C++20 Coroutines

Miłosz Warzecha

Introduction

Coroutines allow you to suspend function execution and return the control of the execution back to the caller

Which .then allows you to run an asynchronous piece of code in the same way as you would normally run synchronous one

Which .then eliminates the need to deal with complicated and verbose syntax when writing asynchronous code

Synchronous program

```cpp
int a()
{
    return 42;
}

int doSomething()
{
    auto answer = a();

    // do something with the answer
    return answer + 1;
}
```

Asynchronous program

```
std::future<int> a()
{
    return std::async(std::launch::async, []{
        return 42;
    });
}

int doSomething()
{
     auto answer = a().get();

     // do something with the answer
     return answer + 1;
}
```

Asynchronous program

```
std::future<int> a()
{
    return std::async(std::launch::async, []{
        return 42;
    });
}

std::future<int> doSomething()
{
    return std::async(std::launch::async, []{
        auto answer = a().get();

        // do something with the answer
        return answer;
    });
}
```

What did we see there?

Problems with **performance** and problems with **structure**:

- Waiting introduces performance penalty
- Dealing with waiting introduces structure penalty (and possibly leaves performance in a bad shape too)

Dealing with performance bottlenecks is important because it can impact latency and throughput of our application.

Dealing with structural bottlenecks is important because it can slow down development process and make reasoning about your code more difficult than it should be.

Asynchronous program & coroutines

```
std::future<int> a()
{
    co_return 42;
}

std::future<int> doSomething()
{
    auto answer = co_await a();

    // do something with the answer
    return answer + 1;
}
```

What’s good about that?

- We are able to increase performance of our application using asynchronous calls

- We are able to introduce those asynchronous functionalities without disrupting the whole application

- We introduce asynchrony without over-compilicated syntax and confusing execution flow

Co_await? What’s that?

```
auto result = co_await a();
```

Co_await? What’s that?

```
auto result = co_await a();
```

```
co_awaitable_type
{
   bool await_ready();
   void await_suspend(coroutine_handle<>);
   auto await_resume();
};
```

- Call await_ready()
- If ready, call await_resume()
- If not ready, suspend the coroutine
- Call await_suspend() and return control back to the caller (or decide where the control flow should go)
- Get the result from await_resume()

Simple (co_)awaitable types

```
suspend_never
{
    bool await_ready() { return true; };
    void await_suspend(coroutine_handle<>) {};
    auto await_resume() {};
};

suspend_always
{
    bool await_ready() { return false; };
    void await_suspend(coroutine_handle<>) {};
    auto await_resume() {};
};
```

Simple (co_)awaitable types

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>42
```

Co_await? What's that?

```
auto result = co_await a();
```

```
co_awaitable_type
{
   bool await_ready();
   void await_suspend(coroutine_handle<>);
   auto await_resume();
};
```

- Call await_ready()
- If ready, call await_resume()
- If not ready, suspend the coroutine
- Call await_suspend() and return control back to the caller (or decide where the control flow should go)
- Get the result from await_resume()

Co_await? What's that?

```
auto result = co_await a();
```

```
co_awaitable_type
{
   bool await_ready();
   void await_suspend( coroutine_handle< >);
   auto await_resume();
};
```

- Call await_ready()
- If ready, call await_resume()
- If not ready, **suspend** the coroutine
- Call await_suspend() and return control back to the caller (or decide where the control flow should go)
- Get the result from await_resume()

Suspension mechanism

Like in the case of functions, the compiler need to construct a frame for the coroutine.

The coroutine frame contains space for:

- parameters
- local variables
- temporaries
- execution state (to restore when resumed)
- promise (to return values)

Coroutine frame is dynamically allocated (most of the time), before coroutine is executed.

Suspension mechanism

Suspension points are signified by the use of **co_await** keyword.

When a suspension is triggered, what happens is:

- Any values held in registers are written to the coroutine frame
- Point of suspension is also written to the coroutine frame, so the resume command will now where to come back (also so that the destroy operation knows what values will need to be destroyed)
- Compiler will create a **coroutine_handle** object, that refers to the state of our coroutine, and which has the ability to e.g. resume or destroy it

Suspension mechanism

Compiler will create a **coroutine_handle** object, that refers to the state of our coroutine, and which has the ability to

- Resume the coroutine
- Destroy it
- Get the promise object
- Check if the coroutine is done executing

Coroutine handle does not provide any lifetime management (it’s just like a raw pointer).

Coroutine handle can de constructed from the reference to the promise object (promise object is part of the coroutine frame, so the compiler knows where’s the coroutine related to that promise).

Co_await? What's that?

