

Coroutines

Mittwoch, 15. März 2023 13:32

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

When we define a task in programming language, we define it from the two aspects:

- Execution context (single thread, sub-tasks divided and hosted on multiple threads, thread pool, etc.)
- Execution flow (synchronous, asynchronous)

Coroutines stands for **cooperative** routines, that can be **voluntarily** (explicitly) **suspended** - passing the control flow to the caller (even another coroutine): the one that can **resume** it.

In C++ we already have `std::async` call - that represents the asynchronous task that can be executed either deferred on the same thread, or launched on a different thread, at which point the caller and the task running concurrently, competing for the same computational resources (CPUs).

This mechanism involves the OS (kernel), in terms of the Scheduler: **preempting** one thread in favor of another, based on some scheme (usually, thread priorities).

⇒ More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Thread_Attributes.pdf

Coroutines offer overall better utilization of system resources, passing the control flow back and forth - without need for nested callbacks and continuation-passing style code: chained operations.

Coroutines provide the sequential (linear) code flow for the asynchronous tasks - which also results in more intuitive syntax that is easier to write and read - and therefore maintain.

This also makes the integration with legacy code easier, since it can be done iterative, gradually replacing the existing callbacks, or `std::async` tasks: but only if there is a reason for that.

What are coroutines?

Technically speaking, coroutines are functional blocks that have

- **Result type**: a type that will be returned by the coroutine, as control object
- Any of the keywords: **co_yield**, **co_await**, **co_return**

Under the hood, compiler will generate the **state-machine**, that allows transition from initial suspension point, to the final state, at which point the coroutine will be destroyed.

The mental model could be

```
{
    promise_type promise;
    auto resultType = promise.get_return_object();

    co_await promise.initial_suspend(); // check initial suspension: outside of try/catch block
    try
    {
        <body-statements>

        co_return promise.return_void(); // or promise.return_value(val);
        goto Finalize;
    }
    catch(...)
    {
        promise.unhandled_exception(); // exception handling
    }
Finalize:
    co_await promise.final_suspend(); // check final suspension: outside of try/catch block

    return resultType;
}
```

@Hint: You can use CompilerExplorer/Insight tool from Andreas Fertig to inspect how the generated code actually looks like. For that, you need to enable: "*show coroutine transformation*"

Result type

The coroutine's return type is a wrapper around usually internally implemented interface that must be named or aliased as **promise_type**.

This interface describes the machinery behind the coroutine:

== Creation ==

The very first call is `get_return_object()` which creates the coroutine's control type.

In order to support suspend/resume mechanism, coroutines are **stackless** - they are created on the heap, which is known as coroutine frame.

The return type of coroutine is nothing but **wrapper around the coroutine frame: `std::coroutine_handle`**

```
template <typename T>
class [[nodiscard]] ResultType final
{
public:
    class promise_type; // forward declaration
    using coroutine_handle = std::coroutine_handle<promise_type>;

    /**
     * Required nested promise_type entity
     */
    struct promise_type {
        ResultType get_return_object() {
            return ResultType{this};
        }
        // the rest of the interface
        ...
    };

    explicit ResultType(promise_type* promise) noexcept: _handle(coroutine_handle::from_promise(*promise))
    {}

    operator std::coroutine_handle<> () const {
        return handle_;
    }

private:
    coroutine_handle handle_;
};
```

The alternative is to use concept: `std::coroutine_traits`, to attach the required *promise_type* to some entity that is this way turned into eligible coroutine's return type

```
class Socket final
{
public:
    explicit Socket(int fd) noexcept: fd_(fd) {}
    ~Socket() { ::close(fd_); }

    using data_type = std::vector<std::byte>;
    ...
};

using domain_type = enum class DomainType {ipv4 = AF_INET, ipv6 = AF_INET6, unix = AF_UNIX};
using socket_type = enum class SocketType {stream = SOCK_STREAM, datagram = SOCK_DGRAM};

template <domain_type domain, socket_type socket>
struct SocketCreator
{
    int make_socket() const {
        return ::socket((int)domain, (int)socket, 0);
    }
};

struct socket_as_coroutine {}; // std template specialization requires at least one user-defined type

template<>
struct std::coroutine_traits<Socket, socket_as_coroutine>
{
    template <domain_type domain, socket_type socket>
    struct promise_type : SocketCreator<domain, socket>
    {
        Socket get_return_object() {return Socket{this->make_socket()}; }
        std::suspend_never initial_suspend() const noexcept { return{}; }
        std::suspend_never final_suspend() const noexcept { return{}; }
        void return_void() {}
    };
};
```

```

        void unhandled_exception() {}
    };
};

```

At which point the caller receives the coroutine's control object - as return type?

