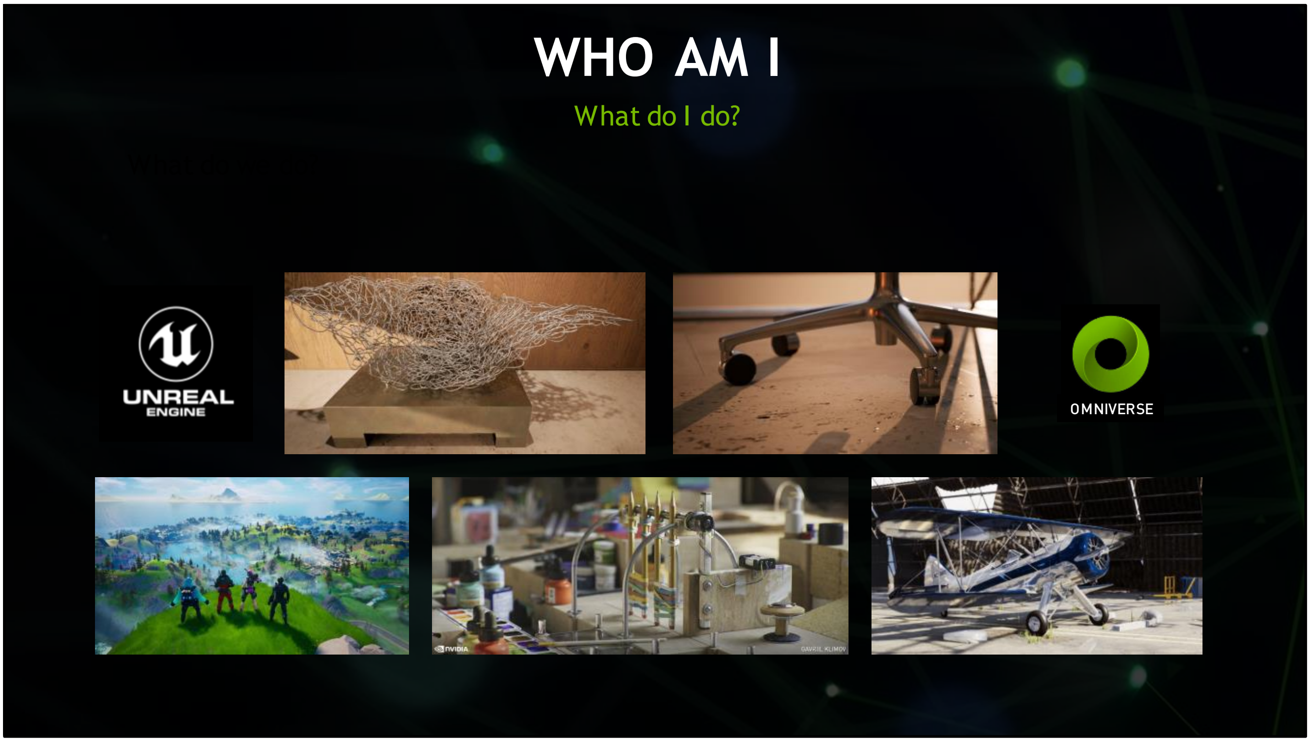
Hi Everyone, Tanki here. Thanks for coming to my talk.

Note:

The sample code I release isn't 100% the same to the one in the slides. The code for slides are more simplified for understanding and to be better fit for display



Before we get started, just a quick self introduction about my background, I am a real-time rendering engineer at NVIDIA working on the RTX renderer in Omniverse. Before that, I was rendering engineer intern on the rendering team of Unreal Engine at Epic Games.

But today, instead of pixels, I am gonna talk about another thing in a game engine – Jobs. Specifically, based on C++ coroutine which we shipped in C++ 20, without standard library support. I because the contents are pretty connected, so I will take all questions at the end but because this is a virtual conference, feel free to put any questions in the Q&A section during the talk. And to avoid missing the Q&A session, I will recommend not to pause the video stream during the talk.

REVIEW C++ COROUTINE IN 10 MINS

- Core concepts: promise object, coroutine handle
- Keywords: `co_return`, `co_await`, `co_yield`(not covered here)
- awaitables, awaiters
- Customization point and how functions are compiled

More resources:
https://gist.github.com/MattPD/9b55db49537a90545a90447392ad3aeb

Before we dive into the system, I want to take 10 mins to have a quick recap about C++ coroutine and share some gotcha moments. I found due to the complexity of the spec, people can be pretty overwhelmed by all different terms, not to mention to put them up together to form a knowledge system.

I hope the following information can be helpful and I attached some links in the slides, or pointers to different part of the specs, for people that need more reference and details.

Promise Object

- User defined type(s) that stores whatever you want/required

Coroutine Handle

- Literally a handle
- Two different types: coroutine_handle<>, coroutine_handle<Promise>
- coroutine_handle<>: resume/destroy coroutine, check status(done)
- coroutine_handle<Promise>: provide access to the promise object(promise)

Useful to know

For convenience:
template<P> using ch<P> = coroutine_handle<P>;

- ch<Promise> -> ch<>: just cast it. Because in spec: template<class Promise> struct coroutine_handle : coroutine_handle<>
- Promise -> ch<Promise>: ch<Promise>::from_promise(Promise&)
- ch<> -> ch<Promise>: more complicated, answer later

More resources:
https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type

So first, Promise Object.

It’s a user defined type. It is part of the coroutine frame. This is the major customization point for user to store persistent coroutine related data.

Then we have coroutine_handle.

…

And it is surprisingly hard at beginning when I trying to figure out how can I access among those three objects. So I listed it here. Notice I did not explain how we can ch<> -> ch<Promise>, because turns out it does not just work and requires some effort and tricks.

Awaitable & awaiter & co_await

- `co_await` is a unary operator(so in case the compiler can find the definition of operator function, it is valid)
- awaitable - type supports co_await operator.
- awaiter - type with await_ready, await_suspend and await_resume defined.
- A set of rules applied at compile time and translate the `co_await` expression into code.
  1. Retreive awaiter & awaitable
  2. Awaiting the awaiter(suspend and resume happens here)

Useful to know

- According to the definition, a type can be awaitable and awaiter at the same time
- It's possible to do co_await `"I am a string"` or in general co_await on some un-awaitable expressions. Or prohibit co_await on certain types. Check out `Promise::await_transform`

More resources:
https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await

co_await is …

Awaitable is …

Awaiter is …

At the compile time, when compiler see co_await expression, it will first retrieve both awaiter and awaitable according to certain rules. Then await on the awaiter.

Customization point - How a compiler may work

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

Non-related details are omitted here, do not 100% follow the spec, for understanding the general behavior only.

More resources:
https://godbolt.org/z/63UnaV (credit to Lewis Baker)

Okay, time for the big boss.

The major mind struggle I had when I try to understand how coroutine works is there are all those customization points, it's like a super extreme version of iterator, so it's hard to connect everything together in code. Especially when compiler will mess with your code.

So let's reveal the black box here and flush out as much as we can.

Here we have simple code example, one is a sum function that adds two numbers together. And another waaaaaaay fancier function.. That adds four numbers together.

And as we all know, if we want to add a, b, c, d together, we need to add ab together, add cd together and add their sums together.

Customization point - How a compiler may work

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```


So first let’s look how compiler transform sum coroutine. I know this is a little bit of overwhelming but don’t worry about it.

Customization point - How a compiler may work

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

8

These code is where we start.

Customization point - How a compiler may work

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

9

So we invoke a coroutine in some function first.

Customization point - How a compiler may work

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

10

And this is what we write in source code, which is what we think we are calling

Customization point - How a compiler may work

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};
```

```
task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

The function we call during runtime

11

And this is what we are actually calling at runtime because compilation happens

Customization point - How a compiler may work

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};
```

When you invoke the coroutine…

Construct coroutine frame

Generate return object

First resume

```cpp
task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```


And according to the spec, several things happens when we invoke a coroutine.

We first construct the coroutine frame;

Second we generate the return object;

And there are some other steps happen in this resume function, but let’s just call it the first time we call into the resume.

Customization point - How a compiler may work

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

Construct coroutine frame

13

We don’t really need to worry about all the code for the coroutine frame. But there are two parts we do care about here, the first one is the params, which includes other input parameters of the function.

The second part is the initial suspend awaitable and final suspend awaitable. We will see them many times in the rest of the talk, and they will have different names in the job system.

Customization point - How a compiler may work

- get_return_object

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

Generate return object

14

Then next, we invoke get_return_object function to generate the return object. As you can see in the original function, it matches the return type of our original coroutine return type.

So that’s how we get that task object. It’s not something we directly return by writing a return statement, it’s returned by implementing get_return_object.

Also, notice I will list all the customization points at the top right corner as a quick recap.

Customization point - How a compiler may work

- get_return_object

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

First resume

15

Finally, after these steps, we are ready to run some actual logics.

Customization point - How a compiler may work

- get_return_object

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr);

struct __sum_frame
{
    __sum_frame(int a, int b);
    promise_type promise;
    int resumeIndex = 0;
    void(* const resumefn)(void*) = __sum_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;
    };
};

task<int> sum(int a, int b) {
    auto frame = make_unique<__sum_frame>(a, b);
    decltype(auto) returnObject = frame->promise.get_return_object();
    __sum_resume(static_cast<void*>(frame.release()));
    return returnObject;
}
```

coroutine_handle::resume()

First resume

16

Notice that we have this __sum_resume function, which is a generally a big state machine generated by compiler.

This function is invoked when we call coroutine_handle::resume

NVIDIA

Customization point - How a compiler may work

- get_return_object

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
    using promise_type = __sum_frame::promise_type;
    using handle_t = std::experimental::coroutine_handle<promise_type>;

    auto* frame = static_cast<__sum_frame*>(ptr);
    std::experimental::coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = std::exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        // initial_suspend
    }
```

Initial suspend

```
    switch(resumeIndex) {
        case 1: goto __sp1;
        default: ABORT();
    }
__sp1:

    // do work
```

Coroutines are translated into state machine

```
__final_suspend:
```

Final suspend, house keeping

```
}
```

17 NVIDIA

During the execution of coroutine there are a couple of different steps will be involved. the first is initial suspend a and then we start the main logic after everything is done we do the final suspend and the cleaning up

Customization point - How a compiler may work

- get_return_object

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

We are now here :)

```
void __sum_resume(void* ptr)
{
    auto* frame = static_cast<__sum_frame*>(ptr);
    coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        frame->initial_suspend_awaitable = frame->promise.initial_suspend();
        if (!frame->initial_suspend_awaitable.await_ready()) {
            frame->resumeIndex = 1;
            frame->initial_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
            return;
        }
        resumeIndex = 1;
    }
//…
```

Initial suspend

18

So first we have initial_suspend. It does several things.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
    auto* frame = static_cast<__sum_frame*>(ptr);
    coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        frame->initial_suspend_awaitable = frame->promise.initial_suspend();
        if (!frame->initial_suspend_awaitable.await_ready()) {
            frame->resumeIndex = 1;
            frame->initial_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
            return;
        }
        resumeIndex = 1;
    }
//…
```

19

We first retrieve initial suspend awaitable, which is returned by the initial suspend method defined on the promise object.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum_resume(void* ptr)
{
    auto* frame = static_cast<__sum_frame*>(ptr);
    coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        frame->initial_suspend_awaitable = frame->promise.initial_suspend();
        if (!frame->initial_suspend_awaitable.await_ready()) {
            frame->resumeIndex = 1;
            frame->initial_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
            return;
        }
        resumeIndex = 1;
    }
//…
```


Then we start to ask the question, are you ready?

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
    auto* frame = static_cast<__sum_frame*>(ptr);
    coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        frame->initial_suspend_awaitable = frame->promise.initial_suspend();
        if (!frame->initial_suspend_awaitable.await_ready()) {
            frame->resumeIndex = 1;
            frame->initial_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
            return;
        }
        resumeIndex = 1;
    }
//…
```

21

If not, okay, let's suspend you. Both await_ready and await_suspend are customization points.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum_resume(void* ptr)
{
    auto* frame = static_cast<__sum_frame*>(ptr);
    coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        frame->initial_suspend_awaitable = frame->promise.initial_suspend();
        if (!frame->initial_suspend_awaitable.await_ready()) {
            frame->resumeIndex = 1;
            frame->initial_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
            return;
        }
        resumeIndex = 1;
    }
//…
```

Current coroutine being executed

22

One thing worth notice is here, remember what are we passing into the await_suspend, it is our current coroutine being executed. This is important, we will see why later.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
    using promise_type = __sum_frame::promise_type;
    using handle_t = std::experimental::coroutine_handle<promise_type>;

    auto* frame = static_cast<__sum_frame*>(ptr);
    std::experimental::coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = std::exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        // initial_suspend
    }
```

Initial suspend

```
    switch(resumeIndex) {
        case 1: goto __sp1;
        default: ABORT();
    }
__sp1:

    // do work
```

Coroutines are translated into state machine

```
__final_suspend:
```

Final suspend, house keeping

```
}
```

23

Then after initial suspend, and coroutine may or may not be suspended. But anyway, when it is resumed eventually… hopefully, we are ready to run our actual logic.

NVIDIA

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__sp1:
    frame->initial_suspend_awaitable.await_resume();
    frame->initial_suspend_awaitable.~__sum_frame::initial_suspend_awaitable();

    int result = frame->params.a + frame->params.b;

    frame->promise.return_value(result);
    goto __final_suspend;
//…
}
```

Coroutines are translated into state machine

24 NVIDIA

How coroutine is implemented is generally break the whole execution into several parts, and compose the whole process as an state machine and use coroutine frame to maintain the necessary intermediate states.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__sp1:
    frame->initial_suspend_awaitable.await_resume();
    frame->initial_suspend_awaitable.~__sum_frame::initial_suspend_awaitable();

    int result = frame->params.a + frame->params.b;

    frame->promise.return_value(result);
    goto __final_suspend;
//…
}
```

25

In this example, we first call await_resume for initial suspend awaitable. Then do house keeping.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__sp1:
    frame->initial_suspend_awaitable.await_resume();
    frame->initial_suspend_awaitable.~__sum_frame::initial_suspend_awaitable();

    int result = frame->params.a + frame->params.b;

    frame->promise.return_value(result);
    goto __final_suspend;
//…
}
```

26

Then eventually, we are adding two numbers now.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__sp1:
    frame->initial_suspend_awaitable.await_resume();
    frame->initial_suspend_awaitable.~__sum_frame::initial_suspend_awaitable();

    int result = frame->params.a + frame->params.b;

    frame->promise.return_value(result);
    goto __final_suspend;
//…
}
```


Then here, we have a co_return expression. Depends on whether there is an expression after the co_return, we call either return_value or return_void to report the result.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
    using promise_type = __sum_frame::promise_type;
    using handle_t = std::experimental::coroutine_handle<promise_type>;

    auto* frame = static_cast<__sum_frame*>(ptr);
    std::experimental::coroutine_handle<> symmetricTransferTarget;

    int resumeIndex = std::exchange(frame->resumeIndex, -2);
    if (resumeIndex == 0) {
        // initial_suspend
    }
```

Initial suspend

```
    switch(resumeIndex) {
        case 1: goto __sp1;
        default: ABORT();
    }
__sp1:

    // do work
```

Coroutines are translated into state machine

```
__final_suspend:
```

Final suspend, house keeping

```
}
```

28

And after we have done everything, it's time to call the final_suspend.

Customization point - How a compiler may work

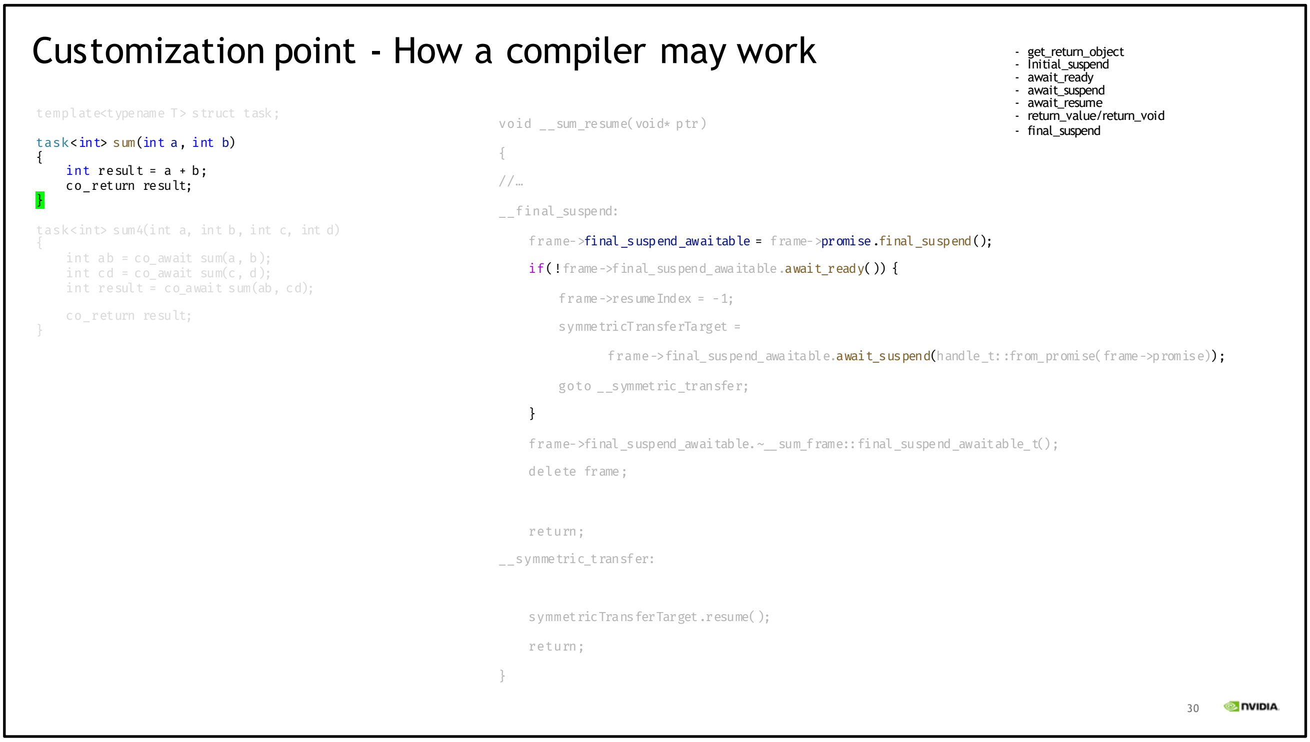
- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__final_suspend:
    frame->final_suspend_awaitable = frame->promise.final_suspend();
    if(!frame->final_suspend_awaitable.await_ready()) {
        frame->resumeIndex = -1;
        symmetricTransferTarget =
              frame->final_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
    frame->final_suspend_awaitable.~__sum_frame::final_suspend_awaitable_t();
    delete frame;

    return;
__symmetric_transfer:

    symmetricTransferTarget.resume();
    return;
}
```

Final suspend, house keeping

The structure of final suspend looks very similar to the initial_suspend.



Again, we first retrieve the final_suspend_awaitable. And then ask the question, are you ready?

If you are not, let’s suspend you.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum_resume(void* ptr)
{
//…
__final_suspend:
    frame->final_suspend_awaitable = frame->promise.final_suspend();
    if(!frame->final_suspend_awaitable.await_ready()) {
        frame->resumeIndex = -1;
        symmetricTransferTarget =
              frame->final_suspend_awaitable.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
    frame->final_suspend_awaitable.~__sum_frame::final_suspend_awaitable_t();
    delete frame;

    return;
__symmetric_transfer:

    symmetricTransferTarget.resume();
    return;
}
```

31

Notice here we have something called symmetric transfer. It is a behavior designed in the spec to immediately resume its parent coroutine. Please refer the spec for more detail. But generally, this is the whole journey of the sum coroutine.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
struct __sum4_frame
{
    // ...
    void(* const resumefn)(void*) = __sum4_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
        int c;
        int d;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;

        struct {
            int ab;
            int cd;

            union {
                task<int> sum_a_b;
                task<int> sum_c_d;
                task<int> sum_ab_cd;
            };
        };
    };
};
```

32

Okay, now that we've walk through the sum coroutine, I think we are more ready to look at the more complicated one, sum4.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
struct __sum4_frame
{
    // ...
    void(* const resumefn)(void*) = __sum4_resume;
    void(* const destroyfn)(void*);

    struct param_t {
        int a;
        int b;
        int c;
        int d;
    } params;

    union {
        initial_suspend_awaitable_t initial_suspend_awaitable;
        final_suspend_awaitable_t final_suspend_awaitable;

        struct {
            int ab;
            int cd;

            union {
                sum_a_b_awaitable sum_a_b;
                sum_c_d awaitable sum_c_d;
                sum_ab_cd_awaitable sum_ab_cd;
            };
        };
    };
};
```

33

For the coroutine frame, we can see that we need to capture some extra information now. The extra two input params and some intermediate variables.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:
    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

    if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

    if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum( frame->ab, frame->cd).operator co_await();

    if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend( handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```


The initial suspend and final suspend are basically the same, so let’s focus on the logic part.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:
    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```


This is what happens when we try to co_await sum(a, b). We store the awaitable returned from operator co_await.