```
auto result = co_await a();
```

```
co_awaitable_type
{
   bool await_ready();
   void await_suspend( coroutine_handle< >);
   auto await_resume();
};
```

- Call await_ready()
- If ready, call await_resume()
- If not ready, **suspend** the coroutine
- Call await_suspend() and return control back to the caller (or decide where the control flow should go)
- Get the result from await_resume()

Co_await? What’s that?

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>42
```

Co_await? What's that?

```
coroutine_type coroutine()
{
    std::cout << "before suspension" << "\n";
    co_await suspend_always{};
    std::cout << "resumed" << "\n";
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << "\n";
}
```

```
Output:

>before suspension

>resumed

>42
```

Simple coroutine type

```
struct coroutine_type
{









    ...
private:





};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {









   };

   ...
private:





};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {








   };

   ...
private:



   coroutine_handle<promise_type> _coro;

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;








   };

   ...
private:



   coroutine_handle<promise_type> _coro;

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }






   };

   ...
private:



   coroutine_handle<promise_type> _coro;

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }








   };

   ...
private:
   coroutine_type(promise_type& p):
      _coro(coroutine_handle<promise_type>::from_promise(p)) {}

   coroutine_handle<promise_type> _coro;

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }







   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }





   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }




   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }

      void return_void() {}


   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }

      void return_value(int val) { _value = val; }


   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }

      void return_value(int val) { _value = val; }

      void unhandled_exception() { std::terminate(); }
   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
    ...

    void resume() {
        _coro.resume();
    }

    void get() {
        _coro.promise()._value;
    }

    ...

private:
    coroutine_handle<promise_type> _coro;
};
```

Simple coroutine type

```
struct coroutine_type
{
   ...

   using HDL = coroutine_handle<promise_type>;

   coroutine_type() = default;
   coroutine_type(const coroutine_type&) = delete;

   ~coroutine_type() {
      _coro.destroy();
   }

private:
   coroutine_type(promise_type& p) : _coro(HDL::from_promise(p)) {}
   coroutine_handle<promise_type> _coro;
};
```

Using the coroutine

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>42
```

Using the coroutine

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>12451356345
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }

      void return_value(int val) { _value = val; }

      void unhandled_exception() { std::terminate(); }
   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
    struct promise_type
    {
        int _value;
```

A coroutine is destroyed when:

- *final-suspend* is resumed
- coroutine_handle<>::destroy() is called

Whichever happens first

```
        void return_value(int val) { _value = val; }

        void unhandled_exception() { std::terminate(); }
    };

    ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_never{}; }

      void return_value(int val) { _value = val; }

      void unhandled_exception() { std::terminate(); }
   };

   ...

};
```

Simple coroutine type

```
struct coroutine_type
{
   struct promise_type
   {
      int _value;
      coroutine_type get_return_object() { return {*this}; }

      auto initial_suspend() { return suspend_never{}; }
      auto final_suspend() { return suspend_always{}; }

      void return_value(int val) { _value = val; }

      void unhandled_exception() { std::terminate(); }
   };

   ...

};
```

Using the coroutine

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>12451356345
```

Using the coroutine

```
coroutine_type coroutine()
{
    std::cout << “before suspension” << “\n”;
    co_await suspend_always{};
    std::cout << “resumed” << “\n”;
    co_return 42;
}

int main()
{
    auto coro = coroutine();

    coro.resume();

    std::cout << coro.get() << “\n”;
}
```

```
Output:

>before suspension

>resumed

>42
```

What we learned

- Creating a coroutine type involves a lot of customization points
- Coroutine suspension and resumption procedure can be controlled in those implementations
- Coroutine handle is our way to communicate with coroutine frame
- Promise object is our way to control the state of our coroutine and return values
- We have to remember that lifetime of our coroutine ends after the last suspension point

Yielding

- Ability to yield a value and suspend the coroutine immediately after
- Useful for lazy generation of infinite sequences

Simple generator type

```
struct generator_type
{
   struct promise_type
   {
      ...

      void return_void() {}

      auto yield_value(int val) {
            _value = val;
            return suspend_always{};
      };

      ...
   };

