propagate its set_value() type which is a tuple of all of the Senders values
- propagate its set_error() type (a variant of errors)
- remember all of the Senders - to be passed later, on connect()

connect() of our when_all() Sender just returns the (operation) state - State_When_All:

```
template<typename Receiver, typename... Senders, auto... Is>
struct State_When_All<Receiver
    , std::tuple<Senders...>     // Senders tuple
    , std::index_sequence<Is...> // Senders indexes
    >
{
    using Results = Result_When_All<
          Receiver
        , std::index_sequence<Is...>
        , Senders...
        >;
    using operations_tuple_t = std::tuple<
        state_storage_t<Senders
            , Receiver_When_All<Is, Results>
```

```
            >...
        >;
    using senders_tuple = std::tuple<Senders...>;

    Results _results;
    senders_tuple _ss;
    operations_tuple_t _states;

    template<auto I>
    void apply_sender()
    {
        using Receiver_ = Receiver_When_All<I, Results>;

        auto& sender = std::get<I>(_ss);
        auto& state_storage = std::get<I>(_states);
        auto& state = state_storage.template emplace<1>(
            ::connect(MOV(sender), Receiver_{._r = &_results}));
        ::start(state);
    }

    void start() noexcept
    {
        (apply_sender<Is>(), ...);
    }
};
```

where starting when_all() operation - starts all of the Senders - they could be "executing" concurrently or even in parallel. We have N operations active.

See how our when_all() state embeds all of the other Senders operations states inline - everything is known at compile time:

```
template<typename Sender, typename Receiver>
using operation_state_t = decltype(::connect(
    std::declval<Sender>(), std::declval<Receiver>()));

// To handle non-default-constructible states.
template<typename Sender, typename Receiver>
using state_storage_t = std::variant<std::monostate
    , operation_state_t<Sender, Receiver>>;

using operations_tuple_t = std::tuple<
    state_storage_t<Senders
        , Receiver_When_All<Senders, Is, Results>
        >...
    >;
```

```
15
16  operations_tuple_t _states;
```

Note, that we need to complete our operation when only last operation ends. To achieve this, we have our own, custom, per-sender Receiver - Receiver_When_All:

```
using Receiver_ = Receiver_When_All<I, Results>;
auto& state = state_storage.template emplace<1>(
    ::connect(MOV(sender), Receiver_{._r = &_results}));
::start(state);
```

so when one of the Senders completes, we notify shared results that given operation (indexed by I) is done:

```
template<typename Sender, auto I, typename Results>
struct Receiver_When_All
{
    Results* _r = nullptr;

    template<typename U>
    void set_value(U&& v) noexcept
    {
        _r->template set_value<I>(FWD(v));
    }
    template<typename U>
    void set_error(U&& e) noexcept
    {
        _r->template set_error<I>(FWD(e));
    }
    void set_stopped() noexcept
    {
        _r->template set_stopped<I>();
    }
};
```

Results must be shared since we count completed operations:

```
template<typename Receiver, auto... Is, typename... Senders>
struct Result_When_All<Receiver, std::index_sequence<Is...>, Senders...>
{
    using Results = std::tuple<
        sync_wait_result<
              sender_value_t<Senders>
            , sender_error_t<Senders>
            >...
        >;
    std::mutex _lock;
```

```
11      Receiver _r;
12      Results _rs;
13      bool _has_error = false;
14      bool _was_stopped = false;
15      std::int32_t _count = sizeof...(Senders);
16
17      template<auto I, typename U>
18      void set_value(U&& v) noexcept
19      {
20          std::lock_guard _(_lock);
21          if ((_has_error || _was_stopped) == false)
22          {
23              std::get<I>(_rs).set_value(FWD(v));
24          }
25          try_finish();
26      }
27      template<auto I, typename U>
28      void set_error(U&& e) noexcept
29      {
30          std::lock_guard _(_lock);
31          if ((_has_error || _was_stopped) == false)
32          {
33              std::get<I>(_rs).set_error(FWD(e));
34              _has_error = true;
35              // real when_all() - also cancels the rest
36          }
37          try_finish();
38      }
39      template<auto I>
40      void set_stopped() noexcept
41      {
42          std::lock_guard _(_lock);
43          if ((_has_error || _was_stopped) == false)
44          {
45              std::get<I>(_rs).set_stopped();
46              _was_stopped = true;
47              // real when_all() - also cancels the rest
48          }
49          try_finish();
50      }
51      // ...
52  };
```

