CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Coroutines in a bit more than a nutshell

Péter Kardos

Suspendable functions

Threads

```
std::future<int> compute(std::future<int> input) {
    // Start function on thread A
    int value = input.get(); // Suspend thread A
    // Resume function on thread A
    return std::async([=]{ return f(value); });
}
```

Coroutines

```
task<int> compute(task<int> input) {
    // Start coroutine on thread A
    int value = co_await input; // Suspend coroutine
    // Resume coroutine on thread B
    co_return f(value);
}
```

- Suspendable functions are not new
- Threads do get suspended in the middle of a function call
- They do pick up where they left off once resumed by the OS
- Coroutines do the same thing essentially, but have different characteristics

Coroutines in C++

How to use C++ coroutines?

1. Decide you need coroutines
2. Sit down and code your own coroutine library
3. Write thousands of lines of code
4. Debug concurrency errors
5. ???
6. Profit

Why so complicated?

- You can't simply use coroutines out of the box
- Downside: `task<T>` and the like have to be implemented by you
- Upside: `task<T>` and the like can be implemented in any way you want it

Compiler support for coroutines

Coroutine body:

```
// Definition:
task<int> my_first_coro() {
    // Body here...
    co_return 1;
}

// Call
task<int> t = my_first_coro();
```

- This is how your code looks like
- You think about this when using a coroutine library

Under the hood:

```
// Definition:
task<int> my_first_coro(
    typename task<int>::promise_type& promise
) {
    try {
        co_await promise.initial_suspend();
        // Body here...
        promise.return_value(1);
    }
    catch (...) {
        promise.unhandled_exception();
    }
final_suspend:
    co_await promise.final_suspend()
    delete &promise;
}

// Call
auto promise = new typename task<int>::promise_type;
task<int> t = promise->get_return_object();
my_first_coro(*promise);
```

- This is how the compiler interacts with your code
- You think about this when implementing `task<T>`

CSCS

ETH zürich

Making `task<T>` and `promise<T>`

- From the under-the-hood view, we can figure out the layout of both
- These are the minimum methods they have to implement

The `task<T>`

```
template <class T>
class task {
public:
    using promise_type = promise<T>
}
```

The `promise<T>`

```
template <class T>
struct promise {
    task<T> get_return_object();
    auto initial_suspend() const noexcept;
    auto final_suspend() const noexcept;
    void return_value(T value) noexcept;
    void unhandled_exception() noexcept;
}
```

Defining `promise<T>::get_return_object`

- This method returns the `task<T>` object
- The returned `task` should most likely know about the promise
- Thus we'll pass the `this` pointer to the `task`

```
template <class T>
task<T> promise<T>::get_return_object() noexcept {
    return task<T>(this);
}
```

Defining `promise<T>::initial_suspend`

- Coroutines can essentially start suspended or start running
- The behavior is determined by `promise::initial_suspend`
- Remember how each coroutine body starts with `co_await promise.initial_suspend()`?
  - This can either suspend or continue the coroutine
- In our case, we want to always suspend the coroutine after starting:

```
template <class T>
auto promise<T>::initial_suspend() const noexcept {
    return std::suspend_always{};
}
```

- `std::suspend_always` is a helper class
- It can be `co_await`ed -- we'll soon see what that means

Defining `promise<T>::final_suspend`

- When coroutines finish they have two options:
  - Suspend the coroutine: it's final state can be inspected
  - Continue the coroutine: this also destroys the coroutine
  - The behavior is determined by `promise::final_suspend`
  - This happens immediately after the `co_return` statement
- We want our coroutines to always suspend on finish
- This is because we want to retrieve the results
- If the coroutine is destroyed, we can't get the results anymore
  - Unless the coroutine forwards it before it's destroyed
  - But we won't take that approach for simplicity

```
template <class T>
auto promise<T>::final_suspend() const noexcept {
    return std::suspend_always{};
}
```

CSCS

ETH zürich

Defining `promise<T>::return_value`

- The statement `co_return X;`
- Translates into `promise.return_value(X);`
- Essentially `return_value` is our chance to store the value returned by the coroutine
- We'll save it into a field of the `promise` object

```cpp
template <class T>
auto promise<T>::return_value(T value) const noexcept {
    m_result = std::move(value);
}
```

For this, we need to modify the promise too by adding the `m_result` field:

```cpp
template <class T>
struct promise {
    ...
    T get_result() noexcept { return std::move(m_result.value); }
    std::optional<T> m_result;
}
```