   ...
};
```

Simple generator type

```
generator_type func()
{
    for(int i = 0; i < 5; ++i)
        co_yield i;
}

int main()
{
    auto gen = func();
    std::cout << gen.get() << "\n";
    gen.resume();
    std::cout << gen.get() << "\n";
    gen.resume();
    std::cout << gen.get() << "\n";
    gen.resume();
    std::cout << gen.get() << "\n";
}
```

```
Output:

0

1

2

3
```

Concurrent and parallel systems

“Concurrency is about structure, and parallelism is about execution” – Rob Pike

Concurrency enables parallelism to thrive and do it’s job properly.

- It’s difficult to make your program worse by making it more concurrent

Parallelism pressures you to invest into creating a viable concurrent model, otherwise you won’t actually make things any better.

- It’s easy to make your program worse by making it more parallel.

“If the next generation of programmers makes more intensive use of multithreading, then the next generation of computers will become nearly unusable” - Edward A. Lee

Concurrent and parallel systems

Not concurrent and not parallel

?

Concurrent but not parallel

?

Not concurrent but parallel

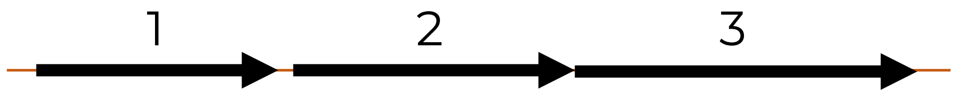
?

Both concurrent AND parallel

?

Concurrent and parallel systems

Not concurrent and not parallel



Concurrent but not parallel

?

Not concurrent but parallel

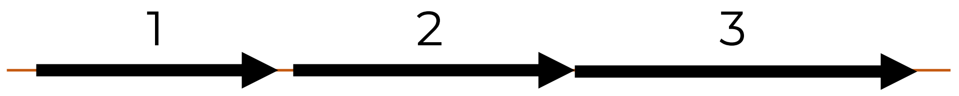
?

Both concurrent AND parallel

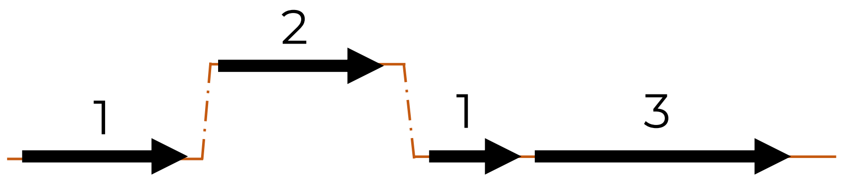
?

Concurrent and parallel systems

Not concurrent and not parallel



Concurrent but not parallel



Not concurrent but parallel

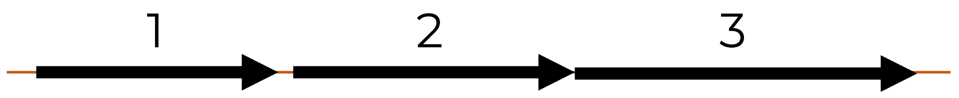
?

Both concurrent AND parallel

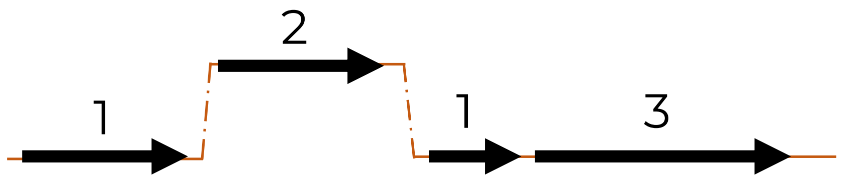
?

Concurrent and parallel systems

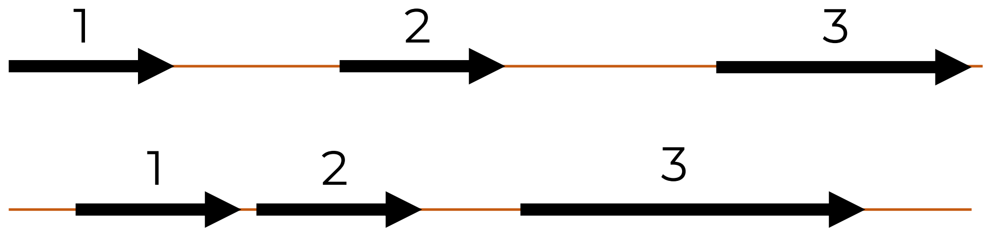
Not concurrent and not parallel



Concurrent but not parallel



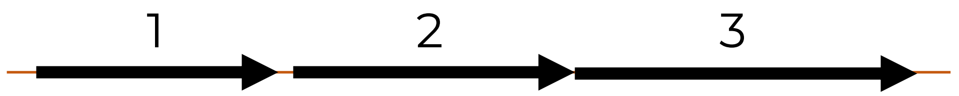
Not concurrent but parallel



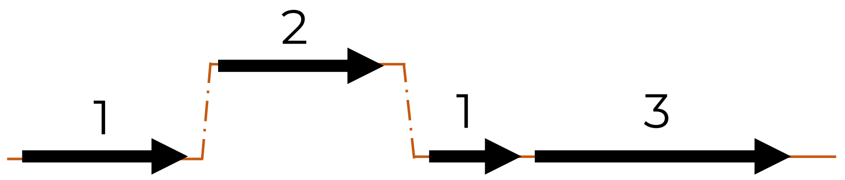
Both concurrent AND parallel

Concurrent and parallel systems

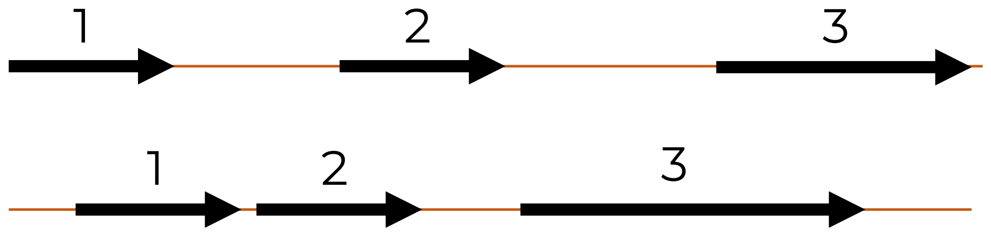