see, when I(th) operation completes with success, we invoke `set_value<I>()` which remembers the value to final tuple of all of the results. In addition:

- when error or cancel/stop was already done, we skip set_value
- for set_error(), we remember the error once
- same for a stop.

Note, how set_error(), set_value() and set_stopped() could be invoked all at the same time since we could have potentially truly parallel operations that complete all at once. Since all of them access same, shared state, we do need to have some kind of lock guard in place.

Finally, try_finish() decrements operations in progress and completes when last operation completes (`_count == 0`):

```
void Result_When_All::try_finish()
{
    _count -= 1;
    assert(_count >= 0);
    if (_count == 0)
    {
        finish();
    }
}

void Result_When_All::finish()
{
    if (_was_stopped)
    {
        ::set_stopped(MOV(_r));
    }
    else if (_has_error)
    {
        std::variant<sender_error_t<Senders>...> es;
        ((std::get<Is>(_rs).has_error()
            ? (void)es.template emplace<Is>(
                MOV(std::get<Is>(_rs).error()))
            : (void)0
            ), ...);
        ::set_error(MOV(_r), MOV(es));
    }
    else
    {
        ::set_value(MOV(_r)
            , std::tuple<sender_value_t<Senders>...>(
                std::get<Is>(_rs).value()...
                )
            );
    }
}
```

Handling templates a bit obfuscates the code, but overall:

- if any Sender was stopped, with complete with set_stopped()
- if there was an Error, we construct our error variant (with a proper index) and complete with set_error(); finally
- when everything completed successfully, we construct a tuple of all of the results and invoke set_value() on our target Receiver.

That allows to wait for N operations:

```
int main()
{
    auto async_just = [](auto v)
    {
        return then(async(), [copy = MOV(v)](void_t) mutable
        {
            return MOV(copy);
        });
    };
    auto op = when_all(
          async_just(6)
        , async_just('v')
        , just(7.2)
        );
    sync_wait(then(MOV(op), [](auto vs) -> void_t
    {
        auto [a, b, c] = vs;
        std::println("{} {} {}", a, b, c);
        return {};
    }));
}
```

(See how we composed async() and then() to create async_just() sender that completes on a worker thread).

In addition, the code for this section implements just_error() and just_stopped() that are identical to just(), but complete with set_error() and set_stopped().

```
int main()
{
    auto r = sync_wait(when_all(
          just(1)
        , just_error('x')
        ));
    assert(r.has_error());
    auto e = r.error();
    assert(e.index() == 1);
    assert(std::get<1>(e) == 'x');
```

```
}
```

9.4 #senders basics: implementing sequence()

source code.

Similar to when_all(), given a set of N Senders, we need to start all of them one by one, in order. We simplify and require all of the Senders to return void on success and error; when one of the Senders fails or is cancelled, we fail or cancel whole sequence:

```
template<typename... Senders>
auto sequence(Senders&&... ss)
{
    static_assert(sizeof...(Senders) >= 1);
    static_assert(std::conjunction_v<
          std::is_same<void_t, sender_value_t<Senders>>...>
        , "we expect all of the Senders to return void on success");
    static_assert(std::conjunction_v<
          std::is_same<void_t, sender_error_t<Senders>>...>
        , "we expect all of the Senders to return void on error");
    return Sender_Sequence<REMOVE_CVR(Senders)...>{FWD(ss)...};
}
```