And start to ask question, are you ready. If not, let’s suspend you.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

36

And when we resume, we retrieve the return value. And in that case, it is assigned to variable ab.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

37

Same thing happens after. we co_await on the next one.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

38

And store the return value from await_resume

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

39

And again for another expression.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum( frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend( handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

40

You might find a repeat pattern here.

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:
    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum( frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend( handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

41

It looks familiar right? Similar to initial_suspend and final_suspend. We first retrieve awaitable,

Customization point - How a compiler may work

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```
void __sum4_resume(void* ptr)
{
// ...
__sp1:

    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

     if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

42

and then do suspend or resume.

Because these are what happen for co_await expressions.

And before, what we were doing were co_await initial_suspend_awaitable and co_await final_suspend_awaitable, both are just a special kind of awaitables.

Customization point - How a compiler may work

Several important conclusions from spec

- await_resume returns the result value for co_await expression.
- The `if(!await_ready()) { … await_suspend(); /*suspend*/ } await_resume()` pattern.
- Notice how code are translated, functions for customized points can also be templated.

  > One way to get coroutine_handle<Promise> from coroutine_handle<>
- Notice the timing when awaiters are destructed - destructors are also available for customized behavior

43

Quick recap for several important conclusion here,

- await_resume returns the result value for co_await expression.
- The `if(!await_ready()) { … await_suspend(); /*suspend*/ } await_resume()` pattern.
- Remember for those await_suspend and we pass in the actual handle, we can have templated await_suspend function so that we can access the underlying promise object.
- And finally, since all those things are structs, dctors are available for customization.

JOB SYSTEM

- Scheduler + User defined workload
- Optimize for CPU throughput

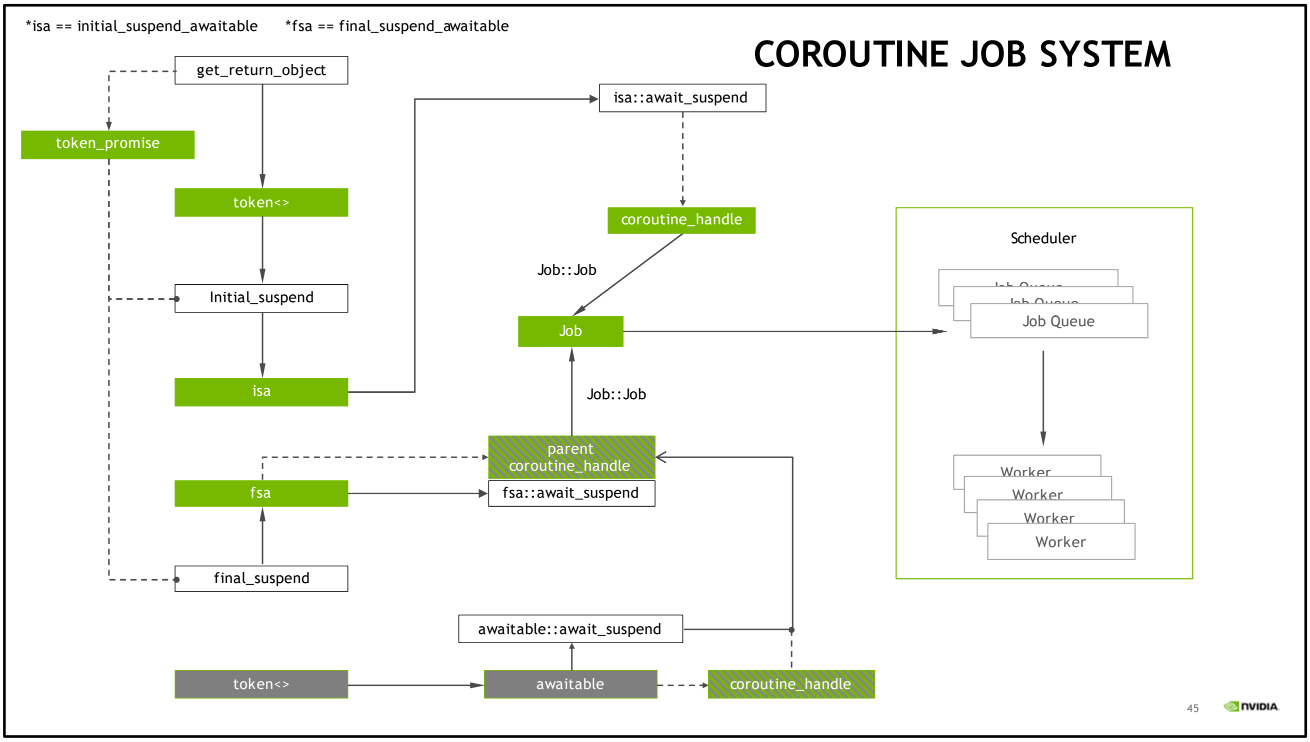
44

Okay, finally, we are ready to talk about the job system!

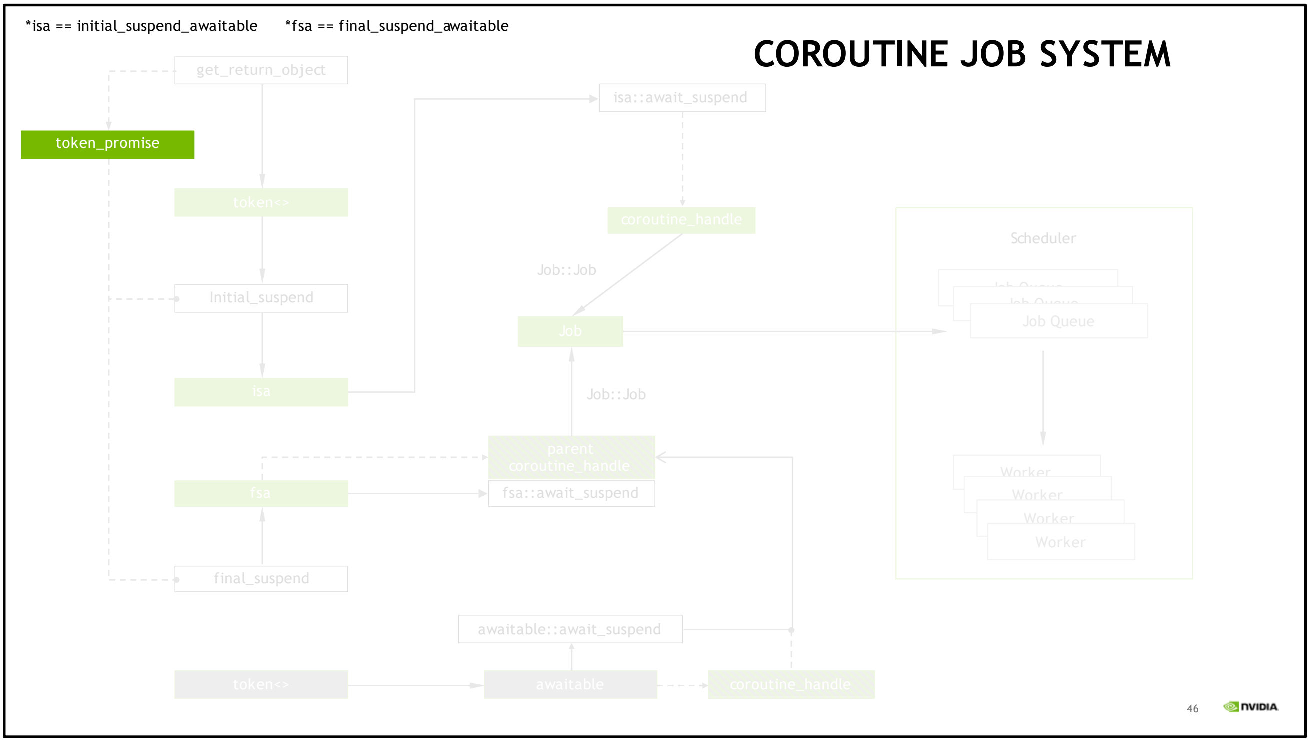
What is a job system? Job system is a kind of task system. Usually user will define a bunch of workload and submit them to the system. The system will pick them up and execute them according to certain dependency order on different threads or fibers etc.

It's also a general practice in game engine to have dedicated threads to handle specialized job types. For example to have an dedicated IO thread to avoid fighting over the IO access and a network thread to only handle network traffic.

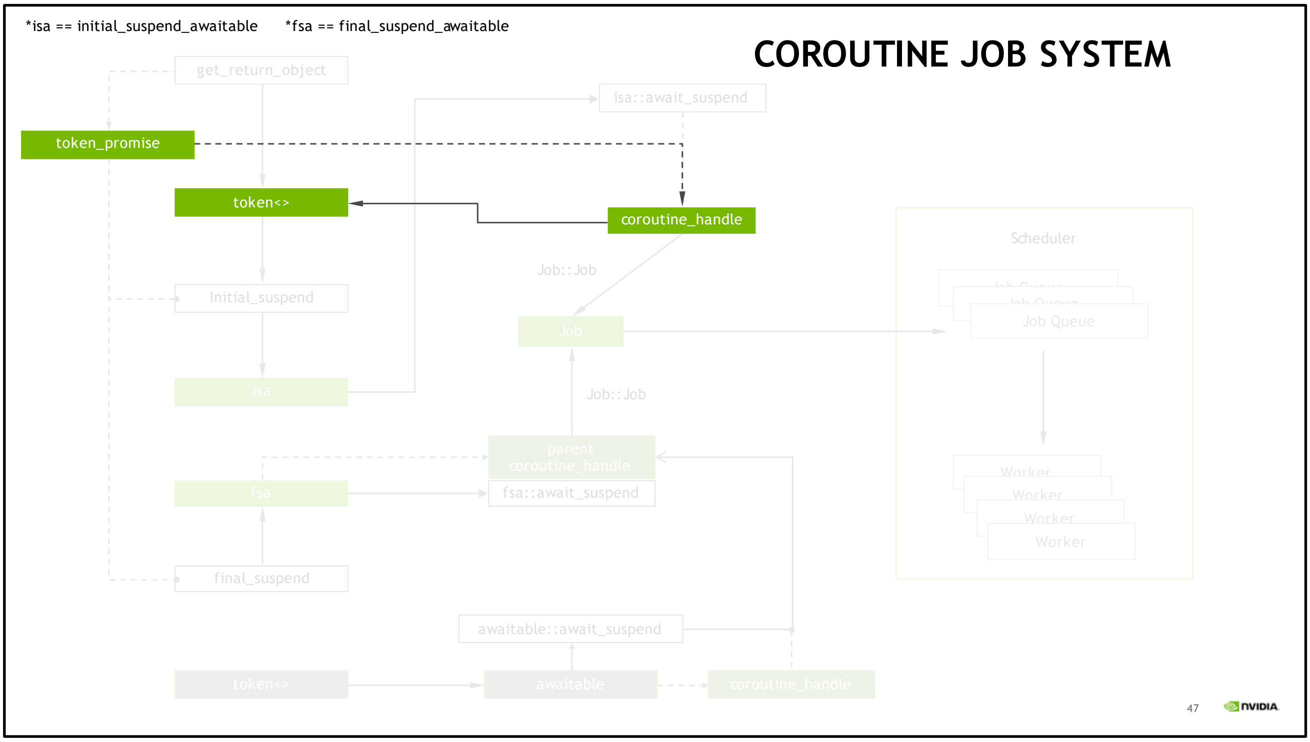
One of the major goal of a job system is to have bigger overall CPU throughput, which means ideally, I will want all threads be busy and have things to do all the times, and we want to avoid bubbles, which is spared time frames on the CPU timeline.



And voila! This is our system.

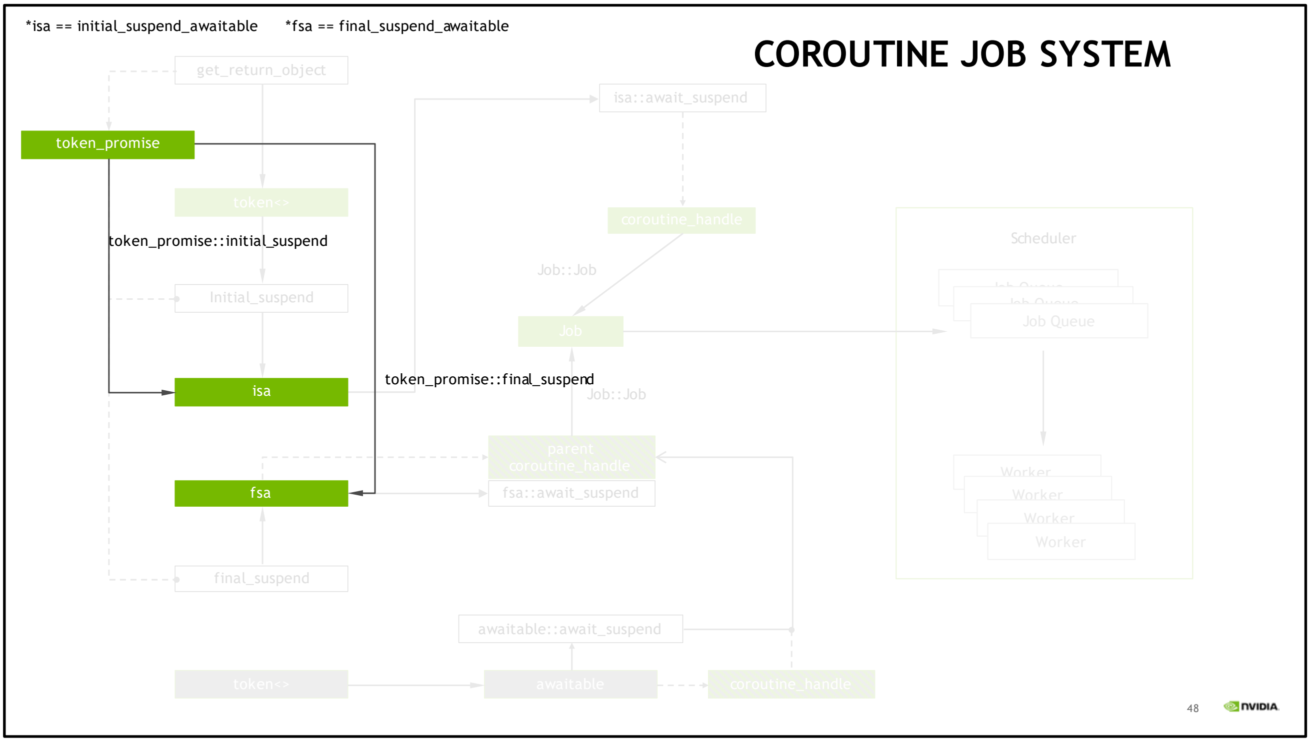


Token promise is the main piece that glues our system and coroutine semantics together.

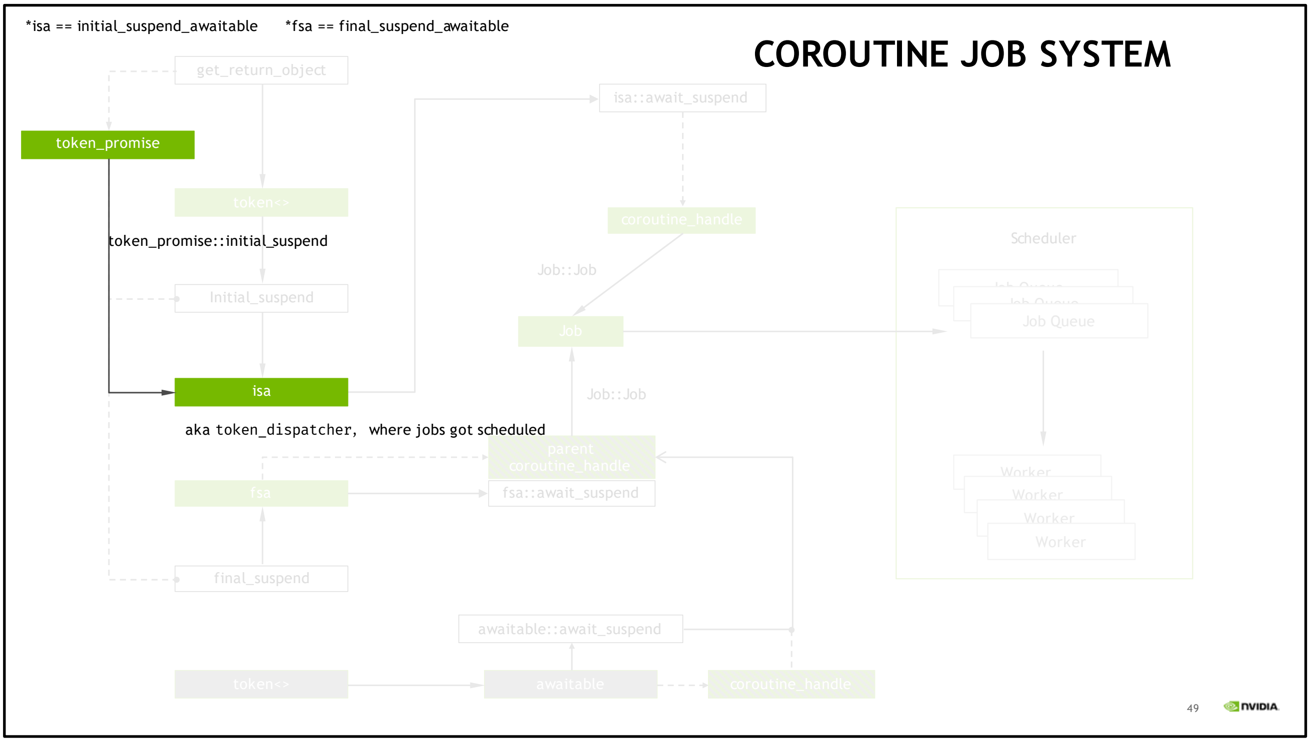


Token is the task object type in the system. The reason is because I used task for a subset of tokens that have extra features, we will see later.