Not concurrent and not parallel



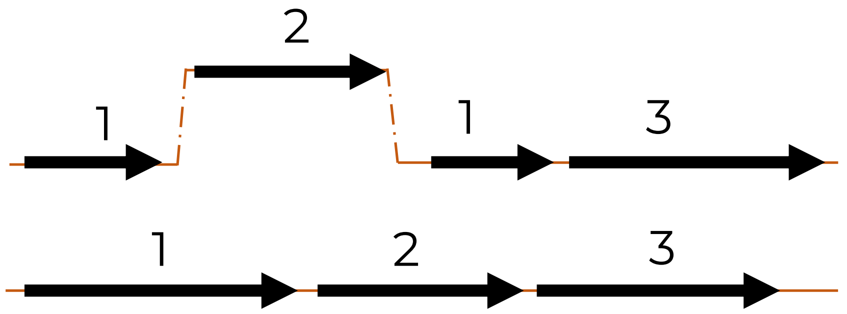
Concurrent but not parallel



Not concurrent but parallel



Both concurrent AND parallel



What does a concurrent structure mean?

There are 2 things which are definitely worth looking at when dealing with a concurrent application:

- shared mutable state
- encapsulation of mutating functions

In concurrent systems, order matters.

The ability to mutate a part of the system can invalidate another part.

This can also happen when there is a need for forceful termination / cancellation.

Avoiding shared mutable state (if possible), ensuring appropriate encapsulation of things that do have access to this shared mutable state, can help in making a program more concurrent.

C++20 Coroutines

- Coroutines can help us to take care of the concurrent structure of the application

- When we have an appropriate structure in place we can scale the execution of this application with the parallelism at our disposal

- Doing so will allow us to make execution efficient, while allowing for scaling according to the number of threads our hardware provides, while also maintaining readability and synchronous-like style of our code

Can I just use a library?

Please say yes.

Can I just use a library?

Please say yes.

- Cppcoro
- Facebook/folly/coro

References

- https://lewissbaker.github.io/2017/09/25/coroutine-theory
- https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await
- https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type
- https://www.youtube.com/watch?v=ZTqHjjm86Bw – James McNellis Cppcon 2016 talk
- https://blog.panicsoftware.com/coroutines-introduction/
- https://blog.panicsoftware.com/your-first-coroutine/
- https://blog.panicsoftware.com/co_awaiting-coroutines/
- http://eel.is/c++draft/dcl.fct.def#coroutine
- https://www.youtube.com/watch?v=7hkqG8i0QaU – Anthony Williams ACCU 2019 talk
- https://github.com/lewissbaker/cppcoro
- https://blog.varunramesh.net/posts/stackless-vs-stackful-coroutines/

Thank you