Sender_Sequence is the same as Sender_When_All, we just return State_Sequence:

```
template<...>
struct State_Sequence
{
    Receiver _r;
    senders_tuple_t _ss;
    operations_tuple_t _states;

    template<auto I>
    void apply_sender();

    void start() noexcept
    {
        apply_sender<0>();
    }
};
```

sequence() operation start() just starts the 1st (at index 0) Sender, by invoking `apply_sender<0>()`. The logic of chaining - starting the next operation is in apply_sender(), where we start next one (I + 1), when previous completes:

```cpp
template<auto I>
void State_Sequence::apply_sender()
{
    auto apply_next = [this](sync_wait_result<void_t, void_t> v)
    {
        if (v.has_error())
        {
            ::set_error(MOV(_r), MOV(v.error()));
        }
        else if (v.was_stopped())
        {
            ::set_stopped(MOV(_r));
        }
        else if constexpr ((I + 1) >= sizeof...(Senders))
        {
            ::set_value(MOV(_r), MOV(v.value()));
        }
        else
        {
            apply_sender<I + 1>();
        }
    };

    auto& sender = std::get<I>(_ss);
    auto& state_storage = std::get<I>(_states);
    auto& state = state_storage.template emplace<1>(
        ::connect(MOV(sender)
            , Receiver_Sequence{._finish = MOV(apply_next)}));
    ::start(state);
}
```

Our internal Receiver_Sequence just invokes completion callback we pass to it, which is our apply_next():

```cpp
struct Receiver_Sequence
{
    using R = sync_wait_result<void_t, void_t>;
    // Should not allocate due to SBO.
    std::function<void (R)> _finish;
    template<typename U>
    void set_value(U&& v) noexcept
    {
        R r;
        r.set_value(MOV(v));
        _finish(MOV(r));
    }
```

```
    template<typename U>
    void set_error(U&& e) noexcept
    {
        R r;
        r.set_error(MOV(e));
        _finish(MOV(r));
    }
    void set_stopped() noexcept
    {
        R r;
        r.set_stopped();
        _finish(MOV(r));
    }
};
```

There are issues with this simplified implementation, like, for instance, it's possible to stack-overflow when all of the Senders finish inline and we start next Sender.

All in all, we can:

```
int main()
{
    auto just_log = [](const char* text)
    {
        return then(async(), [text](void_t) -> void_t
        {
            std::println("{}", text);
            return {};
        });
    };
    sync_wait(sequence(
          just_log("one")
        , just_log("two")
        , just_error(void_t{})
        , just_log("three"))
        );
}
```

which prints:

```
one
two
```

9.5 #senders basics: CURL get

source code.