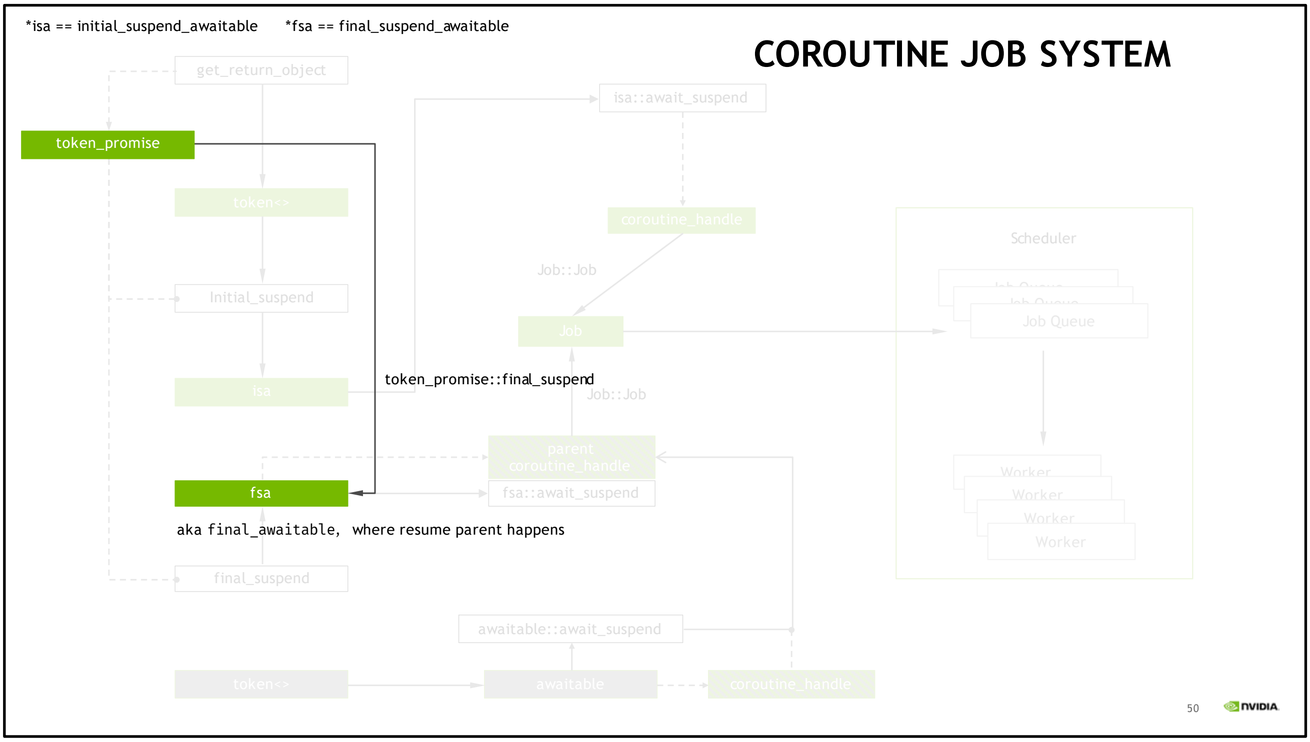
And a token holds a coroutine handle for convenient coroutine access.



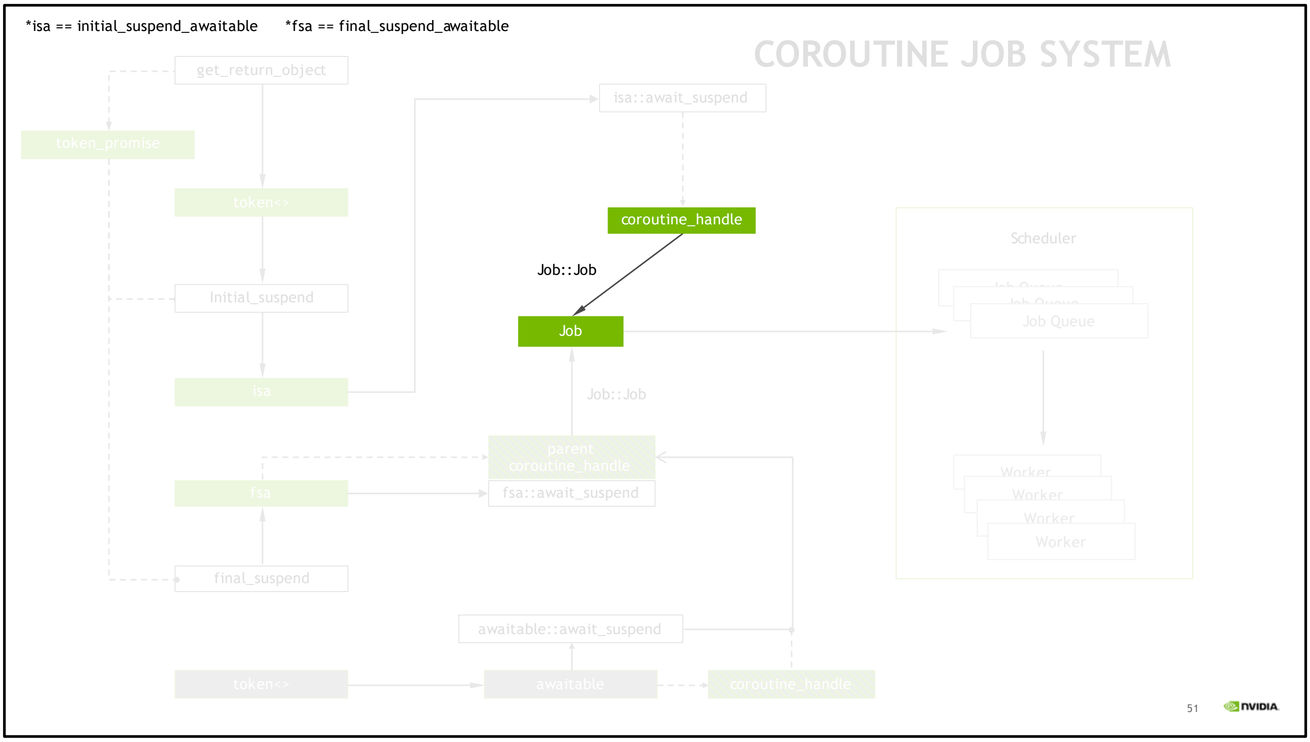
And there are initial_suspend_awaitable and final_suspend_awaitable. As we mentioned before, they are from the initial_suspend method and final_suspend method.



In our system, initial_suspend_awaitable is called token_dispatcher. We customize this type to dispatch our jobs.



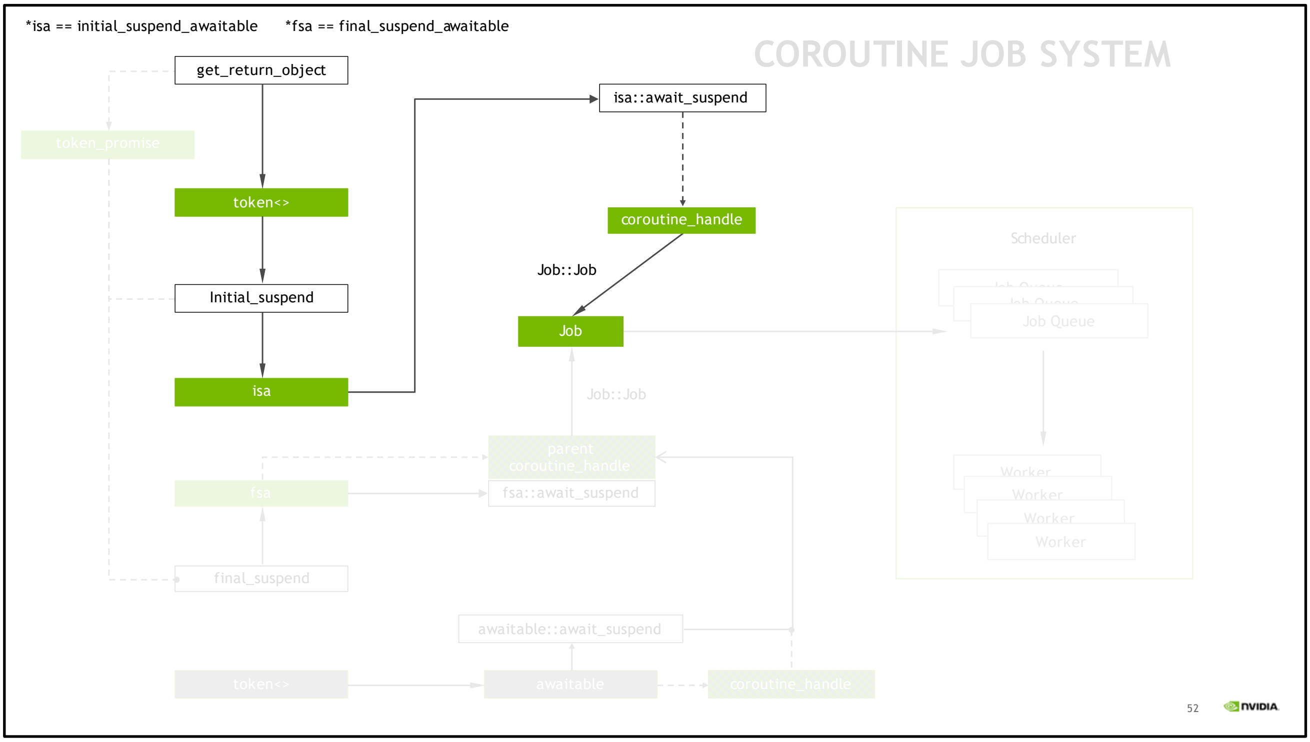
And the finial_suspend_awaitable is called final_awaitable, this is where resume parent happens. We will get back to the concept of resuming parent in a second.



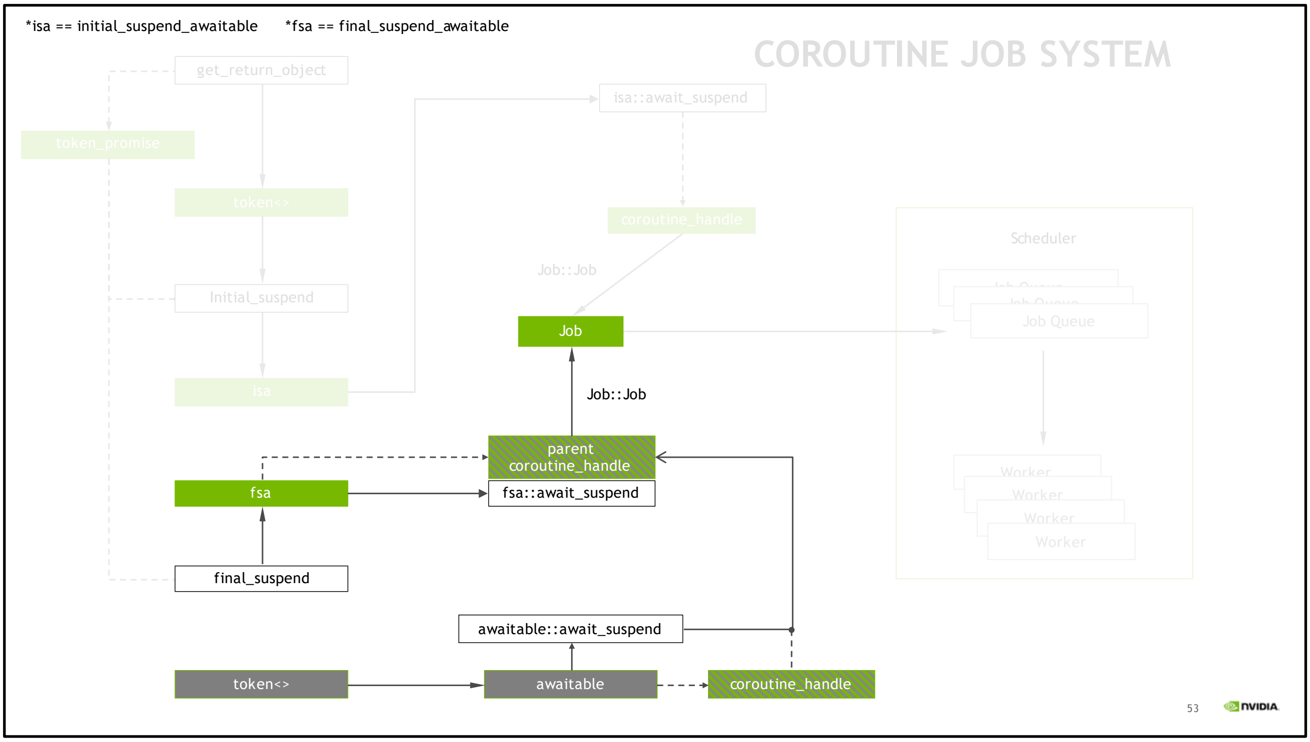
And of course, the heart of the system is a job. Scheduler don't interact with coroutine directly, that's how we can implement a scheduler in whatever way we want, which is not included in today's talk btw. And potentially we can even handle both coroutine and non-coroutine type of jobs.

But anyway, here, the job is a very lightweight object, it mainly stores a coroutine handle.

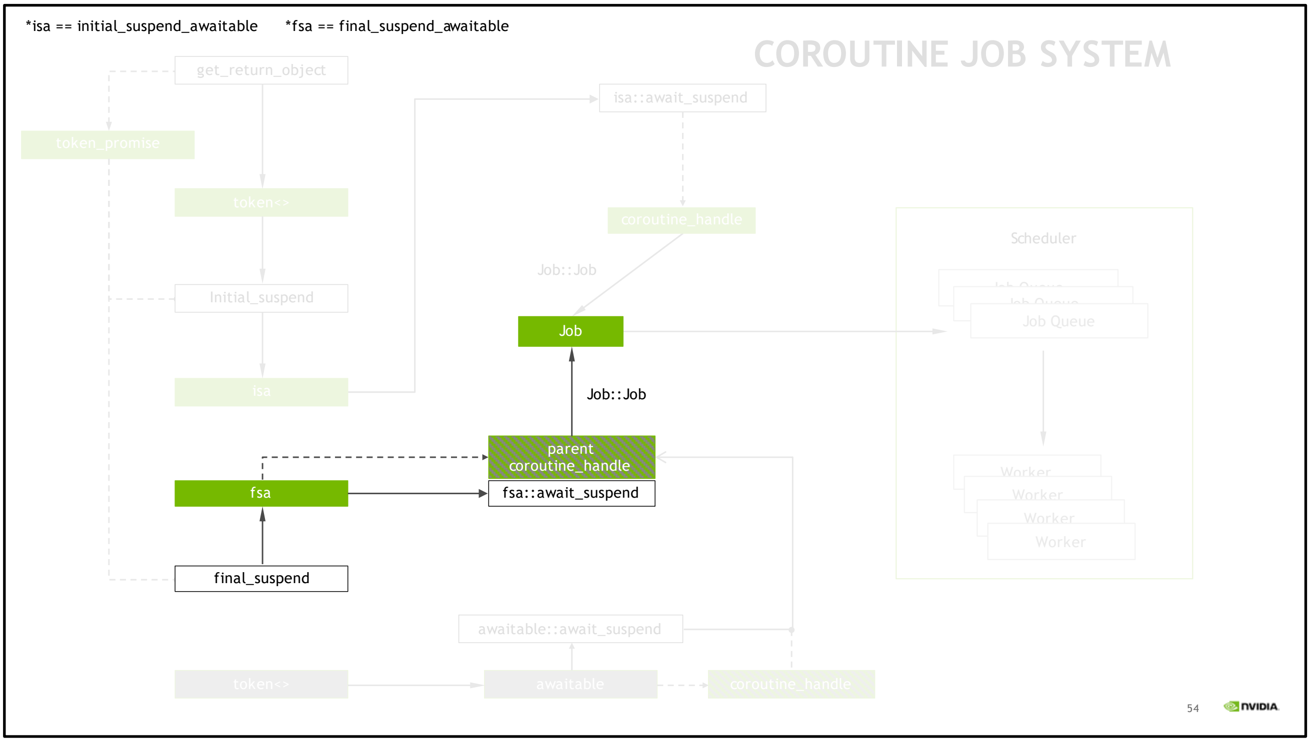
And we will have two ways to construct and dispatch a job.