When the coroutine is for the first time suspended, or - if there is no suspension point, when the coroutine is destroyed (which is not quite useful).

== Suspension/Resumption ==

Promise interface describes the initial and final suspension points of a coroutine.

As you can see in the mental model, they are outside of the try/catch block - so it's important that they do not throw exception.

If initial suspension point is actively set (using predefined `std::suspend_always`), the coroutine is created with laziness in mind, returning immediately control to the caller.

The caller is responsible to resume it.

Final suspension point gives the possibility to make the last snapshot of the coroutine frame, delaying it's destruction - giving control back to the caller. To caller can inspect - query the state, or preserved value of coroutine at the point when finalization is reached.

And the caller is responsible to destroy it.

To avoid memory leaking, we can ensure proper cleanup, by capturing the finalization in the control object destructor (RAII)

```

~ResultType() {
    if(_handle) {
        _handle.destroy();
    }
}

```

== Exception handling ==

The coroutine's error handling is based on catching the exceptions, in try/catch block around the coroutines body statements.

The *promise_type* gives the callback entry that can be implemented as user-specific error handling.

There are different kind of strategies:

- Call `std::terminate` to terminate the application, indicating non-recoverable error.
Not recommendable for the real applications, especially in embedded domain.
- Similar to `std::promise<T>`, capture either value or exception within `std::variant<std::monostate, T, std::exception_ptr>`, or `std::expected<T, std::error_code>` in C++23 library
- We can just rethrow, delegating responsibility (what we like the most) to the caller

```

void unhandled_exception() {
    std::rethrow_exception(std::current_exception());
}

```

- The most relaxed one: completely ignore the exception (UB)

```

void unhandled_exception() {}

```

Suspending coroutines

** co_await **

In case that we **wait on the result of asynchronous task**, i.e. having the pseudo-code like

```

auto result = co_await Awaitable<Task, Result>{};

```

our *Awaitable* type could look something like this

```

template <typename Task, typename Result = std::invoke_result_t<Task>>
struct Awaitable: std::suspend_always {
    using co_handle = promise_type::co_handle; // Awaitable as coroutine!

    /*
     * Caller passing at suspension point: co_await,
     * the handle of the suspended coroutine - other, which
     * waits on the result of the asynchronous task.
     */
    co_handle await_suspend(std::coroutine_handle<> other) {
        coro_.promise().continuation_ = other; // so that we can resume caller
        // Execute hosted task
        result_ = task_();

        return coro_; // we return our own handle - handing over control the resumed caller
    }
}

```

```

/*
    The final state - that will be invoked at the point when
    Awaitable is resumed explicitly - by the caller (Resumer) obtaining the awaited result
*/
Result&& await_resume() {
    return std::move(result_);
}
};

```

This would mean that *Awaitable* interface can be part of the coroutine wrapper type itself, since

await_suspend() returns a **handle of the enclosing coroutine type**, so that:

- callee **remains suspended** - handing over control to the caller: to be explicitly resumed by the caller (*Resumer*)
- caller **itself is resumed** - it controls now when the callee will be resumed (*handle.resume()*) - the *awaited* result retrieved (by implicitly calling the *await_resume()*)

This is also known as "symmetric transfer"

Furthermore, the *await_suspend()* can return:

- **Boolean flag**, where

- TRUE: means that the caller remains on return suspended, until it's explicitly resumed (by *Resumer*, having the suspending coroutine's handle - *other*)
- FALSE: means the transition to the new state: invoking the *await_resume()* and returning the control back to the suspended caller that will be resumed

- **void** - has the same meaning as Boolean flag TRUE: the caller will remain suspended
In that case, the caller - the suspended coroutine, needs to be explicitly resumed

```

void CoResultType::resume() {
    if (not coro_.promise().continuation_.done()) {
        coro_.promise().continuation_.resume();
    }
}

```

<Example_1>: <https://godbolt.org/z/ibdeE3snz>

Since coroutines guarantee the "happens before" relationship – unlike the threads for which we either need to use synchronization primitives, or for lock-free programming: memory barriers to prevent compiler of doing optimization - reordering of operations, ensuring that all threads, synchronized on the same atomic variable, observe the same order of write/read operations

⇒ https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Lock-free%20programming%2C%20part%201.pdf

we can

- write asynchronous code in a procedural – sequential way
- pass the local variable in the scope of coroutine by reference

This enables us to write "in-place" code like

```

auto doubleTask(int& value)
{
    struct Awaitable : std::suspend_always
    {
        explicit Awaitable(int& value) noexcept: val_(value) {}
        bool await_suspend(std::coroutine_handle<> )
        {
            val_ *= 2;

            return true;
        }
    private:
        int& val_;
    };

    return Awaitable{value};
}