Finally, we can implement the same CURL_sender_get() we did with stdexec, but using our simplified senders and receivers implementation.

```
template<typename Receiver>
struct State_CURL_Get
{
    void start() noexcept
    {
        assert(_curl_async);
        CURL_async_get(_curl_async, _url
            , this
            , [](void* user_data, std::string response)
        {
            State_CURL_Get& state =
                *static_cast<State_CURL_Get*>(user_data);
            ::set_value(MOV(state._receiver), MOV(response));
        });
    }

    Receiver _receiver;
    CURL_Async _curl_async = nullptr;
    std::string _url;
};

struct Sender_CURL_Get
{
    using value_t = std::string;
    using error_t = void_t;

    CURL_Async _curl_async = nullptr;
    std::string _url;

    template<typename Receiver>
    auto connect(Receiver&& r)
    {
        return State_CURL_Get<REMOVE_CVR(Receiver)>
        {
            ._receiver = FWD(r),
            ._curl_async = _curl_async,
            ._url = std::move(_url)
        };
    }
};

Sender_CURL_Get CURL_sender_get(CURL_Async curl_async, const std::string& url)
{
```

```
44      return Sender_CURL_Get
45      {
46          ._curl_async = curl_async,
47          ._url = url
48      };
49  }
50
51  auto App_SendersV0(CURL_Async curl_async)
52  {
53      return sequence(
54          then(CURL_sender_get(curl_async, "localhost:5001/file1.txt")
55              , [](std::string r1) -> void_t
56          {
57              std::println("{}", r1);
58              return {};
59          }),
60          then(CURL_sender_get(curl_async, "localhost:5001/file2.txt")
61              , [](std::string r2) -> void_t
62          {
63              std::println("{}", r2);
64              return {};
65          })
66          );
67  }
68
69  auto App_SendersV1(CURL_Async curl_async)
70  {
71      return then(
72          when_all(
73                CURL_sender_get(curl_async, "localhost:5001/file1.txt")
74              , CURL_sender_get(curl_async, "localhost:5001/file2.txt")
75              )
76          , [](auto vs) -> void_t
77          {
78              auto [r1, r2] = vs;
79              std::println("{}", r1);
80              std::println("{}", r2);
81              return void_t{};
82          });
83  }
84
85  struct AnyReceiver
86  {
87      template<typename T>
88      void set_value(T&&...) noexcept { finish(); }
```

```
    template<typename E>
    void set_error(E&&) noexcept    { finish(); }
    void set_stopped() noexcept     { finish(); }

    void finish() noexcept
    {
        assert(_done);
        assert(*_done == false);
        *_done = true;
    }

    bool* _done = nullptr;
};

int main()
{
    CURL_Async curl_async = CURL_async_create();

    bool done1 = false;
    auto state1 = ::connect(App_SendersV0(curl_async), AnyReceiver{&done1});
    ::start(state1);

    bool done2 = false;
    auto state2 = ::connect(App_SendersV1(curl_async), AnyReceiver{&done2});
    ::start(state2);

    while ((done1 == false)
        || (done2 == false))
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

See requests with senders/std::execution for comparison with stdexec.

10 #reactive streams

11 #SAMPLES

11.1 #synchronous requests

source code, API section.

For completeness, our trivial case - doing 2 GET requests sequentially:

```cpp
static void App_Blocking()
{
    const std::string r1 = CURL_get("localhost:5001/file1.txt");
    const std::string r2 = CURL_get("localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
}

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

11.2 #requests with callbacks

source code, API section.

With callbacks API, there are 2 variations:

1. doing 2 sequential requests, one after another: see `App_CallbacksV0()`
2. doing 2 requests concurrently: see `App_CallbacksV1()`

```cpp
int main()
{
    App_CallbacksV0(); // sequential
    App_CallbacksV1(); // concurrent
}
```

All the other sections have the same naming convention and the same main().

SEQUENTIAL requests:

```cpp
struct App_StateV0 // sequential
{
    CURL_Async _curl_async{};
    bool _finished = false;
    std::string _r1;
    std::string _r2;

    explicit App_StateV0(CURL_Async curl_async) noexcept
        : _curl_async(curl_async)
    {
    }
    App_StateV0(const App_StateV0&) = delete;
    ~App_StateV0() noexcept
    {
        assert(_finished);
    }
```

```
17
18      void start()
19      {
20          CURL_async_get(_curl_async, "localhost:5001/file1.txt", this
21              , [](void* user_data, std::string response1)
22          {
23              App_StateV0& state = *static_cast<App_StateV0*>(user_data);
24              state._r1 = std::move(response1);
25
26              CURL_async_get(state._curl_async, "localhost:5001/file2.txt", user_data
27                  , [](void* user_data, std::string response2)
28              {
29                  App_StateV0& state = *static_cast<App_StateV0*>(user_data);
30                  state._r2 = std::move(response2);
31                  state._finished = true;
32                  state.done();
33              });
34          });
35      }
36
37      void done()
38      {
39          assert(_finished);
40          std::println("{}", _r1);
41          std::println("{}", _r2);
42      }
43  };
44
45  static void App_CallbacksV0()
46  {
47      CURL_Async curl_async = CURL_async_create();
48      App_StateV0 app{curl_async};
49      app.start();
50      while (!app._finished)
51      {
52          CURL_async_tick(curl_async);
53      }
54      CURL_async_destroy(curl_async);
55  }
```