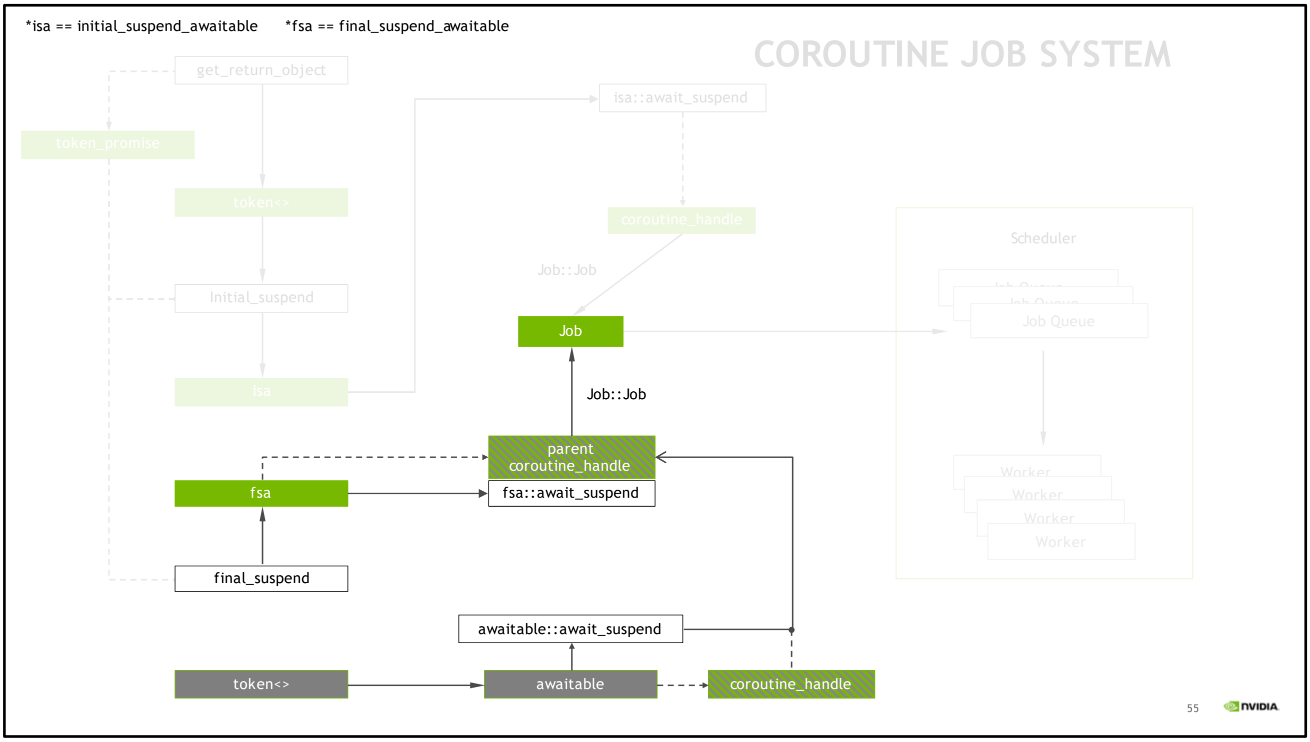
The first path is when we invoke a new coroutine. It happens, again, in the initial suspend awaitable, aka, token dispatcher. When we invoke await_suspend.



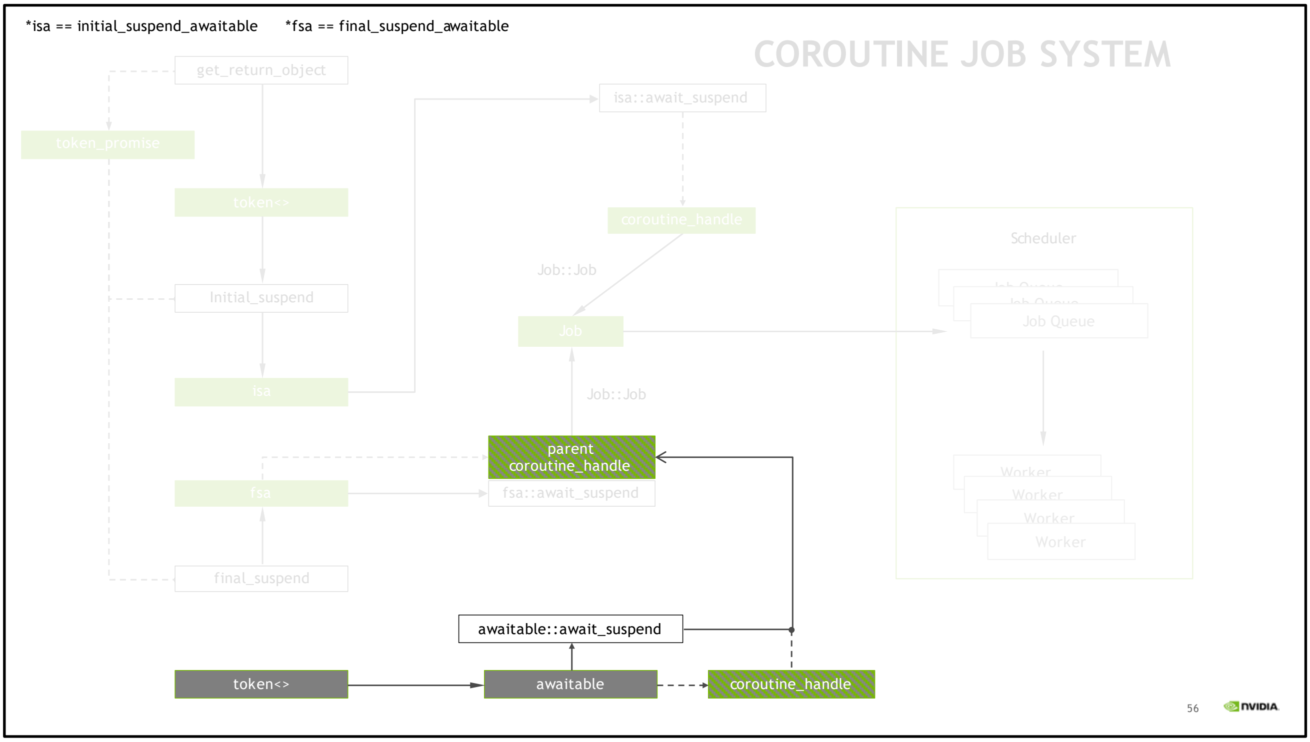
And the second is when we finish a coroutine and we need to resume the parent that was suspended.



That happens in the final_suspend_awaitable, when we finish the execution of the current coroutine. And we reschedule the suspended parent.

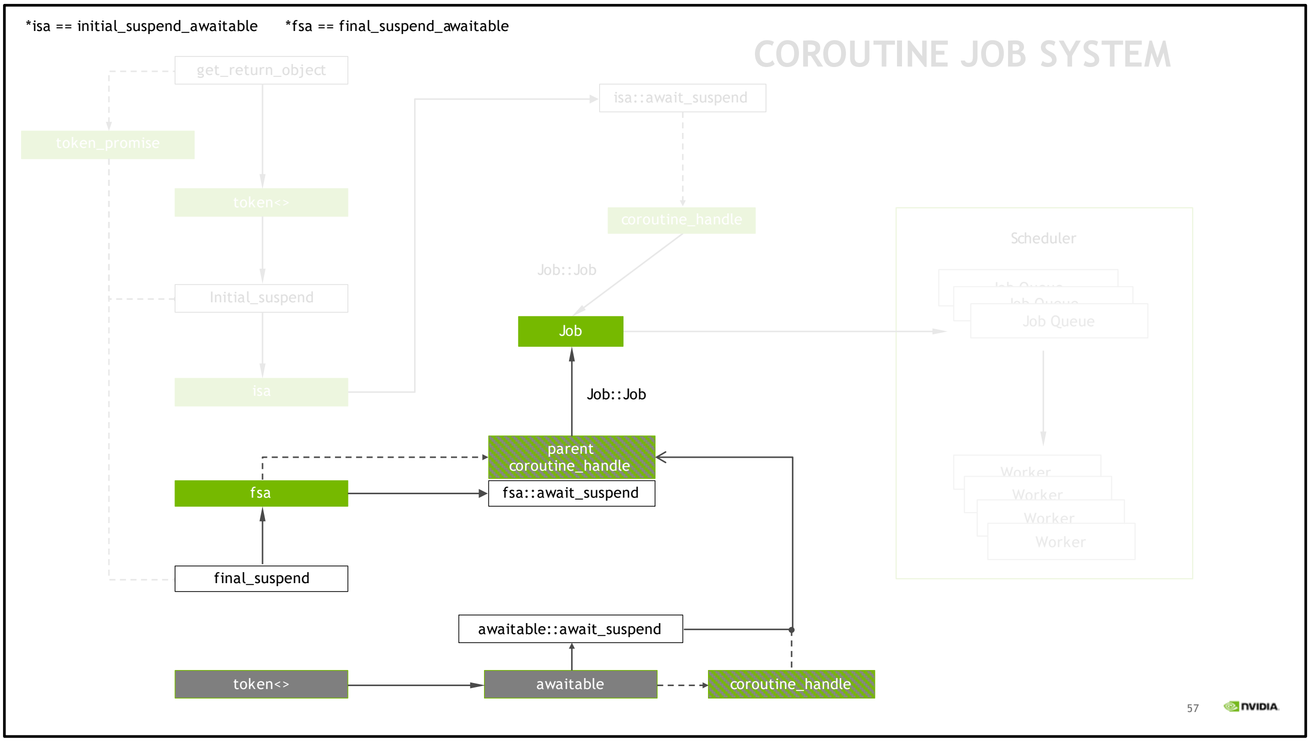


But then it’s natural to have the question – where is the parent from.



Well, that’s from when we call co_await on a token object, and we ask the question are you ready. If not, okay, we gonna suspend you in the await_suspend, where we will have the handle, and we will suspend the parent coroutine.

Notice this patterned background color, that means these two are the same object. And we store that so that later we can resume it.



Combine them together, we have the second path. And is the core how dependencies are handled in this system.

GOAL

```
auto saveWorld = ApplyImmunization( 3000 ); // 3000 healthy people left on earth :)
```


As we all know, we have a pandemic going on, So let’s say, for example, we gonna end this pandemic by offering everyone vaccine and save the world.

GOAL

```
void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 ); // 3000 healthy people left on earth :)
    saveWorld.Wait();
}
```

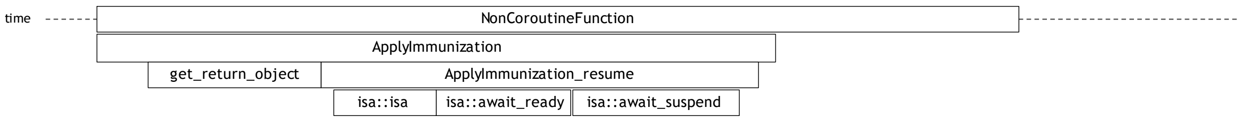

Well, technically we can't pause the time and say hold on, no more time moving forward before we end this pandemic, but for simplicity, let's say we gotta wait before we do anything else.

token<void>

```
token<> ApplyImmunization(uint  healthPeople)
{
   //......
}
```


So what we need to do? Well, if we need to do something, we create a function and hope that function will do what we want.

But because this is a coroutine talk, we create a coroutine.

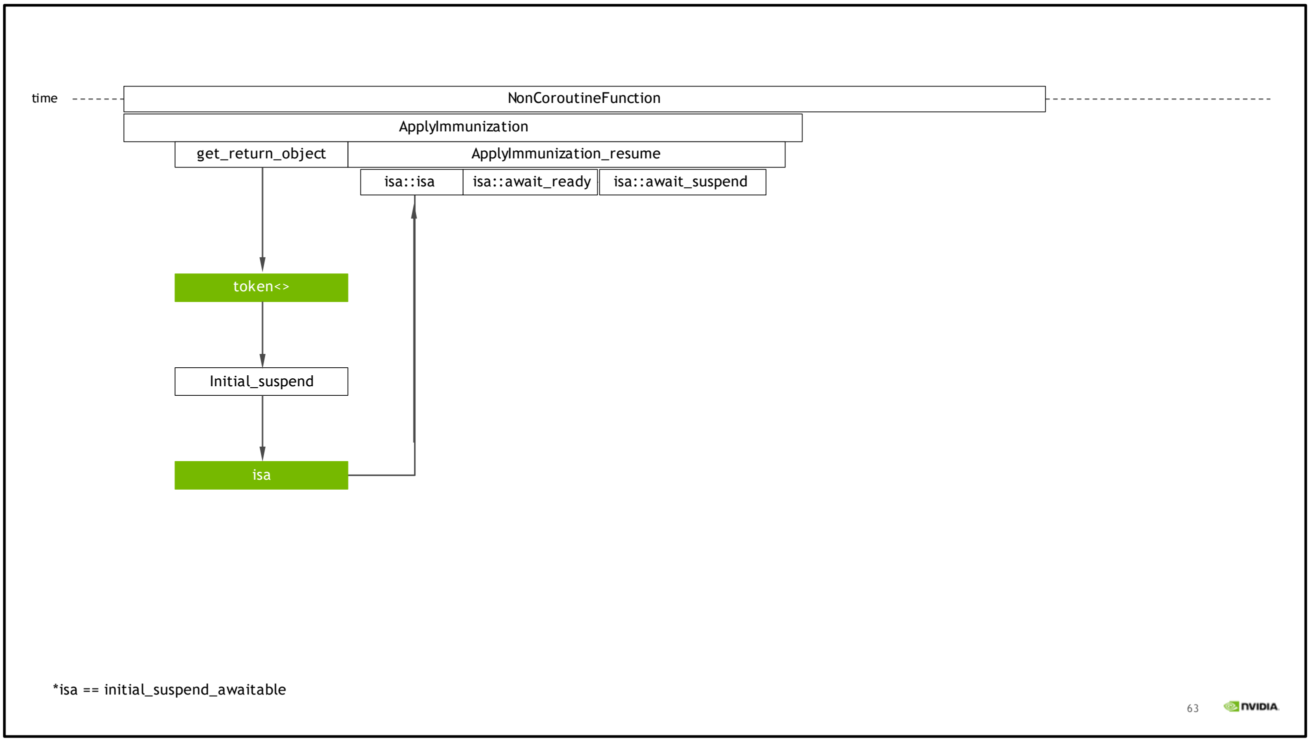


*isa == initial_suspend_awaitable

61

Using what we discussed in the previous 60 slides, we can have this timeline.

So when we call into the coroutine, again, we first construct the coroutine frame, which is not shown here. And then we call get_return_object to get the token object. Then finally, we call into the state machine, call initial_suspend, and ask are you ready, in this system, the answer will always be no, which we will see in a second. And we suspend.



That’s how this small piece of the system fit into the picture.

token<void>

```
class token_promise
{
    token get_return_object();
}

class token
{
public:
    using promise_type  = token_promise;
    using coro_handle_t  = std::experimental::coroutine_handle<promise_type>;

    token(coro_handle_t  handle);

protected:
    coro_handle_t  mHandle;
}

token token_promise::get_return_object()
{
    token::coro_handle_t  handle  = std::experimental::coroutine_handle<token_promise>::from_promise(  *this  );
    return  { handle  };
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```


Let's see some code. Notice from now on, all coroutine customization functions are in snake_case, and all function I defined in the system are in BigCamelCase.

As we know we will have those two structs, token_promise and token.

So first, recall the translated code we were discussing before, the first customization point we need to handle is provide a way to return a token object, which is the get_return_object function here.

token<void>

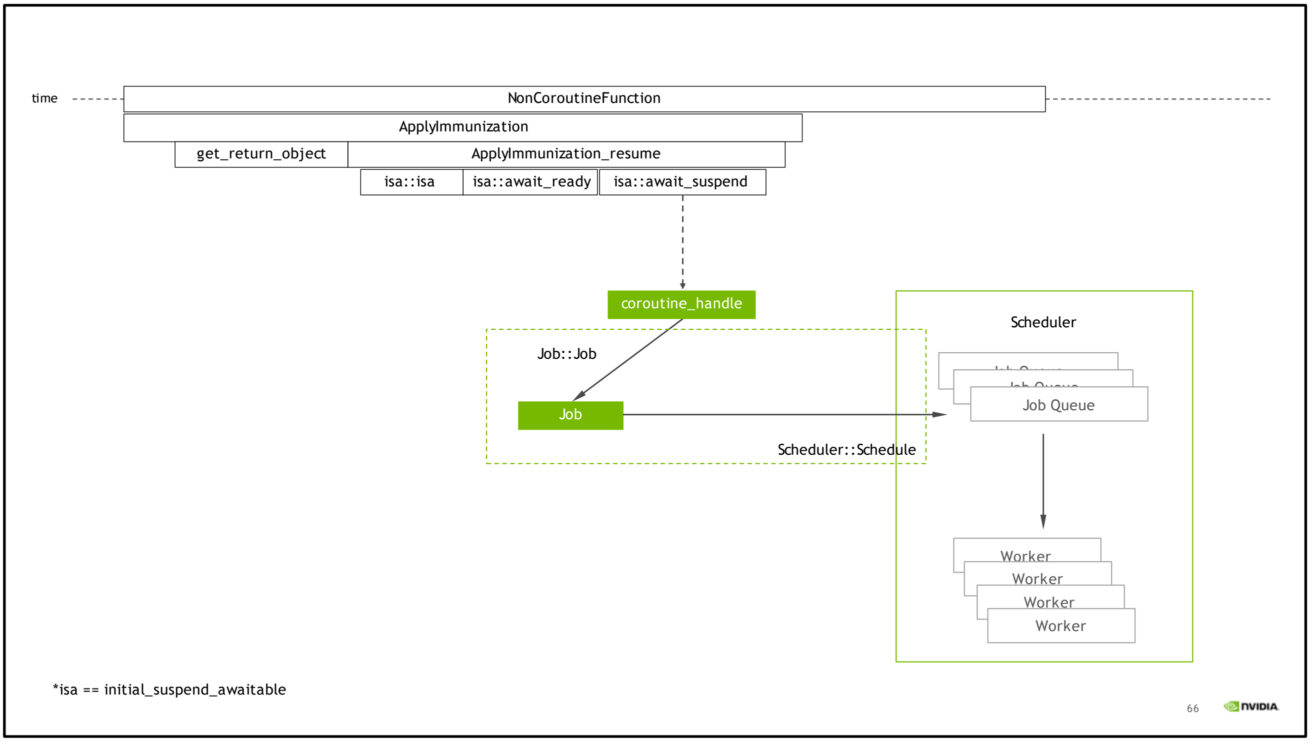
```
class token_promise
{
    token get_return_object();
}


class token
{
public:
    using promise_type  = token_promise;
    using coro_handle_t  = std::experimental::coroutine_handle<promise_type>;

    token(coro_handle_t  handle);

protected:
    coro_handle_t  mHandle;
}



token token_promise::get_return_object()
{
    token::coro_handle_t  handle  = token::coro_handle_t::from_promise(  *this  );
    return { handle  };
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```

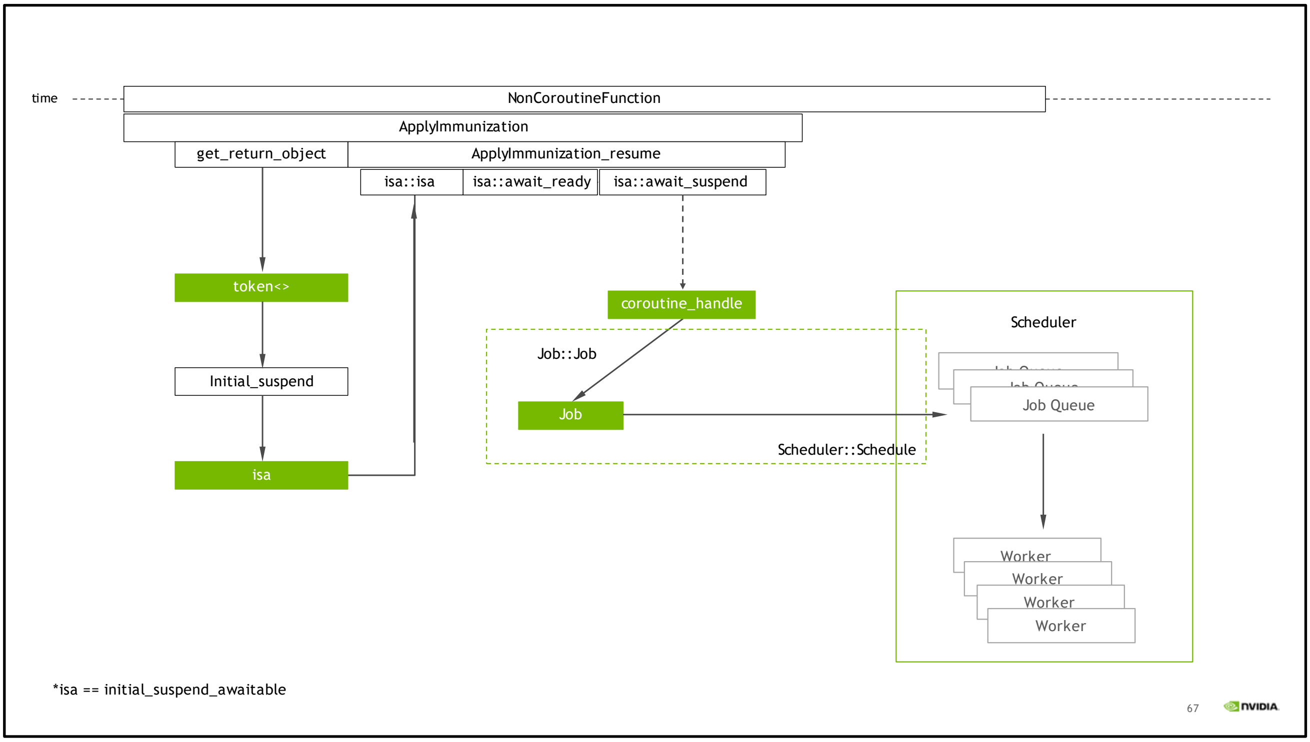
65

It’s a little bit too long, so let’s shorten it a little.

from_promise is the function we want to use here, this is the case we want to retrieve a coroutine handle from promise.