Defining `promise<T>::unhandled_exception`

- This is called when instead of `co_return`, we exit the scope because of an exception
- We can call `std::current_exception` to store the exception
- Then later use `std::rethrow_exception` when someone tries to retrieve the results
- But for now, we'll just terminate the application:

```
template <class T>
auto promise<T>::unhandled_exception() const noexcept {
    std::terminate();
}
```

Summary of the implementation so far (1)

We can now write this and it compiles:

```
task<int> my_first_coro() {
    co_return 1;
}
```

Despite all the work it still has a few shortcomings:

- The coroutine body never runs
- The coroutine stack never gets deleted - it leaks memory
- No way to retrieve the results
    - We cannot `co_await` this coroutine yet
- No way to synchronize the results
    - We cannot obtain the results from a regular function either

Getting the results by synchronization

Let's add a `get` method to the task, similarly to `std::future`:

```
template <class T>
T task<T>::get() {
    // Logic here...
}
```

Regarding the logic:

- The coroutine is initially suspended
- The first thing we want to do is resume it
  - Otherwise it will never `co_return` us the result
- The second is to retrieve the result using the `promise<T>::get_result` we wrote earlier
- Finally, we pass the result on to the caller

CSCS ETH zürich

1. Resuming the coroutine: the coroutine handle

For this, we will need the so-called *coroutine handle*:

```cpp
template <class T>
auto promise<T>::handle() noexcept {
    return std::coroutine_handle<promise>::from_promise(*this);
}
```

What is this handle anyway?

- When you create a coroutine (i.e. `my_first_coro()`), its promise and local variables get allocated on the heap
- This is exactly the same as a function's stack frame
  - Only that `RBP` and `RSP` now point to an arbitrary address
  - Instead of `SUB RBP, $s` now you have `MOV RBP, $coro_frame`
- This way the coroutine's stack frame can outlive its caller
- The `std::coroutine_handle` is just a pointer to the coroutine's stack frame

1. Resuming the coroutine: resume method

Now that we have access to the coroutine handle, we can use it to resume a suspended coroutine:

```
template <class T>
T task<T>::get() noexcept {
    m_promise->handle().resume();
    // TODO: get result
    // TODO: forward result to caller
}
```

- **WARNING:** resuming a running coroutine is undefined behavior!
- We don't have to worry about this:
  - The `task` instance is the sole owner of its `promise` instance (i.e. `task` is not `CopyConstructible`)
  - The coroutine is always suspended initially
  - The coroutine is only ever resumed in `get`

2. & 3. Getting the result

The rest is very simple:

```
template <class T>
T task<T>::get() noexcept {
    m_promise->handle().resume();
    return m_promise->get_result();
}
```

Summary of the implementation so far (2)

Now we can write a coroutine as well as get its result:

```
task<int> my_first_coro() {
    co_return 1;
}

int main() {
    auto result = my_first_coro();
    const auto value = result.get();
    std::cout << value << std::endl;
}
```

However:

- It's still leaking memory
- We still cannot `co_await` the `task`

The `co_await` expression

What you write:

```
task<int> my_second_coro() {
    const int value = co_await my_first_coro();
    co_return value;
}
```

- **NOTE:** you can only use `co_await` inside a coroutine, thus `my_second_coro` is also a coroutine

Under the hood:

```
task<T> my_second_coro() {
    auto&& task = my_first_coro();
    auto&& awaitable = task.operator co_await();
    bool suspend = awaitable.await_ready();
    if (suspend) {
        // Magic: coroutine is now suspended.
        //  await_suspend is still called.
        awaitable.await_suspend(handle);
        // Magic: control returned to caller
        //  of handle.resume().
    }
    // Magic: someone called handle.resume() again.
    //  Coroutine continues here.
    int value = awaitable.await_resume();
}
```

CSCS

ETH zürich

Making `task<T>` awaitable

From the under the hood picture, we can figure out the necessary methods:

- `task<T>` must have an `operator co_await`. Let's take an educated guess that this method returns an awaitable object that needs to know about the `promise<T>` too:

```
template <class T>
awaitable<T> task<T>::operator co_await() noexcept {
    return awaitable<T>(m_promise);
}
```

- According to the `co_await` expression's expanded view, `awaitable<T>` must have this declaration:

```
template <class T>
struct awaitable {
    promise<T>* m_promise;
    bool await_ready() const noexcept;
    void await_suspend(std::coroutine_handle<>) const noexcept;
    T await_resume() const noexcept;
};
```