CONCURRENT requests:

```
1   struct App_StateV1 // concurrent
2   {
3       CURL_Async _curl_async{};
4       std::int32_t _requests = 0;
```

```
    bool _finished = false;
    std::string _r1;
    std::string _r2;

    explicit App_StateV1(CURL_Async curl_async) noexcept
        : _curl_async(curl_async)
    {
    }
    App_StateV1(const App_StateV1&) = delete;
    ~App_StateV1() noexcept
    {
        assert(_finished);
    }

    void start()
    {
        _requests = 2;
        CURL_async_get(_curl_async, "localhost:5001/file1.txt", this
            , [](void* user_data, std::string response)
        {
            App_StateV1& state = *static_cast<App_StateV1*>(user_data);
            state._r1 = std::move(response);
            state.try_finish();
        });
        CURL_async_get(_curl_async, "localhost:5001/file2.txt", this
            , [](void* user_data, std::string response)
        {
            App_StateV1& state = *static_cast<App_StateV1*>(user_data);
            state._r2 = std::move(response);
            state.try_finish();
        });
    }

    void try_finish()
    {
        assert(_requests > 0);
        _requests -= 1;
        if (_requests == 0)
        {
            _finished = true;
            done();
        }
    }

    void done()
```

```
    {
        assert(_finished);
        std::println("{}", _r1);
        std::println("{}", _r2);
    }
};

static void App_CallbacksV1()
{
    CURL_Async curl_async = CURL_async_create();
    App_StateV1 app{curl_async};
    app.start();
    while (!app._finished)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

11.3 #requests with coroutines

source code, API section.

SEQUENTIAL requests:

```
static Co_Task<void> Coro_MainV0(CURL_Async curl_async) // sequential
{
    const std::string r1 = co_await CURL_await_get(curl_async, "localhost:5001/file1.txt");
    const std::string r2 = co_await CURL_await_get(curl_async, "localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
}

static void App_CoroutinesV0()
{
    CURL_Async curl_async = CURL_async_create();
    Co_Task<void> task = Coro_MainV0(curl_async);
    task.resume();
    while (task.is_in_progress())
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

main() loop takes a bit of space, but the actual business logic is almost the same

as regular synchronous code.

CONCURRENT requests:

```
static Co_Task<void> Coro_MainV1(CURL_Async curl_async) // concurrent
{
    auto [r1, r2] = co_await CO_await_all(
        CURL_coro_get(curl_async, "localhost:5001/file1.txt"),
        CURL_coro_get(curl_async, "localhost:5001/file2.txt"));
    std::println("{}", r1);
    std::println("{}", r2);
}

static void App_CoroutinesV1()
{
    CURL_Async curl_async = CURL_async_create();
    Co_Task<void> task = Coro_MainV1(curl_async);
    task.resume();
    while (task.is_in_progress())
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

11.4 #requests with fibers

source code, API section.

SEQUENTIAL requests:

```
static void Fiber_MainV0(CURL_Async curl_async) // sequential
{
    const std::string r1 = CURL_fiber_get(curl_async, "localhost:5001/file1.txt");
    const std::string r2 = CURL_fiber_get(curl_async, "localhost:5001/file2.txt");
    std::println("{}", r1);
    std::println("{}", r2);
}

static void App_FibersV0()
{
    Fiber::Boot _;
    FiberPool fiber_pool{8};
    FiberTaskScheduler fibers_scheduler{fiber_pool};
    CURL_Async curl_async = CURL_async_create();
    FiberTask<void> task = FF_async(fibers_scheduler
        , &Fiber_MainV0, curl_async);
```