And on the other hand, after we have the coroutine handle, we can schedule the job.



And combine them together. This is what happened, the bars in the top represent the timeline and this shows how our system interact in the code when we invoke the coroutine.

token_dispatcher

```
struct token_dispatcher;
class token_promise
{
    token get_return_object();
    token_dispatcher  inital_suspend()
    {
        auto& scheduler  = Scheduler::Get();
        bool isOnMainThread  =
            scheduler.GetMainThreadIndex()  == scheduler.GetThreadIndex();
        return token_dispatcher{  isOnMainThread  };
    }
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```


I still have not explain what happens in the initial_suspend_awaitable, or token_dispatcher. All magic happens in the initial_suspend.

Notice what we pass in here, isOnMainThread.

token_dispatcher

```
template<typename Promise>
void Scheduler::Schedule( const std::experimental::coroutine_handle<Promise>& handle );

struct token_dispatcher // ret of initial_suspend for token_promise
{
    bool shouldSuspend;
    token_dispatcher(bool shouldSuspend): shouldSuspend(shouldSuspend) {}
    bool await_ready() { return false; }
    void await_resume() {}
    bool await_suspend(coroutine_handle<> handle)
    {
        using namespace std::experimental;
        coroutine_handle<token_promise> realHandle = coroutine_handle<token_promise>::from_address( handle.address() );
        mSuspendedPromise = &realHandle.promise();
        ENSURES( mSuspendedPromise != nullptr );

        if( shouldSuspend ) {
            Scheduler::Get().Schedule( realHandle );
        }

        return shouldSuspend;
    }

protected:
    token_promise* mSuspendedPromise = nullptr;
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```


And when we construct the token_dispatcher, we store that to control whether we want to suspend it.

We do it in the await_suspend instead of await_ready because we want the access to the promise object. And according to the spec, when we return a bool for await_suspend, if it’s true, that means the coroutine is suspended, if it is false, that means we do not suspend it and it will be resumed immediately.

So here, you can notice how we use that “isOnMainThread” is to treat that as a switch to decide whether we need to actually schedule it. The idea is, if it’s on a worker thread already, we don’t need to schedule it and it’s safe to directly run it.

token_dispatcher

```
template<typename Promise>
void Scheduler::Schedule( const std::experimental::coroutine_handle<Promise>& handle );

struct token_dispatcher // ret of initial_suspend for token_promise
{
    bool shouldSuspend;
    token_dispatcher(bool shouldSuspend): shouldSuspend(shouldSuspend) {}
    bool await_ready() { return false; }
    void await_resume() {}
    bool await_suspend(coroutine_handle<> handle)
    {
        using namespace std::experimental;
        coroutine_handle<token_promise> realHandle = coroutine_handle<token_promise>::from_address( handle.address() );
        mSuspendedPromise = &realHandle.promise();
        ENSURES( mSuspendedPromise != nullptr );

        if( shouldSuspend ) {
            Scheduler::Get().Schedule( realHandle );
        }

        return shouldSuspend;
    }

protected:
    token_promise* mSuspendedPromise = nullptr;
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```

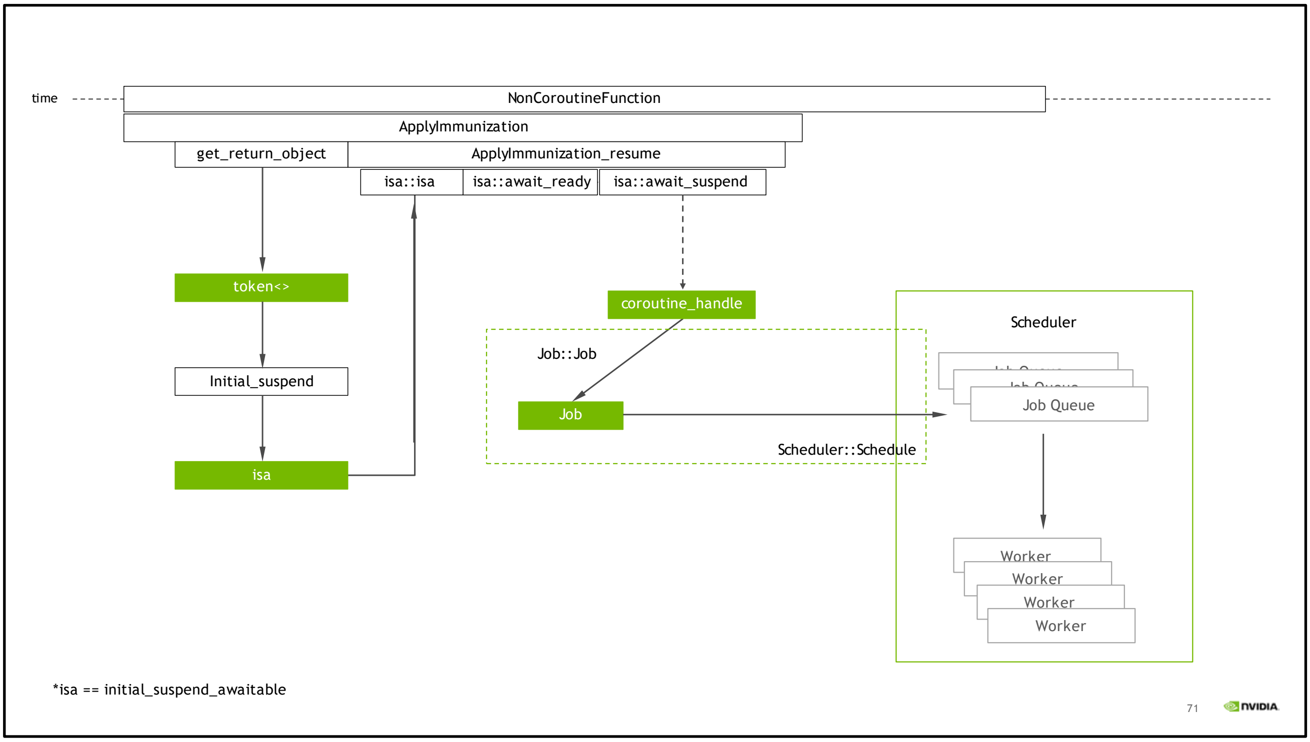
Wrong in general but works for this case:

This is in inital_suspend, so we know exactly what we are dealing with

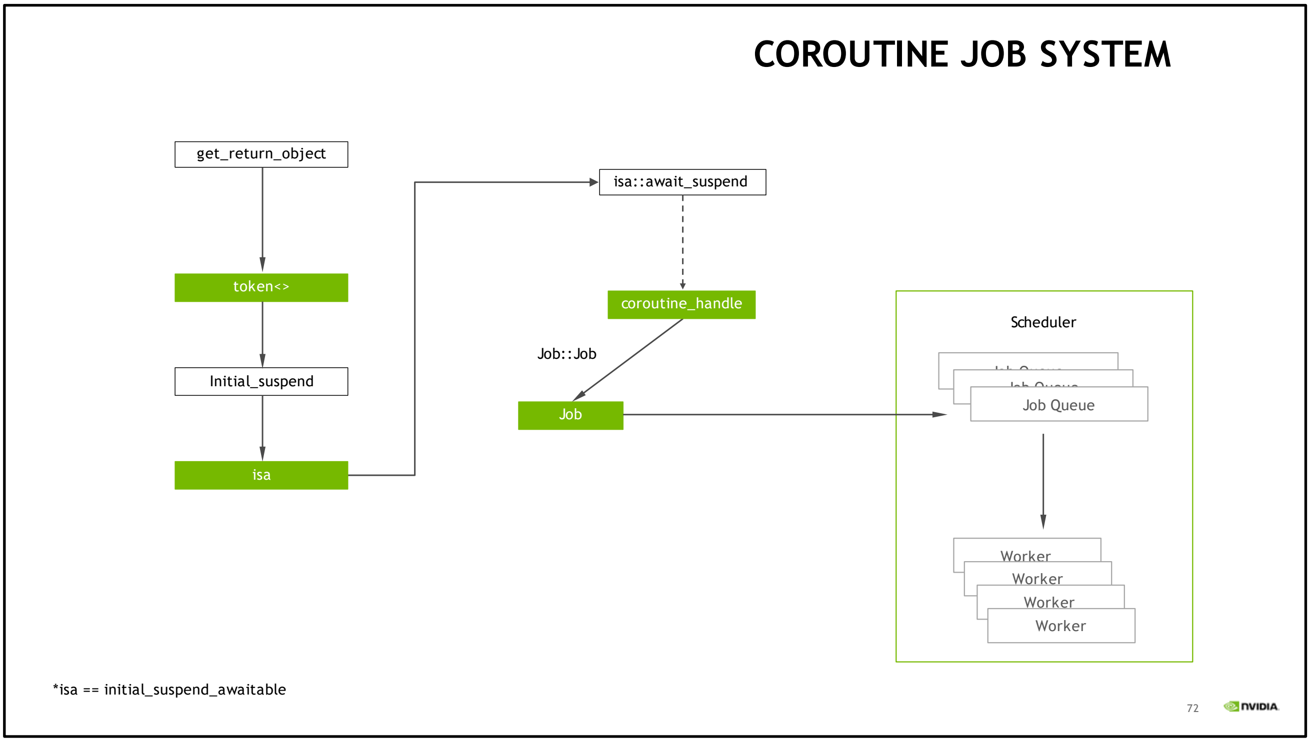
70

Another thing worth notice is this line is generally wrong, except for several special cases. For this one, because it’s in initial_suspend, we know the exact type of coroutine handle passed into the function.

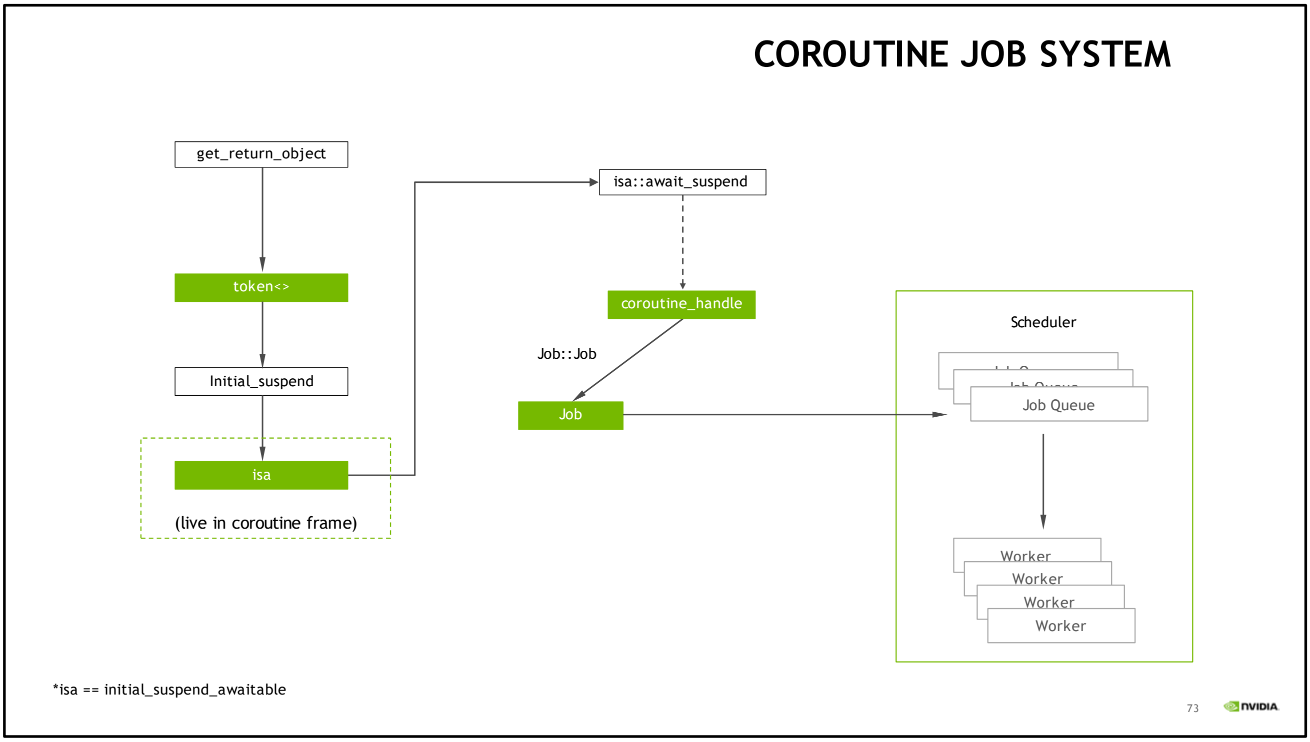
So we do not need to template the whole function, which further slows down the compilation time.



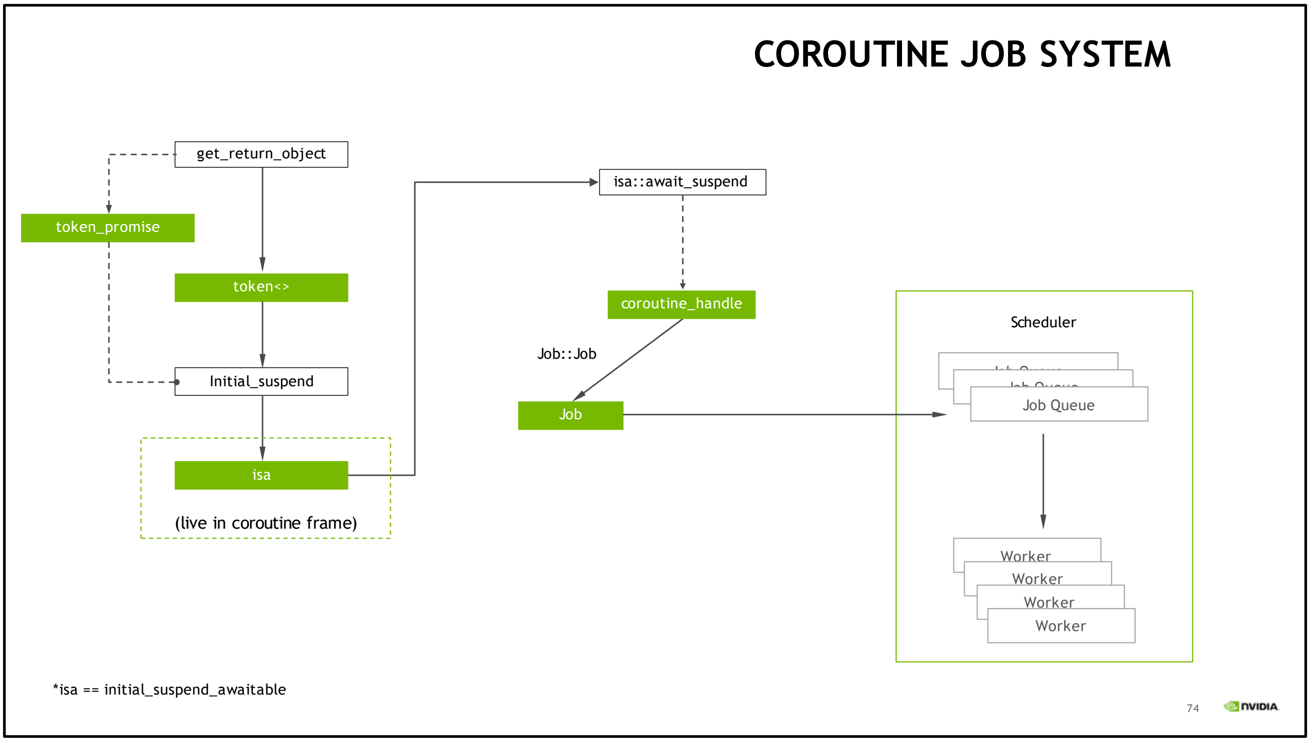
So through explaining how `ApplyImmunization` is invoked,



We derived this part of the system.



And as we've discussed, token dispatcher, which is the initial suspend awaitable, lives in coroutine frame.



And promise object will be passed into get_return_object.
And initial_suspend is customization point in the promise.

token<void>

```
struct token_dispatcher;
struct final_awaitable;
class token_promise
{
    token get_return_object();
    token_dispatcher inital_suspend()
    {
        auto& scheduler = Scheduler::Get();
        bool isOnMainThread =
            scheduler.GetMainThreadIndex() == scheduler.GetThreadIndex();
        return token_dispatcher{ isOnMainThread };
    }
    final_awaitable final_suspend() { return {}; }
}
```

```
token<> ApplyImmunization(uint healthPeople)
{
    //......
}
```

75

Until this point, if we compile the code, we will have compile error, because we need to define two extra things.

The first one is final_awaitable, which we will discuss later.

token<void>

```
struct token_dispatcher;
struct final_awaitable;
class token_promise
{
    token get_return_object();
    token_dispatcher inital_suspend()
    {
        auto& scheduler = Scheduler::Get();
        bool isOnMainThread =
            scheduler.GetMainThreadIndex() == scheduler.GetThreadIndex();
        return token_dispatcher{ isOnMainThread };
    }
    final_awaitable final_suspend() { return {}; }
    void return_void() {}
}
```

```
token<> ApplyImmunization(uint healthPeople)
{
    //......
}
```

That’s it!

And another one, because we do not return any actual value here, so return void. And that’s it! We just launch a job to save the world! And hope it will be fast.

token<T>