Defining `awaitable<T>::await_ready`

The meaning of this function (*"is ready?"*):

- If `await_ready` returns `true`:
  - The enclosing coroutine is continued without suspension
  - `await_resume` is called immediately after
- If `await_ready` returns `false`:
  - The enclosing coroutine is suspended immediately
  - `await_suspend` is called immediately after suspension

We'll never suspend coroutines, we'll resume the awaited ones instead:

```
template <class T>
bool awaitable<T>::await_ready() const noexcept {
    m_promise->handle().resume();
    return true;
}
```

Defining `awaitable<T>::await_suspend`

The meaning of this function (*"on suspend" / "do suspend?"*):

- Called when `await_ready` returns false ==> for us, it'll never be called
- Its argument is the enclosing coroutine: the one that's `co_await`-ing the `task<T>` that returned this `awaitable<T>`
  - We could save the enclosing coroutine and resume it at a later time
- Its return value may be
  - `void`: in this case, the enclosing coroutine stays suspended
  - `bool`: in this case, even though `await_ready` caused the enclosing coro to suspend, we can decide to rather resume it right away by returning `false`
  - `std::coroutine_handle<void>`: in this case, the enclosing coroutine stays suspended, but we resume will be called on the returned handle.

```
template <class T>
void awaitable<T>::await_suspend(std::coroutine_handle<>) const noexcept {}
```

Defining `awaitable<T>::await_resume`

The meaning of this function (*"on resume"*):

- Called when the enclosing coroutine is resumed
  - This can happen immediately when `await_ready` returns `true`
  - Or asynchronously in the future
- Provides the type and value of the `co_await` expression

Our implementation returns the result of the `task<T>` this `awaitable<T>` belongs to:

```
template <class T>
T awaitable<T>::await_resume() const noexcept {
    return m_promise->get_result();
}
```

CSCS

ETH zürich

Patching that memory leak

- The coroutine is always suspended when it has finished
- It's safe to destroy a suspended coroutine
    - Destroying a running coroutine is certainly undefined behavior
- Let's take care of it in the destructor of `task<T>`
- We can use the coroutine handle

```
template <class T>
task<T>::~task() {
    m_promise->handle().destroy();
}
```

Now the coroutine's stack frame on the heap is properly freed.

Summary of the implementation so far (3)

Now we can also `co_await` coroutines:

```
task<int> my_first_coro() {
    co_return 1;
}

task<int> my_second_coro() {
    const int value = co_await my_first_coro();
    co_return value;
}

int main() {
    auto result = my_second_coro();
    const auto value = result.get();
    std::cout << value << std::endl;
}
```

- Wow, it's useless!
- We've just reimplemented plain old functions in a complicated way

CSCS

ETH zürich

Going async

- The issue: all our coroutines execute synchronously
- We need to change the implementation of `await_ready` and `await_suspend`
- Combining coroutines with other event sources:
  - We can offload computation of a coroutine to another thread and `co_await` or `get()` it elsewhere
  - We can make a coroutine resume only once an I/O operation is finished
  - We can make a coroutine resume on other operations such as DB queries, HTTP requests, etc.
- **MOST IMPORTANT POINT**:
  - This only changes the implementation of `task<T>` and similar primitives
  - The syntax to use them stays the exact same
    - Which is currently the same syntax as regular functions
    - Thus our async code will look like the usual sync code (not bad!)
  - This is the main motivation behind coroutines

Syntax of async code

Future-then pattern

```
std::future<bowl&> make_dough(bowl& b) {
    return get_flour()
        .then([&b](ingredient flour){
            b.add(flour);
            return get_water();
        })
        .then([&b](ingredient water){
            b.add(water);
            return get_milk();
        })
        .then([&b](ingredient milk){
            b.add(milk);
            return get_eggs();
        })
        .then([&b](ingredient eggs){
            b.add(eggs);
            return b;
        });
}
```

- This looks pretty disastrous
- You don't want your pancakes to be full of bugs

Coroutine pattern

```
task<bowl&> make_dough(bowl& b) {
    b.add(co_await get_flour());
    b.add(co_await get_water());
    b.add(co_await get_milk());
    b.add(co_await get_eggs());
    co_return b;
}
```

- This looks pretty
  - Basically the same as blocking code
- Much more likely to be free of bugs

CSCS

ETH zürich

Creating a coroutine library

We have seen:

- `task<T>`

But there is also:

- `generator<T>`
- `stream<T>`
- `shared_task<T>`
- `mutex`
- `event`
- `fence`
- `semaphore`
- ...
- You're essentially free to implement whatever you want

Example: networking with coroutines (1)

```
class socket {
    struct awaitable {
        bool await_ready() {
            return poll(m_fd, 0);
        }
        void await_suspend(std::coroutine_handle<> handle) {
            network_scheduler::enqueue(m_fd, handle);
        }
        std::vector<std::byte> await_resume() {
            return recv(m_fd);
        }
    }
public:
    void send(std::span<const std::byte> data);
    auto operator co_await() {
        return awaitable(m_fd);
    }
private:
    int m_fd;
}
```

- `class socket` can be an awaitable
  - No need for a coroutine promise
- `network_scheduler` has a background thread that does the polling
- `co_await`ing a `socket` simply adds the socket to the polled sockets
- If data is available, the `network_scheduler` calls `resume` on the coroutine handle associated with that socket

Example: networking with coroutines (2)

Why coroutines?

- Reducing synchronization overhead
  - Imagine 1000s of connections
  - The kernel has to switch between 1000s of thread: lot of processing
  - Coroutines on a thread-pool hardly use CPU cycles in comparison
- Reducing resource allocation overhead
  - Each thread has a stack and kernel data structures allocated
  - Creating and destroying threads is expensive
  - Coroutines are lightweight and you can have millions of them alive at the same time
- Your syntax is still pretty much the same as single-threaded blocking code

Mini case study: game engine job system (1)

Everything a game engine does to update the game can be broken down into smaller tasks:

- Render scene
  - Frustum culling (batch & parallelize)
  - Rendering (batch & parallelize)
  - Planar reflections (batch & parallelize)
  - Shadow maps (batch & parallelize)
  - Post processing
- Timestep physics
  - Space partitioning
  - Collision detection
  - Forces & integration
- Sound...
- UI...
- Scripts...

Mini case study: game engine job system (2)

- Game engines have strong performance requirements:
  - High throughput
  - Low latency
  - Both are very important
- Throughput: needs to spread work across CPU cores
- Latency: needs to keep synchronization overhead small even when CPU cores are not saturated
- Solution: *job system*
  - Often implemented with fibers (stackful coroutines)
  - The tasks to update the frame are organized into a DAG
  - The tasks are converted to fibers
  - And scheduled on a thread pool

Mini case study: game engine job system (3)

With C++ coroutines, you could write it like this:

```
task<void> update_frame() {
    auto task_scripts = launch(exec_scripts(), thread_pool);
    auto task_graphics = launch(render_scene(), thread_pool);
    auto task_physics = launch(update_physics(), thread_pool);
    auto task_sound = launch(mix_sound(), thread_pool);
    auto task_ui = launch(update_ui(), thread_pool);
    co_await task_scripts;
    co_await task_graphics;
    co_await task_physics;
    co_await task_sound;
    co_await task_ui;
    co_return;
}
```

- The code looks linear, but is fully parallelized
- The task DAG is now the same as the call graph: you don't need explicit job objects and graphs
- Low overhead: you can split jobs as much as you can to help parallelization

Performance comparison with threads

- Threads are expensive:
  - They have 1 MiB stack allocated each
  - Synchonization requires OS kernel calls
  - Suspension involves the OS scheduler
  - Creation and destruction is expensive

- Coroutines are cheap:
  - They are stackless
  - Synchronization happens in userspace: uses cheap atomics or spinlocks
  - Suspension is just saving some registers
  - Creation and destruction is just a new/delete

- Number of threads my PC can finish in a second:
  - Linux: 54k/s
  - Windows: 53k/s
- Number of coroutines my PC can finish in a second:
  - Linux: 163M/s
  - Windows: 27M/s

Notes on performance comparison

- Apples to apples?
  - You could use a thread pool without coroutines
  - You could use a static task graph (coroutines are always dynamic!)
  - You could use TBB or a similar high performance library
  - It would be as fast as coroutines, maybe faster
  - You could also add better memory allocators for the coroutine frames
- What we measured:
  - Cost of thread setup vs cost of coroutine setup
  - In that sense it's a fair comparison
- Conclusion:
  - You have to do a lot more work to get threads up to speed
  - Coroutines stay ergonomic despite their high performance

Remarks

- As usual, coroutines are difficult, like everything else in C++
- As usual, coroutines are powerful, like... many things in C++
- You need a good coroutine library to make use of the feature
- Coroutines can substantially improve the code quality and performance of certain applications