```
    while (task.is_completed() == false)
    {
        CURL_async_tick(curl_async);
        fibers_scheduler.schedule();
    }
    CURL_async_destroy(curl_async);
}
```

CONCURRENT requests:

```
static void Fiber_MainV1( // concurrent
    FiberTaskScheduler* fiber_scheduler, CURL_Async curl_async)
{
    auto [r1, r2] = FF_await_all(
        CURL_fiber_get(curl_async, "localhost:5001/file1.txt", *fiber_scheduler),
        CURL_fiber_get(curl_async, "localhost:5001/file2.txt", *fiber_scheduler)
        );
    std::println("{}", r1);
    std::println("{}", r2);
}

static void App_FibersV1()
{
    Fiber::Boot _;
    FiberPool fiber_pool{8};
    FiberTaskScheduler fibers_scheduler{fiber_pool};
    CURL_Async curl_async = CURL_async_create();
    FiberTask<void> task = FF_async(fibers_scheduler
        , &Fiber_MainV1, &fibers_scheduler, curl_async);
    while (task.is_completed() == false)
    {
        CURL_async_tick(curl_async);
        fibers_scheduler.schedule();
    }
    CURL_async_destroy(curl_async);
}
```

11.5 #polling requests with std::futures

source code, API section.

SEQUENTIAL requests:

```
////////////////////////////////////////////////////////////
struct App_StateV0 // sequential
{
    CURL_Async _curl_async{};
```

```
    bool _finished = false;
    std::optional<std::future<std::string>> _r1;
    std::optional<std::future<std::string>> _r2;

    explicit App_StateV0(CURL_Async curl_async) noexcept
        : _curl_async(curl_async)
    {
    }
    App_StateV0(const App_StateV0&) = delete;
    ~App_StateV0() noexcept
    {
        assert(_finished);
    }

    void tick()
    {
        if (_finished)
        {
            return;
        }
        if ((_r1.has_value() == false)
            && (_r2.has_value() == false))
        { // nothing started yet, run 1st task
            _r1 = CURL_future_get(_curl_async, "localhost:5001/file1.txt");
            return;
        }
        if (_r1.has_value()
            && (_r2.has_value() == false))
        { // 1st task is in progress
            if (is_future_ready(_r1.value()) == false)
            {
                return;
            }
            _r2 = CURL_future_get(_curl_async, "localhost:5001/file2.txt");
            return;
        }
        // 2nd task is in progress
        assert(_r1.has_value() && _r2.has_value());
        if (is_future_ready(_r2.value()) == false)
        {
            return;
        }
        _finished = true;
        done();
        _r1.reset();
```

```
        _r2.reset();
    }
    void done()
    {
        assert(_finished);
        std::println("{}", _r1->get());
        std::println("{}", _r2->get());
    }
};

static void App_PollingV0()
{
    CURL_Async curl_async = CURL_async_create();
    App_StateV0 app{curl_async};
    while (app._finished == false)
    {
        CURL_async_tick(curl_async);
        app.tick();
    }
    CURL_async_destroy(curl_async);
}
```

CONCURRENT requests:

```
struct App_StateV1 // concurrent
{
    CURL_Async _curl_async{};
    bool _finished = false;
    std::optional<std::future<std::string>> _r1;
    std::optional<std::future<std::string>> _r2;

    explicit App_StateV1(CURL_Async curl_async) noexcept
        : _curl_async(curl_async)
    {
    }
    App_StateV1(const App_StateV0&) = delete;
    ~App_StateV1() noexcept
    {
        assert(_finished);
    }