```
template<typename  T>
class token
{
public:
    using promise_type  = token_promise<T>;
    using coro_handle_t
          = std::experimental::coroutine_handle<promise_type>;

    token(coro_handle_t  handle);

protected:
    coro_handle_t  mHandle;
}
```


At this point, I want to discuss how should we template the token so that it can carry out result.

First step, of course, we template everything.

token<T>

```
template<typename  T>
class  token_promise
{
    token<T>  get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    void  return_value(  VALUE&&  v );
}
```

```
class  token_promise
{
    token  get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    void  return_void()  {}
}
```


And instead of return void, we will need to actually return a value

token<T>

```
template<typename  T>
class token_promise
{
    token<T> get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    template<typename  VALUE> void return_value(  VALUE&& v )
    {
        value = v;
    }
    T value;
}
```

```
class token_promise
{
    token get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    void return_void()  {}
}
```


And cache off the return value for future use. In practice, we will need to put certain constraint on this value type to limit what user can pass into for co_return

token<T>

```
template<typename  T>
class token_promise
{
    token<T> get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    template<typename  VALUE> void return_value(  VALUE&& v )
    {
        value = v;
    }
    T value;
}
```

How should user retrieve the result?

1. Our own way outside the coroutine: de-coroutine
2. co_await under coroutine context

We will have ways in the system to retrieve this result. The first one is co_await, as we've seen before.

At the same time, we still need a way to get the result without turning a function into coroutine, which I called de-coroutine.

“de-coroutine”

```
template<typename  T>
class token_promise
{
    token<T> get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    template<typename  VALUE> void return_value(  VALUE&& v )
    {
        value = v;
        if(futurePtr)  {
            futurePtr->Set(  std::forward<T>(v)  );
        }
    }
    T value;
    future<T>*  futurePtr  = nullptr;
}
```

81

To do that, we need to provide an extra side channel, or say back door… We store a future pointer.

Notice here, future is not the std::future, std::future has too much overhead for our usage here, this is a utility type provided in the system.

future

```
template<typename T>
class future
{
public:
   future(): mSetEvent( 1 ) { }
   void Set( T&& v )
   {
      value = v;
      mSetEvent.decrement();
   }

   bool IsReady() { return mSetEvent.IsReady(); }

   const T& Get() const
   {
      mSetEvent.Wait();
      return value;
   }


protected:
   T value = {};
   mutable single_consumer_counter_event mSetEvent;
};
```

```
__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();
```


How it is implemented is pretty straightforward. It internally cache off the value. And here you can see we have this single consumer counter event, implemented based on the Operating system API. In this case, I use

WaitForSingleObject on windows, you can also use waitOnAddress etc.

future

```cpp
__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();
```

```cpp
template<typename T>
class future
{
public:
   future(): mSetEvent( 1 ) { }
   void Set( T&& v )
   {
      EXPECTS( !IsReady() );
      value = v;
      mSetEvent.decrement();
   }

   bool IsReady() { return mSetEvent.IsReady(); }

   const T& Get() const
   {
      mSetEvent.Wait();
      return value;
   }

protected:
   T value = {};
   mutable single_consumer_counter_event mSetEvent;
};
```

```cpp
void single_consumer_counter_event::Wait()
{
   Scheduler::Get().RegisterAsTempWorker( mEvent );
}
```


When the caller trying to ask for the result, it register itself as temporary work for the scheduler and unregister itself when the event get triggered.

“de-coroutine”

```
template<typename  T>
class token_promise
{
    token<T>  get_return_object();
    token_dispatcher  inital_suspend();
    final_awaitable  final_suspend()  { return  {}; }
    template<typename  VALUE>  void return_value(  VALUE&&  v )
    {
        value  =  v;
        if(futurePtr)  {
            futurePtr->Set(  std::forward<T>(v)  );
        }
    }
    T value;
    future<T>*  futurePtr  =  nullptr;
}
```

```
template<typename  T>
class token
{
public:
    using promise_type  = token_promise<T>;
    using coro_handle_t  = (same  thing,  too long to fit)

    token(coro_handle_t  handle);
    token(token&)  noexcept  = delete;
    const  T&  Result()
    {
        return  mFuture.Get();
    }
protected:
    coro_handle_t  mHandle;
    future<T>  mFuture;
}
```

84

Where that pointer point to is the future we store in the token. And outside the coroutine context, we can call this result function to retrieve the result.

co_await token<T>

```
template<typename  T> class  token
{
auto  operator  co_await()  const
{
     struct  awaitable:  awaitable_base
     {
          using  awaitable_base::awaitable_base;

          decltype(auto)   await_resume()
          {
               auto& coro  = this->coroutine;
               using  ret_t  = decltype(coro.promise().result());
               if  constexpr  (std::is_void_v<ret_t>)  {
                       return;
               } else  {
                       return  coro ? T{}  :  coro.promise().Result();
               }

          }
     };
     return  awaitable{  mHandle  };
}
}
```

```
__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();
```

85

For co_await, it’s more straightforward, what we need to do is follow the spec, and return the value in await_resume. Because as we mentioned before, the return value of the await_resume will be used as the result of the co_await expression.

GOAL

```
token<> ApplyImmunization(uint healthPeople)
{









}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```


Go back to our save world mission, we all know an empty function won’t work. We need to do something real.

GOAL

```
token<> ApplyImmunization(uint healthPeople)
{
    co_await sequential_for( saveWorldSteps );
}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```


We take steps one by one. I will explain the detail of sequential_for, but for now, this is just another coroutine, which does something.

Customization point - How compiler works

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

```cpp
template<typename T> struct task;

task<int> sum(int a, int b)
{
    int result = a + b;
    co_return result;
}

task<int> sum4(int a, int b, int c, int d)
{
    int ab = co_await sum(a, b);
    int cd = co_await sum(c, d);
    int result = co_await sum(ab, cd);

    co_return result;
}
```

```cpp
void __sum4_resume(void* ptr)
{
// ... initial_suspend
__sp1:
    frame->sum_a_b = sum(frame->params.a, frame->params.b).operator co_await();

    if (!frame->sum_a_b.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->sum_a_b.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp2:
    frame->ab = frame->sum_a_b.await_resume();
    frame->sum_a_b.~();

    frame->sum_c_d = sum(frame->params.c, frame->params.d).operator co_await();

     if (!frame->sum_c_d.await_ready()) {
        frame->resumeIndex = 3;
        symmetricTransferTarget = frame->sum_c_d.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }
__sp3:
    frame->cd = frame->sum_c_d.await_resume();
    frame->sum_c_d.~();

    frame->sum_ab_cd = sum(frame->ab, frame->cd).operator co_await();

     if (!frame->sum_ab_cd.await_ready()) {
        frame->resumeIndex = 4;
        symmetricTransferTarget = frame->sum_ab_cd.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

__sp4:
    // ...
    goto __final_suspend;

// ...
}
```

88

And recall this slides, similar thing happens again! We are awaiting on something. And we invoke the await_suspend

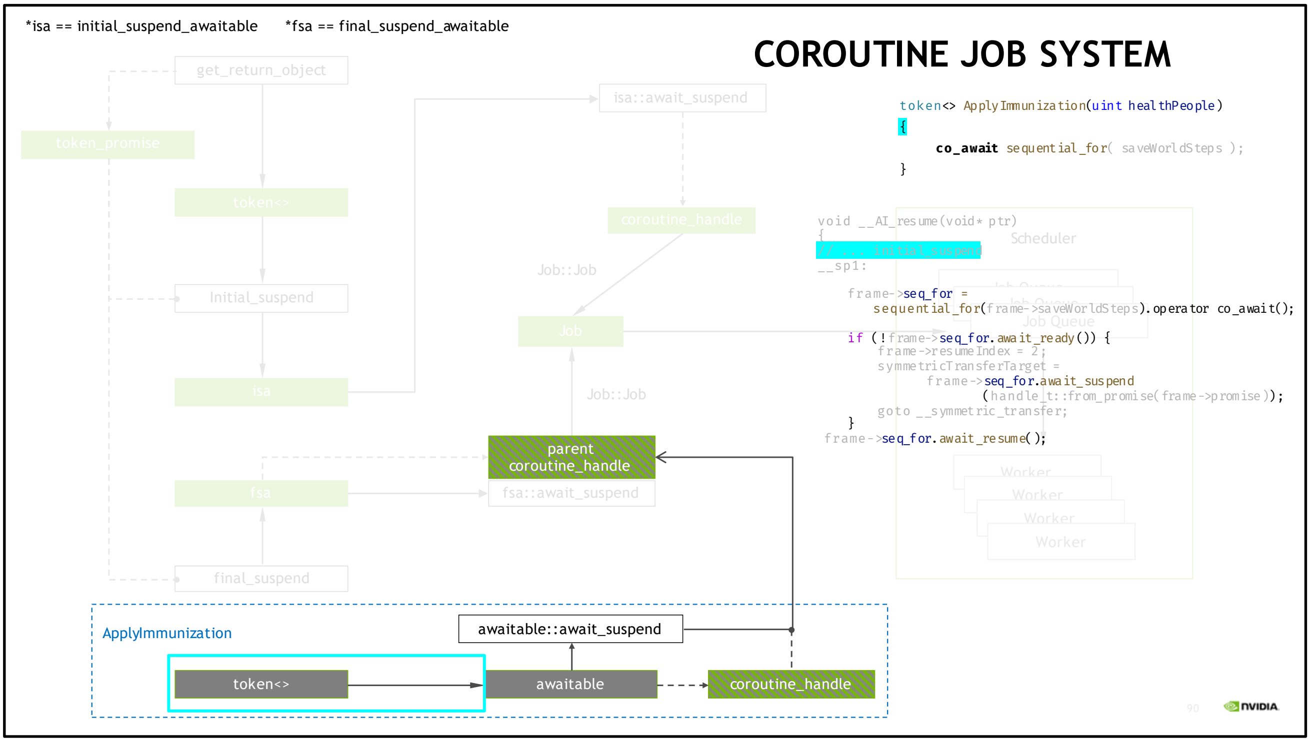
GOAL

- get_return_object
- Initial_suspend
- await_ready
- await_suspend
- await_resume
- return_value/return_void
- final_suspend

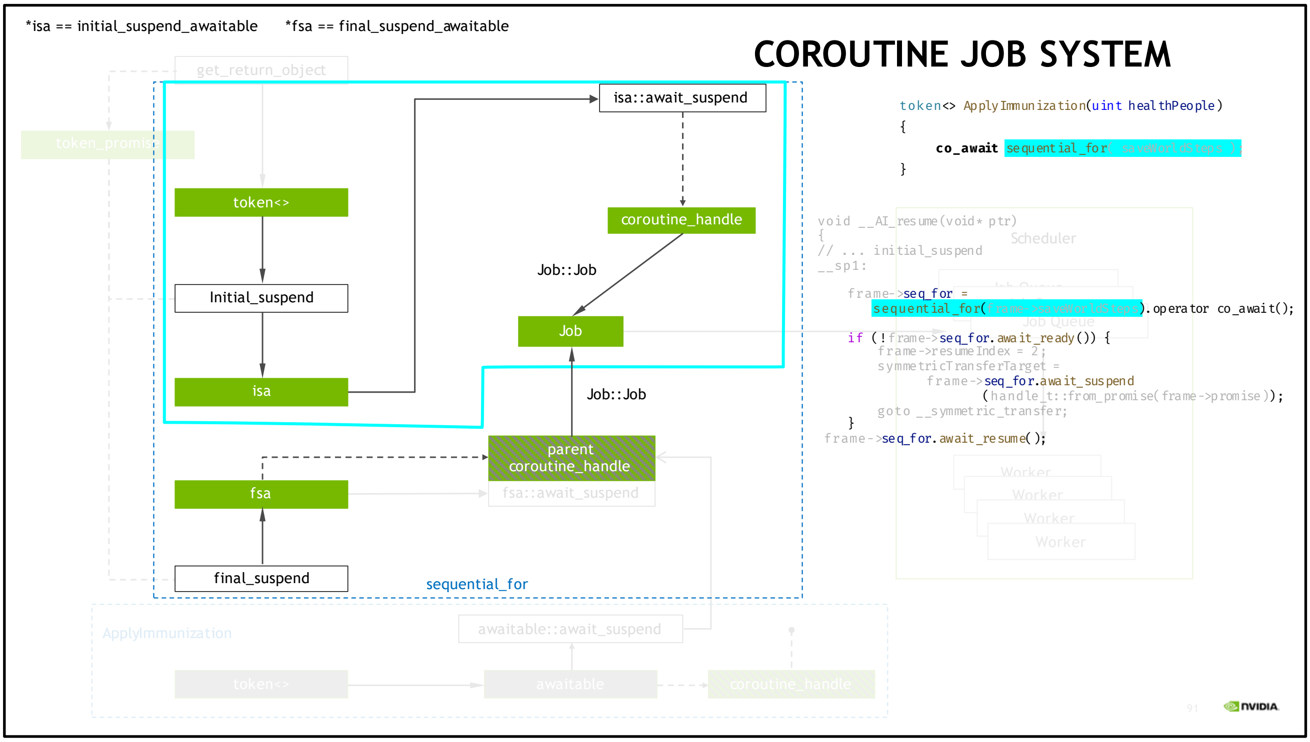
```
token<> ApplyImmunization(uint healthPeople)
{

    co_await sequential_for( saveWorldSteps );
}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```

```
void __AI_resume(void* ptr)
{
// ... initial_suspend
__sp1:
    frame->seq_for = sequential_for(frame->saveWorldSteps).operator co_await();

    if (!frame->seq_for.await_ready()) {
        frame->resumeIndex = 2;
        symmetricTransferTarget = frame->seq_for.await_suspend(handle_t::from_promise(frame->promise));
        goto __symmetric_transfer;
    }

   frame->seq_for.await_resume();
```

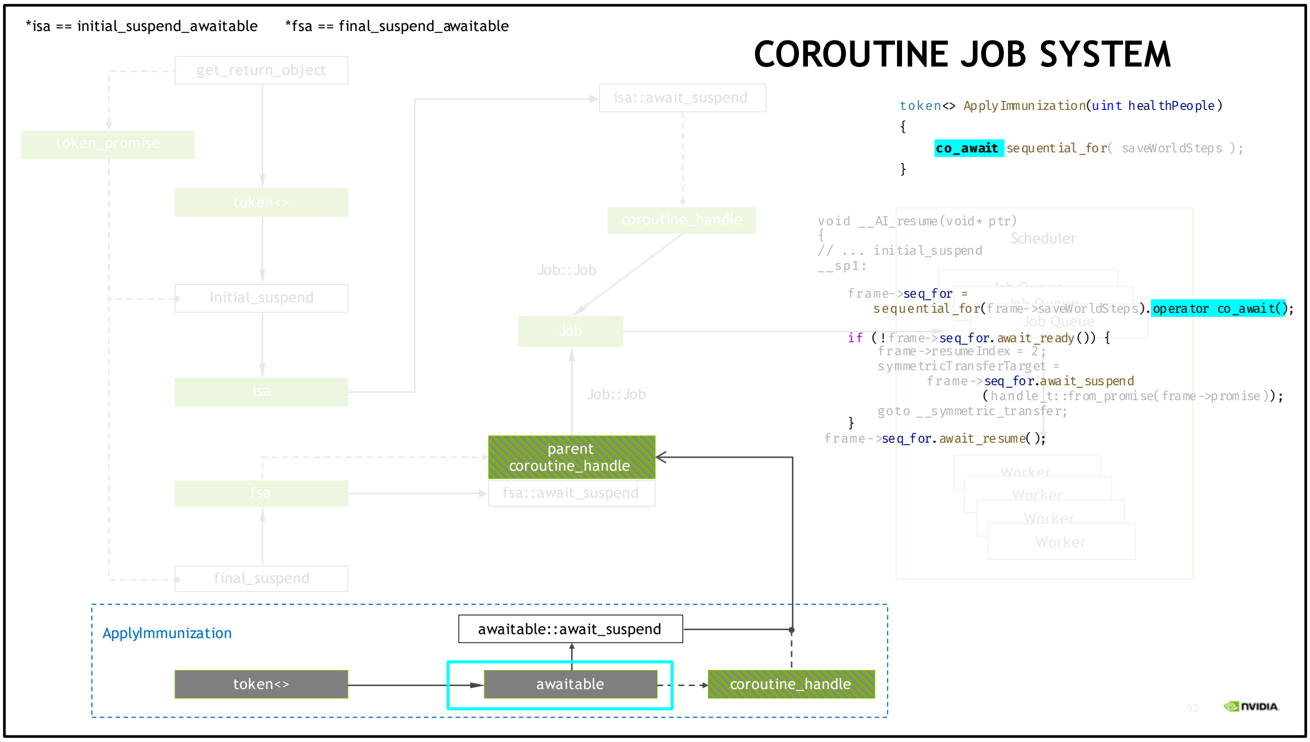

Which is like this, the difference is literally only the names.