```

There is another, concise way, overriding the ***co_await*** unary operator

```

auto doubleTask(int& value) {

```

```

struct Awaitable
{
    explicit Awaitable(int& value) noexcept: val_(value) {}

    std::suspend_always operator co_await()
    {
        val_ *= 2;
        return {}; // defaulted - parameterless c-tor of the return type
    }
private:
    int& val_;
};

return Awaitable{value};
}

```

We can now write our small unit test

```

// coroutine
MyResultType myCoroutine(int& value) {
    ...
    co_await doubleTask(value);
}

// caller
int value = 10;
auto co = myCoroutine(value);
co.resume();
assert(value == 20);

```

What if our coroutine needs to produce the value, in produce-consumer scenario?

**** co_yield ****

In case that coroutine generates the value on which the caller is synchronized, there is a dedicated *co_yield* unary operator, which is shortcut for **co_await promise.yield_value(value)**

In other words, we need to ensure that *promise_type* has implemented *yield_value()* callback, which basically stores the value inside the *promise_type* and suspend itself

```

std::suspend_always yield_value(T&& value) noexcept(std::is_nothrow_move_constructible_v<T>) {
    value_ = std::move(value);
    return{}; // suspend itself
}

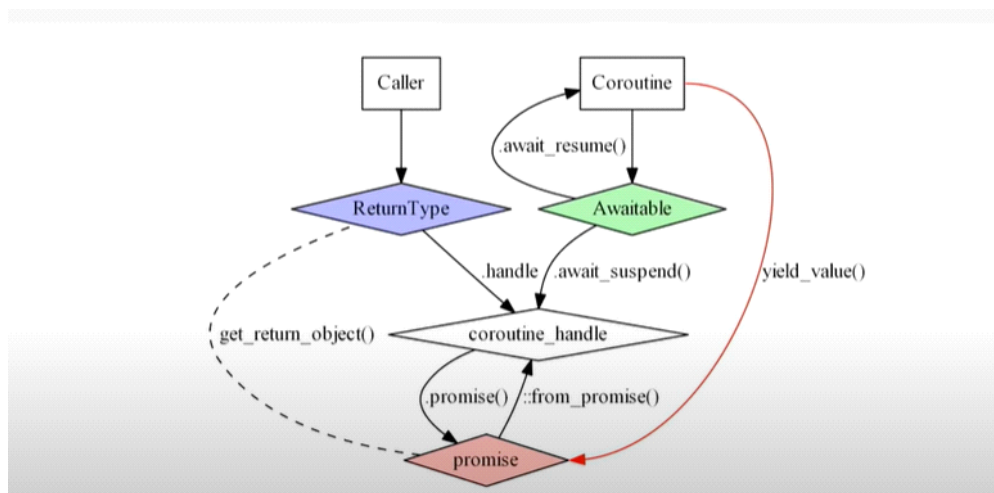
```

<Example_2>: <https://godbolt.org/z/qeKe75v71>

Simple **producer-consumer** example:

<Example_3> <https://godbolt.org/z/zrP175rs7>

At the end, this is a fine overview of the relationships and APIs that are involved around the coroutines, which I borrowed from Andreas Weis talk



Final thought

We do already have reactive libraries that implements the event-based asynchronous communication. The coroutines can be useful alternative to that, in terms of better utilization on system resources and having structural, deterministic code that simplifies the design solutions. When it comes to I/O asynchronous data transfer, coroutines can be particularly useful. They allow you to write code that appears sequential and intuitive, mimicking synchronous operations, while still benefiting from the asynchronous nature of I/O. This can result in code that is easier to write, read, and maintain. This also applies on the network: socket-based communications. The state-machines are also one example of handling the events and state-machine transitions in efficient way.

Links

Official paper

<https://timsong-cpp.github.io/cppwp/n4861/dcl.fct.def.coroutine>

Andreas Weis - CppCon 2022

[Deciphering C++ Coroutines - A Diagrammatic Coroutine Cheat Sheet - Andreas Weis - CppCon 2022](#)

Andreas Fertig

[\(34\) C++ Insights - Episode 36: Coroutine customization points - YouTube](#)

Lewis Baker: Structured Concurrency

<https://lewissbaker.github.io/>

<https://www.youtube.com/watch?v=1Wy5sq3s2rg>

Pavel Novikov: examples

<https://www.youtube.com/watch?v=7sKUAyWXNHA>

Šimon Tóth

[C++20 Coroutines — Complete* Guide | by Šimon Tóth | ITNEXT](#)

<https://itnext.io/daily-bit-e-of-c-coroutines-step-by-step-e726b976d239>