    void tick()
    {
        if (_finished)
        {
            return;
```

```
        }
        if ((_r1.has_value() == false)
            && (_r2.has_value() == false))
        { // nothing started yet, launch 2 requests
            _r1 = CURL_future_get(_curl_async, "localhost:5001/file1.txt");
            _r2 = CURL_future_get(_curl_async, "localhost:5001/file2.txt");
            return;
        }
        assert(_r1.has_value() && _r2.has_value());
        if (is_future_ready(_r1.value())
            && is_future_ready(_r2.value()))
        {
            _finished = true;
            done();
            _r1.reset();
            _r2.reset();
        }
    }
    void done()
    {
        assert(_finished);
        std::println("{}", _r1->get());
        std::println("{}", _r2->get());
    }
};

static void App_PollingV1()
{
    CURL_Async curl_async = CURL_async_create();
    App_StateV1 app{curl_async};
    while (app._finished == false)
    {
        CURL_async_tick(curl_async);
        app.tick();
    }
    CURL_async_destroy(curl_async);
}
```

11.6 #requests with Tasks .then()

source code, API section.

SEQUENTIAL requests:

```
static Task<void> Task_MainV0(CURL_Async curl_async) // sequential
{
```

```
    return CURL_task_get(curl_async, "localhost:5001/file1.txt")
        .then([curl_async](std::string r1)
    {
        std::println("{}", r1);
        return CURL_task_get(curl_async, "localhost:5001/file2.txt")
            .then([](std::string r2)
        {
            std::println("{}", r2);
        });
    });
}

static void App_TasksV0()
{
    CURL_Async curl_async = CURL_async_create();
    Task<void> task = Task_MainV0(curl_async);
    while (task.has_value() == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

CONCURRENT requests:

```
static Task<void> Task_MainV1(CURL_Async curl_async) // concurrent
{
    return Tasks_WhenAll(
        CURL_task_get(curl_async, "localhost:5001/file1.txt"),
        CURL_task_get(curl_async, "localhost:5001/file2.txt")
        )
        .then([](std::tuple<std::string, std::string> vs)
    {
        const auto& [r1, r2] = vs;
        std::println("{}", r1);
        std::println("{}", r2);
    });
}

static void App_TasksV1()
{
    CURL_Async curl_async = CURL_async_create();
    Task<void> task = Task_MainV1(curl_async);
    while (task.has_value() == false)
    {
        CURL_async_tick(curl_async);
```

```
    }
    CURL_async_destroy(curl_async);
}
```

11.7 #requests with senders/std::execution

source code, API section.

See the same code done without stdexec, just basic senders and receivers implementation: senders basics: CURL get.

SEQUENTIAL requests:

```
auto Sender_MainV0(CURL_Async curl_async) // sequential
{
    return exec::sequence(
        CURL_sender_get(curl_async, "localhost:5001/file1.txt")
            | stdexec::then([](std::string r1)
        {
            std::println("{}", r1);
        }),
        CURL_sender_get(curl_async, "localhost:5001/file2.txt")
            | stdexec::then([](std::string r2)
        {
            std::println("{}", r2);
        }));
}

void App_SendersV0()
{
    CURL_Async curl_async = CURL_async_create();
    bool done = false;
    auto state = stdexec::connect(Sender_MainV0(curl_async), AnyReceiver{&done});
    stdexec::start(state);
    while (done == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```

CONCURRENT requests:

```
auto Sender_MainV1(CURL_Async curl_async) // concurrent
{
    return stdexec::when_all(
        CURL_sender_get(curl_async, "localhost:5001/file1.txt"),
```

```
        CURL_sender_get(curl_async, "localhost:5001/file2.txt"))
            | stdexec::then([](std::string&& r1, std::string&& r2)
        {
            std::println("{}", r1);
            std::println("{}", r2);
        });
}

void App_SendersV1()
{
    CURL_Async curl_async = CURL_async_create();
    bool done = false;
    auto state = stdexec::connect(Sender_MainV1(curl_async), AnyReceiver{&done});
    stdexec::start(state);
    while (done == false)
    {
        CURL_async_tick(curl_async);
    }
    CURL_async_destroy(curl_async);
}
```