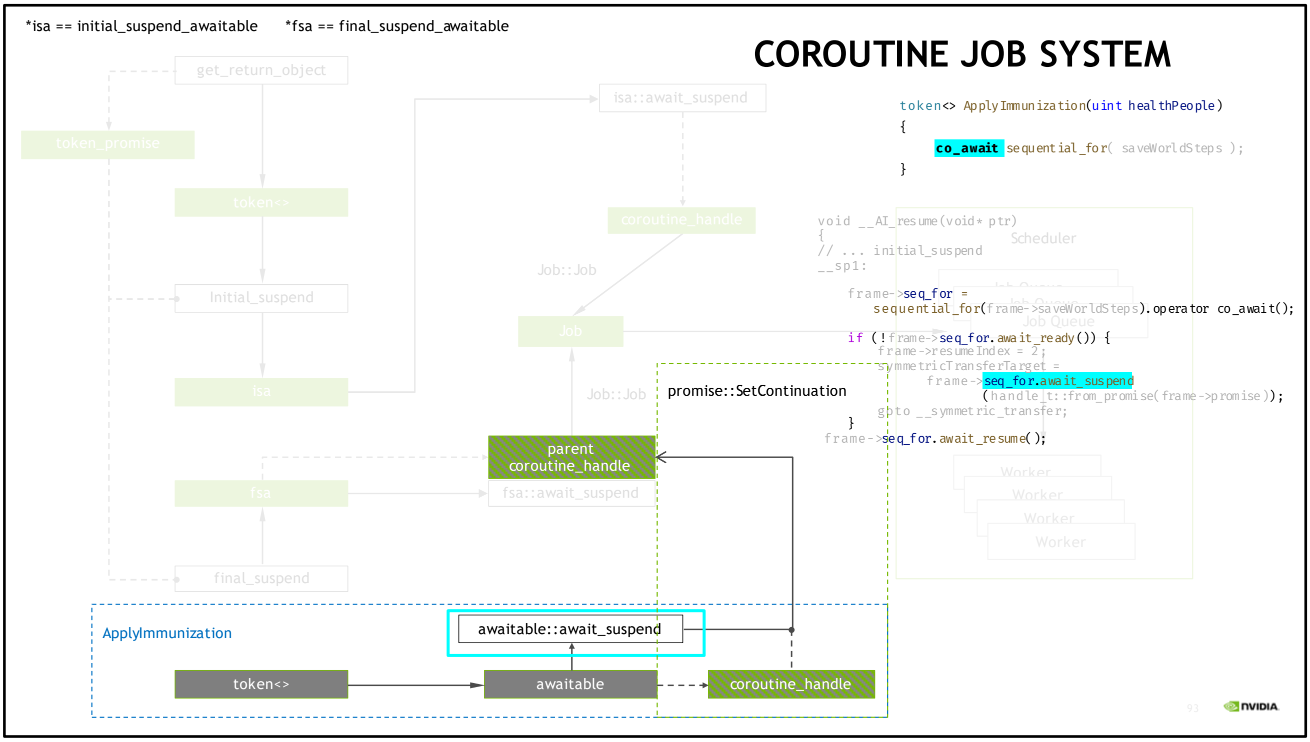
And, this is our ApplyImmunization. We’ve walk through the process. We will get the token, and initial suspend



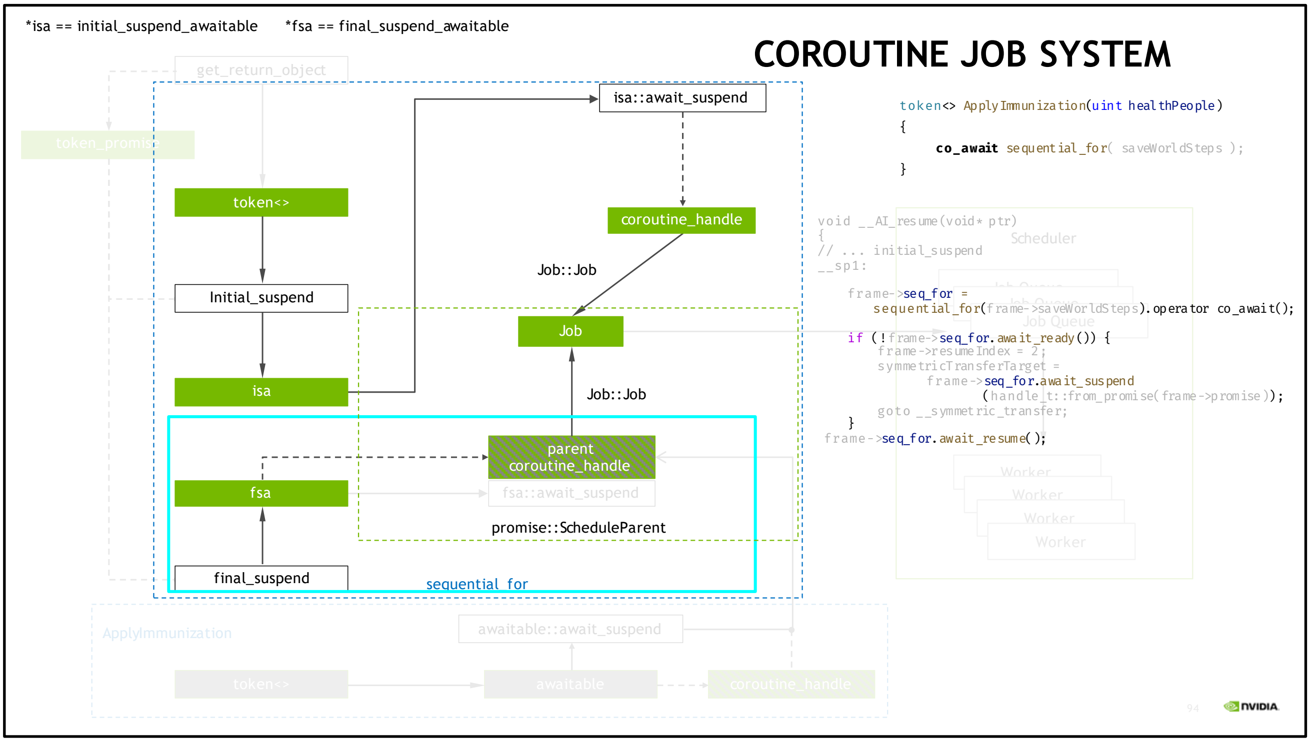
And then we invoke the sequential_for, same process, but in this graph, more details. And eventually, it will be scheduled on the scheduler.



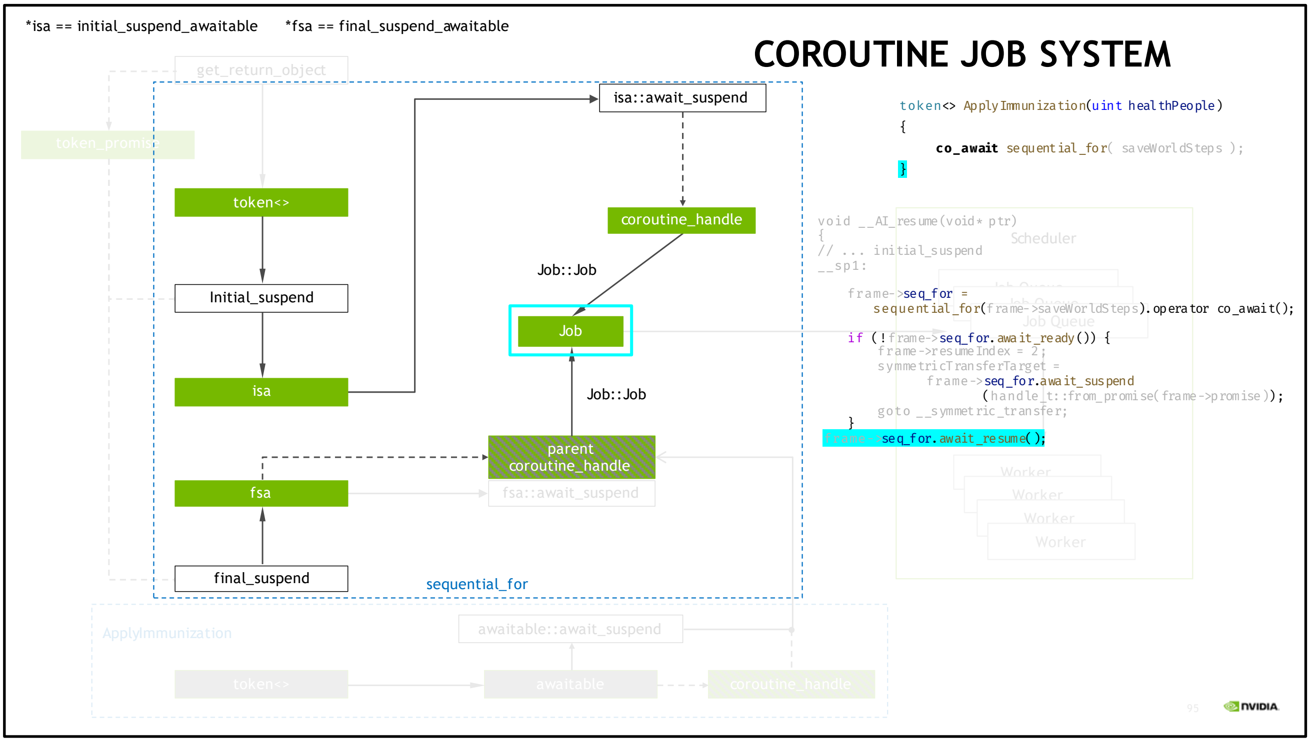
Then outside the function we immediately invoke operator co_await on the returned token object.



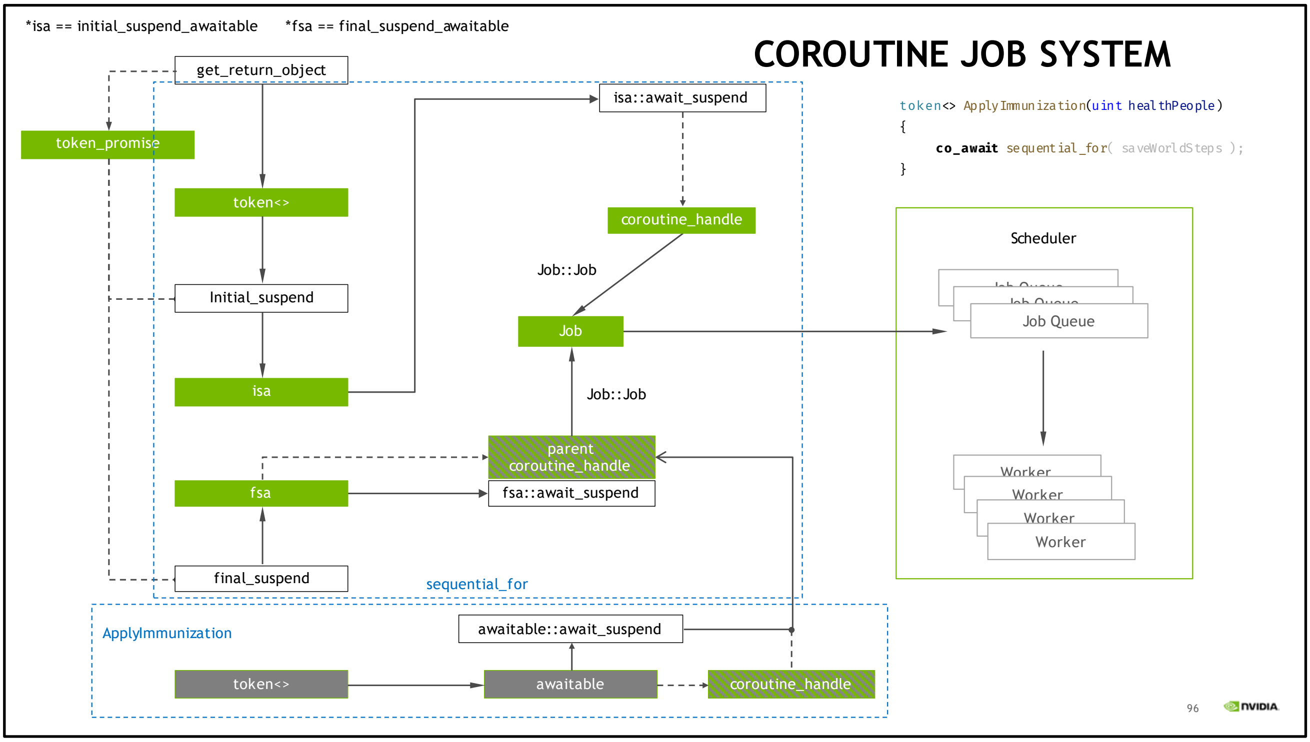
And we will suspend it, because await_ready will always return false. When we suspend, we will call setContinuation to remember what we need to resume after we are done with the sequential_for



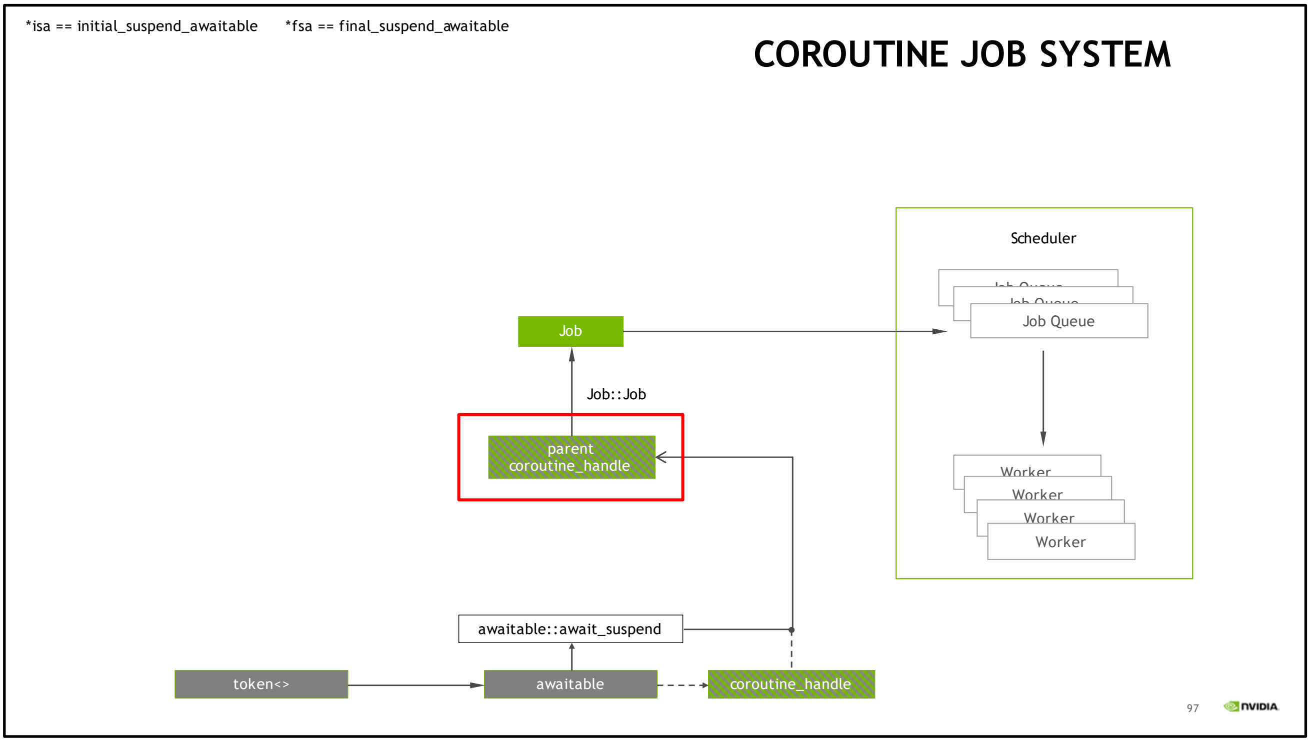
Then when sequential for is almost done, we will invoke finial_suspend, where we call scheduleParent, to re-schedule the ApplyImmunization on the scheduler.



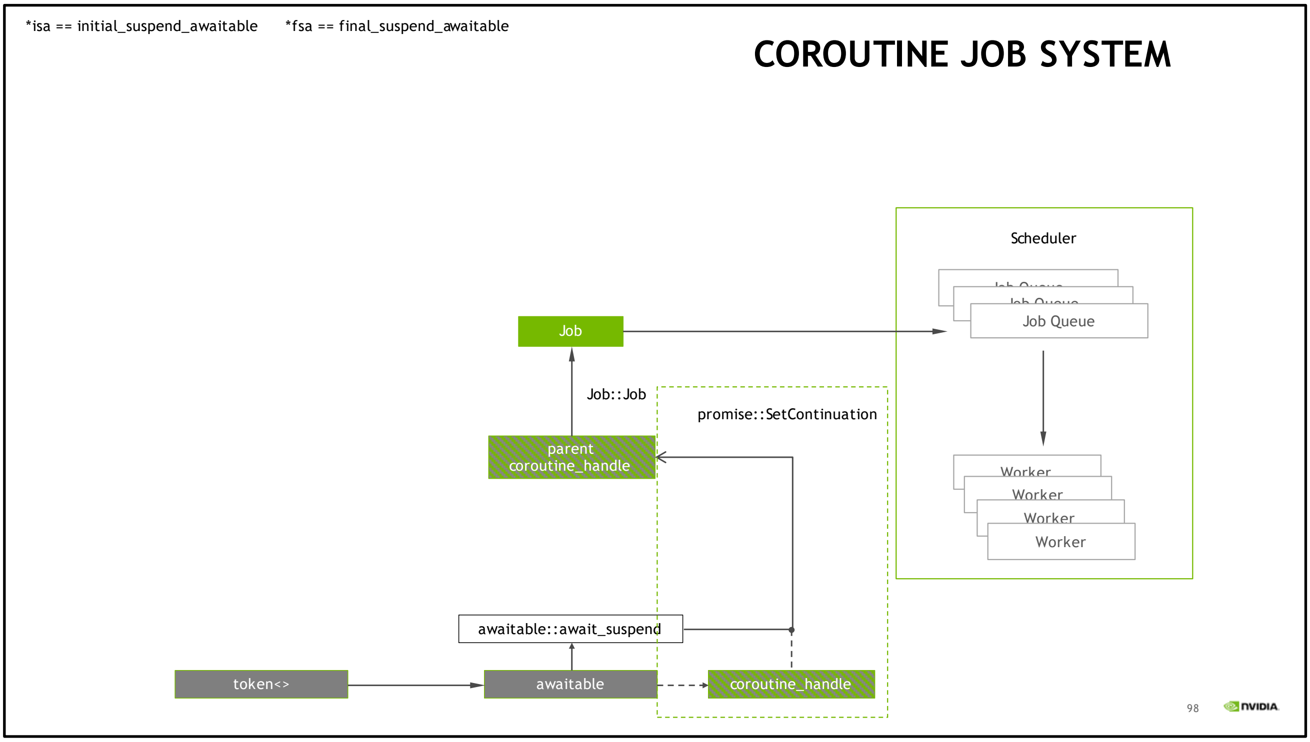
At eventually, applyImmunization will resume again



So again, in this case, ApplyImmunization is the bottom part, and the sequential for is the top part.



Okay, now let's talk about some code. This time, I strip out the most of the graph, and let's look at those piece one at a time. [animation] But focus on how the parent coroutine handle changes. First for this part, this is where SetContinuation happens.



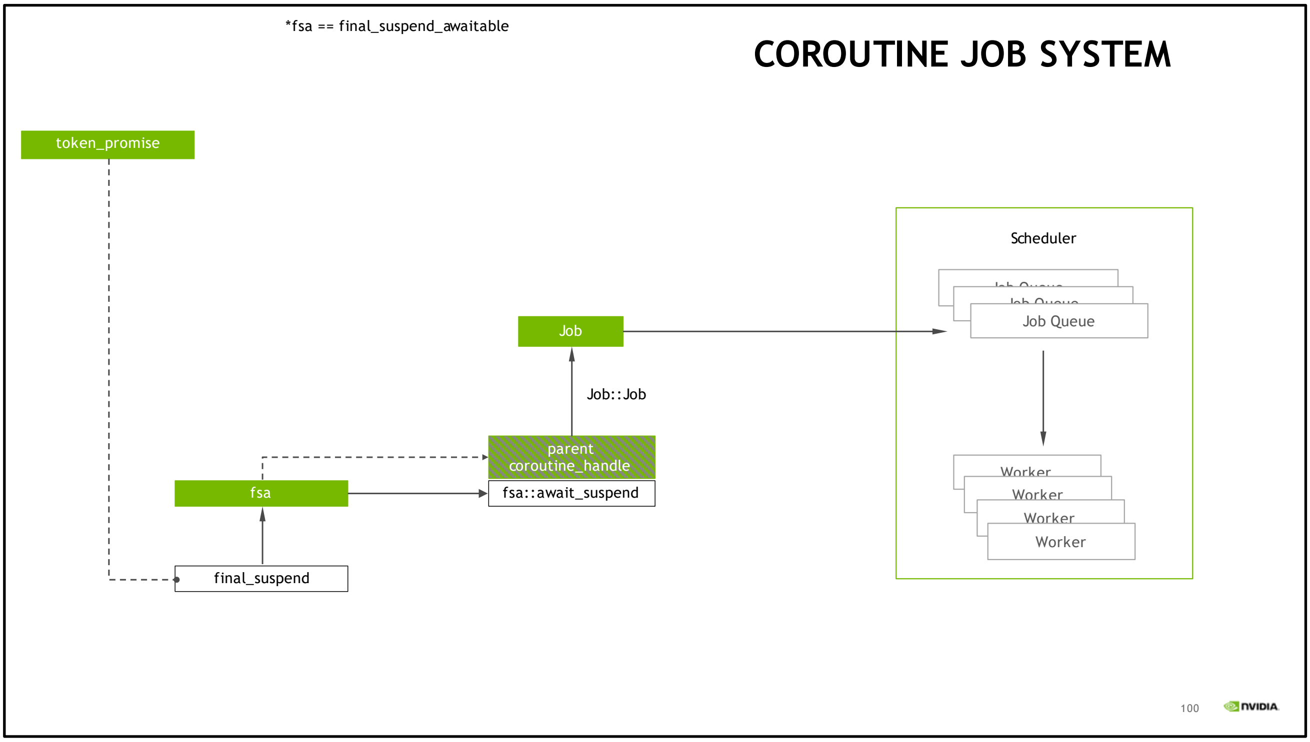
And btw, SetContinuation is here.

co_await token<T>

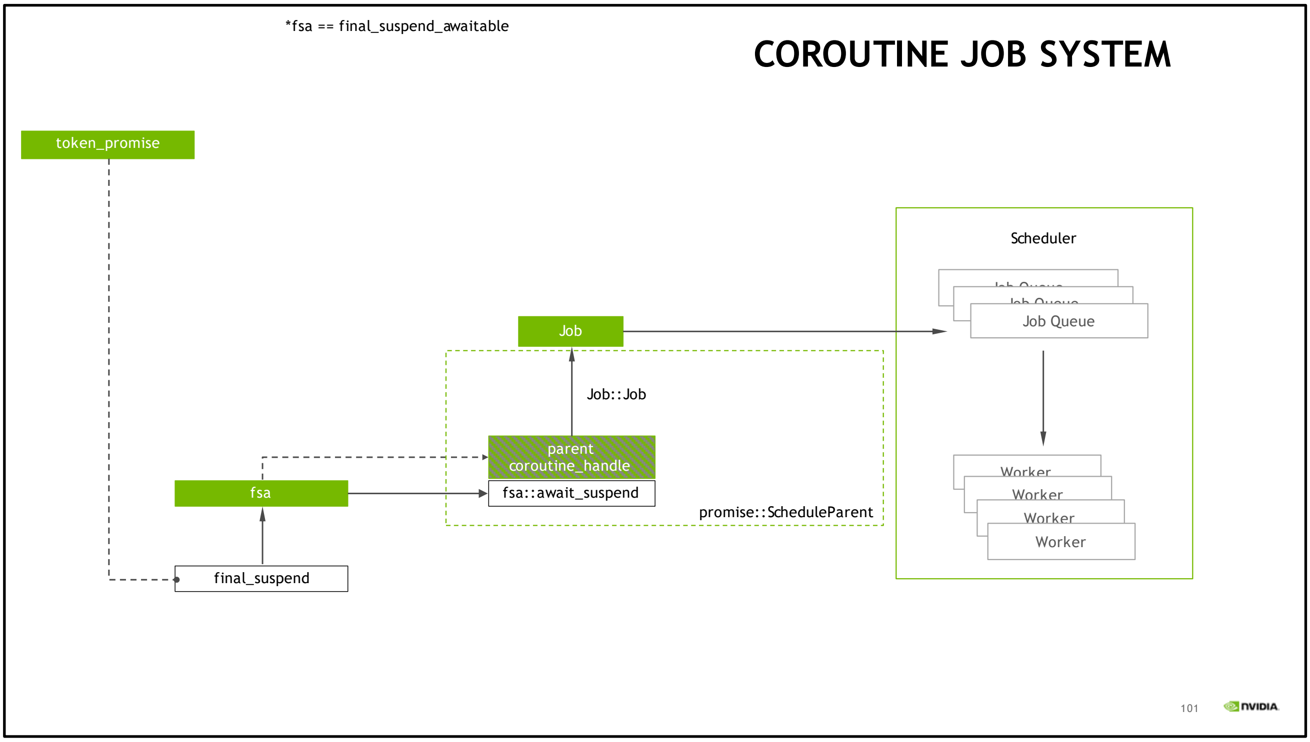
```
template<typename  T> class  token
{
auto operator co_await() const
{
    struct awaitable: awaitable_base
    {
        using awaitable_base::awaitable_base;

        decltype(auto)  await_resume();

        template<typename  Promise>
        bool await_suspend(  std::experimental::coroutine_handle<Promise>  awaitingCoroutine  ) noexcept
        {
            return coroutine.promise().SetContinuation(  awaitingCoroutine  );
        }

    }
    return awaitable{ mHandle };
}
}
```

99 NVIDIA

And in code form – We construct a awaitable object, where we define the await_suspend function, then inside, we have the access of the parent coroutine so that we can remember it and resume it when it’s eventually ready.



On the other hand, we resume it when sequential_for is done,



And we invoke ScheduleParent to re-schedule it again.

token<void>

```
struct token_dispatcher;
struct final_awaitable;
class token_promise
{
    token get_return_object();
    token_dispatcher inital_suspend()
    {
        auto& scheduler = Scheduler::Get();
        bool isOnMainThread =
            scheduler.GetMainThreadIndex() == scheduler.GetThreadIndex();
        return token_dispatcher{ isOnMainThread };
    }
    final_awaitable final_suspend() { return {}; }
}
```

```
token<> ApplyImmunization(uint healthPeople)
{
    //......
}
```


And in code, we need to go back to here, it’s time to explain this final_awaitable, which we skipped before

Resume Parent

```
struct final_awaitable
{
    bool await_ready() { return false; }
    void await_resume() {}

    template<typename Promise>
    void await_suspend( std::experimental::coroutine_handle<Promise> handle )
    {
        static_assert(std::is_base_of<promise_base, Promise>::value);
        promise_base& promise = handle.promise();
        promise.ScheduleParent();
    }
};
```

```
token<> ApplyImmunization(uint healthPeople)
{
    //......
}
```

103

In its await_suspend, we run ScheduleParent. And as we mentioned before, we always return false for await_ready so that we can access promise object when we run the logic.

Resume Parent

```
struct final_awaitable
{
    bool await_ready() { return false; }
    void await_resume() {}

    template<typename Promise>
    void await_suspend( std::experimental::coroutine_handle<Promise> handle )
    {
        static_assert(std::is_base_of<promise_base, Promise>::value);
        promise_base& promise = handle.promise();
        promise.ScheduleParent();
    }
};
```

```
token<> ApplyImmunization(uint healthPeople)
{
    //......
}
```

- Relaunch the parent coroutine as another job
- Can optionally skip the Scheduler and directly run it to trim the overhead

So far we say we need to put it on scheduler, but potentially we can construct the job and immediately declare it to avoid the overhead of scheduling.

GOAL

```
token<> ApplyImmunization(uint healthPeople)
{
    co_await sequential_for( saveWorldSteps );
}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```


Okay, go back to our save world mission, feel like so far, we are only writing some empty check. We need to stop and do something practical, since we don't have much time left.

GOAL

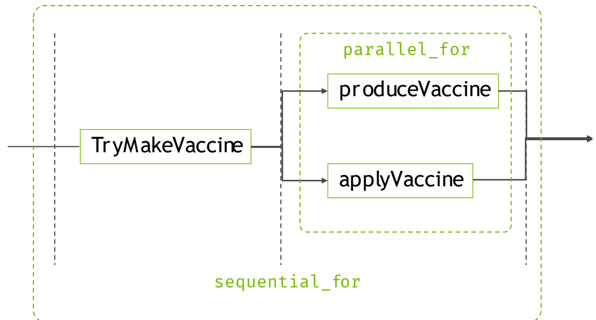
```
token<> ApplyImmunization(uint healthPeople)
{
    uint i = 0;

    std::atomic<uint> vaccineStock = 0;
    bool vaccineProductionTermniationSignal = false;

    std::vector<> step2 = {
2       ProduceVaccine( vaccineStock, vaccineProductionTermniationSignal ),
2       ClinicApplyVaccine( healthPeople, vaccineStock, vaccineProductionTermniationSignal )
    };

    std::vector<> saveWorldSteps = {
1       TryMakeVaccine(),
        parallel_for( step2 )
    }

    co_await sequential_for( saveWorldSteps );
}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```


To achieve this, we have three steps. First one is to do research and develop a vaccine.

Then when we have a recipe, we can produce those vaccine and apply those in the clinics.

GOAL

```
token<> ApplyImmunization(uint healthPeople)
{
    uint i = 0;

    std::atomic<uint> vaccineStock = 0;
    bool vaccineProductionTermiationSignal = false;

    std::vector<> step2 = {
        ProduceVaccine( vaccineStock, vaccineProductionTermiationSignal ),
        ClinicApplyVaccine( healthPeople, vaccineStock, vaccineProductionTermiationSignal )
    };

    std::vector<> saveWorldSteps = {
        TryMakeVaccine(),
        parallel_for( step2 )
    }

    co_await sequential_for( saveWorldSteps );
}

void NonCoroutineFunction()
{
    auto saveWorld = ApplyImmunization( 3000 );
    saveWorld.Wait();
}
```



107

and we can do these two in parallel. We don’t need to stock enough vaccine and them start to use them. They are more like producer and consumer.

But definitely we cannot do anything before we figure out a effective vaccine

Job Composite

```
token<int> add_one(int a) {
    co_return a+1;
}
token<> sequential_tasks()
{
    int result = 0;
    result = co_await add_one(result);
    result = co_await add_one(result);
    result = co_await add_one(result);

    printf("%d", result); // 3
}
```

```
token<>  long_slow_work(atomic<int>&  counter)
{
    Sleep(1000);
    counter--;
    return;
};
token<>  parallel_tasks()
{
    atomic<int>  counter  = 3;
    long_slow_work(counter);
    long_slow_work(counter);
    long_slow_work(counter);

    while(counter  != 0);

    co_return;
}
```

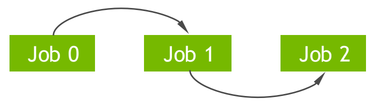
108

So we have those two kind of job composition. Sequential and parallel.
Sequential express a chain of tasks that have to be finished in determined order.
Parallel means they can be done at the same time.

Job Composite

Sequential Compositing, chaining

```
template<typename  Deferred>
token<>  sequential_for(  std::vector<Deferred>  deferred  )
{
   auto makeTask  = []( co::token<>  before,  Deferred  job ) -> co::token<>
   {
      co_await  before;
      co_await  job;
   };

   co::token<>  dependent;
   for(size_t  i  = 0;  i  < deferred.size();  ++i)  {
      dependent   = makeTask(  std::move(dependent),   deferred[i]  );
   }

   co_await  dependent;
}
```



Look at what happens here. So first, we transform each job into a new job. It first waits on the previous job, then execute the workload.

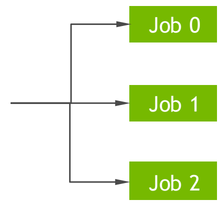
And the whole new job can be waited by the next job.

And notice that because of the composition we want, the job cannot be executed until the dependency is done.

Job Composite

Parallel Compositing

```
template<typename  Deferred>
token<>  parallel_for(  std::vector<Deferred>  deferred  )
{
   single_consumer_counter_event  counter(deferred.size());

   auto makeTask = [&counter](  Deferred  job  ) -> co::token<>
   {
      co_await  job;
      counter.decrement(  1  );
   };

   for(auto&  d:  deferred) {
      makeTask(  std::move(d)  );
   }

   co_await  counter;
}
```



On the other hand, for parallel, what we need is a counter. We decrement on finishing each job, and we wait on the counter reaching 0.

Job Composite

- Easy to implement
- Can be efficient if the overhead of creating/disapatching job is low
- Job composition requires `deferred_token` - creating job but do not dispatch it until we need to do so(lazy job)

```
template<bool Deferred, typename T>
class token;

template<typename T>
using deferred_token = token<true, T>;
```

Implementation that can avoid the overhead:
https://github.com/lewissbaker/cppcoro/blob/master/include/cppcoro/when_all_ready.hpp

token_dispatcher

```
template<typename Promise>
void Scheduler::Schedule( const std::experimental::coroutine_handle<Promise>& handle );
struct token_dispatcher // ret of initial_suspend for token_promise
{
    bool shouldSuspend;
    token_dispatcher(bool shouldSuspend): shouldSuspend(shouldSuspend) {}
    bool await_ready() { return false; }
    void await_resume() {}
    bool await_suspend(coroutine_handle<> handle)
    {
        using namespace std::experimental;
        coroutine_handle<token_promise> realHandle = coroutine_handle<token_promise>::from_address( handle.address() );
        mSuspendedPromise = &realHandle.promise();
        ENSURES( mSuspendedPromise != nullptr );

        if( shouldSuspend ) {
            Scheduler::Get().Schedule( realHandle );
        }

        return shouldSuspend;
    }

protected:
    token_promise* mSuspendedPromise = nullptr;
}
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```


Recall this slide when we were discussing the token dispatcher.

deferred_token<T>

```
template<typename Promise>
void Scheduler::Schedule( const std::experimental::coroutine_handle<Promise>& handle );
struct token_dispatcher // ret of initial_suspend for token_promise
{
    bool shouldSuspend;
    token_dispatcher(bool shouldSuspend): shouldSuspend(shouldSuspend) {}
    bool await_ready() { return false; }
    void await_resume() {}
    bool await_suspend(coroutine_handle<> handle)
    {
        using namespace std::experimental;
        coroutine_handle<token_promise> realHandle = coroutine_handle<token_promise>::from_address( handle.address() );
        mSuspendedPromise = &realHandle.promise();
        ENSURES( mSuspendedPromise != nullptr );

        bool scheduled;
        if constexpr( Deferred ) {
            scheduled = true;
        } else {
            if( shouldSuspend ) {
                Scheduler::Get().Schedule( realHandle );
            }
            scheduled = shouldSuspend;
        }

        return scheduled;
    }
```

```
token<> ApplyImmunization(uint  healthPeople)
{
    //......
}
```

113

This is what need to change. So basically if this is a deferred task, we always pretend this is already scheduled. But do not actually schedule it.

co_await token<T>

```
template<typename  T> class  token
{
auto operator  co_await()  const
{
    struct  awaitable:  awaitable_base
    {
        using awaitable_base::awaitable_base;

        decltype(auto)  await_resume()
        {
            auto& coro  = this->coroutine;
            using ret_t  = decltype(coro.promise().result());
            if constexpr  (std::is_void_v<ret_t>)  {
                  return;
            } else  {
                  return  coro  ? T{}  : coro.promise().Result( );
            }
        }
    };
    return  awaitable{  mHandle  };
}
}
```


On the other hand, we schedule it when someone is waiting on it. Which is in co_await

co_await deferred_token<T>

```
template<typename  T> class  token
{
auto operator  co_await()  const
{
    struct  awaitable:  awaitable_base
    {
        using awaitable_base::awaitable_base;

        decltype(auto)  await_resume()
        {
            auto& coro  = this->coroutine;
            using ret_t  = decltype(coro.promise().result());
            if constexpr  (std::is_void_v<ret_t>)  {
                  return;
            } else  {
                  return  coro  ? T{}  : coro.promise().Result( );
            }
        }
    };
    if constexpr  (Deferred)  {
        Dispatch();
    }
    return  awaitable{  mHandle  };
}
```

115

This essentially provide a lazy job mechanism. If no one ever waits on the job, the job will not be scheduled.

COROUTINE LIFETIME

When to call handle.destroy()

Multiple (necessary)elimination points:

- `~token()` – no one is waiting for the result
- awaitable dtor for `token::operator co_await` – after retrieving the result
- `~Job()` – work is done

Solution – Last wait to release:

- Add ref counting mechanism to promise(an atomic int)
- +ref on entering the scope(token(), awaitable ctor, Job())
- -ref on leaving the scope(~token(), awaitable dtor, ~Job()), destroy if ref == 0
- The ref won't re-increase because of how system works, so it's safe to destroy if ref == 0

116

~operation? - operator co_await needs some way to fetch the result & knowing the status of execution & we want the data dependency is one direction(token depends on the promise),

~token? – I want to support launch and forget type of job, so the lifetime is better not bound with the token

awaitable? – Same, people not necessarily want to wait on something

After ctor token, ref counting will at least have one and co_await needs the token.

COROUTINE LIFETIME

When to call handle.destroy()

Multiple (necessary)elimination points:

- ~token() – no one is waiting for the result
- dtor of awaitable for token::operator co_await – after retrieving the result
- ~Operation() – work is done

Another option – Expose the lifetime to user:

- Forward the coroutine ownership to RAII pin object – destroy in dtor
- User hold them within scope and manually release
- Higher level system can impose different mechanism

Credit to Arne Schober(@Khipu_Kamayuq)

LET'S TRY IT!


Trying to find a vaccine......
============== current status ================
People left: 3000
vaccine left: 0
lab 2 did not find a vaccine.... Retry....
lab 0 did not find a vaccine.... Retry....
lab 3 did not find a vaccine.... Retry....
lab 1 found a vaccine!
lab 4 did not find a vaccine.... Retry....
============== current status ================
People left: 3000
vaccine left: 0
lab 0 did not find a vaccine.... Retry....
lab 3 did not find a vaccine.... Retry....
lab 2 found a vaccine!
============== current status ================
People left: 3000
vaccine left: 0
============== current status ================
People left: 1371
vaccine left: 0
============== current status ================
People left: 531
vaccine left: 0
============== current status ================
People left: 0
vaccine left: 144
WORLD SAVED

WHAT'S THE GAIN

More than syntax sugar

- Dependency Management is gone!
  - No more bookkeeping, reduced overhead
  - Potential immediate execution of child job
- Cleaner interface
  - No need to deal with user data – Handled by coroutine frame
  - No syntax noise – Not even lambda
- Improved memory allocation
  - Heap allocation might be avoidable
  - Custom allocator integration is seamless

AND REAL WORLD IS NOT PERFECT

Even better if not……

- Challenge to integrate profiler to task
  - Coroutine can be resumed in a different thread, hence hard to visualize
  - Missing handy preprocessors like __COROUTINE_SUSPENSION_POINT__
  - RAII style mechanism is broken (thread sensitive context)
- Challenge for system developer to implement the system
  - Lack of Tool support
  - Harder to debug – a lot of generated “invisible” code
- Require users to have extra knowledge to use the system

120

Real world is not perfect, especially comes to engineering.

Arne Schober

Christopher Forseth

Gor Nishanov

Jennifer Yao

Jonathan Emmett

Lewis Baker

Victor Tong

Thanks for all the help that makes this possible!

Question?