C++ Coroutines: Understanding Symmetric Transfer

May 11, 2020

The Coroutines TS provided a wonderful way to write asynchronous code as if you were writing synchronous code. You just need to sprinkle `co_await` at appropriate points and the compiler takes care of suspending the coroutine, preserving state across suspend-points and resuming execution of the coroutine later when the operation completes.

However, the Coroutines TS, as it was originally specified, had a nasty limitation that could easily lead to stack-overflow if you weren't careful. And if you wanted to avoid this stack-overflow then you had to introduce extra synchronisation overhead to safely guard against this in your `task<T>` type.

Thankfully, a tweak was made to the design of coroutines in 2018 to add a capability called "symmetric transfer" which allows you to suspend one coroutine and resume another coroutine without consuming any additional stack-space. The addition of this capability lifted a key limitation of the Coroutines TS and allows for much simpler and more efficient implementation of async coroutine types without sacrificing any of the safety aspects needed to guard against stack-overflow.

In this post I will attempt to explain the stack-overflow problem and how the addition of this key "symmetric transfer" capability lets us solve this problem.

First some background on how a task coroutine works

Consider the following coroutines:

```
task foo() {
  co_return;
}

task bar() {
```

```
  co_await foo();
}
```

Assume we have a simple `task` type that lazily executes the body when another coroutine awaits it. This particular `task` type does not support returning a value.

Let's unpack what's happening here when `bar()` evaluates `co_await foo()`.

- The `bar()` coroutine calls the `foo()` function. Note that from the caller's perspective a coroutine is just an ordinary function.
- The invocation of `foo()` performs a few steps:
  - Allocates storage for a coroutine frame (typically on the heap)
  - Copies parameters into the coroutine frame (in this case there are no parameters so this is a no-op).
  - Constructs the promise object in the coroutine frame
  - Calls `promise.get_return_object()` to get the return-value for `foo()`. This produces the `task` object that will be returned, initialising it with a `std::coroutine_handle` that refers to the coroutine frame that was just created.
  - Suspends execution of the coroutine at the initial-suspend point (ie. the open curly brace)
  - Returns the `task` object back to `bar()`.
- Next the `bar()` coroutine evaluates the `co_await` expression on the `task` returned from `foo()`.
  - The `bar()` coroutine suspends and then calls the `await_suspend()` method on the returned task, passing it the `std::coroutine_handle` that refers to `bar()`'s coroutine frame.
  - The `await_suspend()` method then stores `bar()`'s `std::coroutine_handle` in `foo()`'s promise object and then resumes the `foo()` coroutine by calling `.resume()` on `foo()`'s `std::coroutine_handle`.
- The `foo()` coroutine executes and runs to completion synchronously.
- The `foo()` coroutine suspends at the final-suspend point (ie. the closing curly brace) and then resumes the coroutine identified by the `std::coroutine_handle` that was stored in its promise object before it was started. ie. `bar()`'s coroutine.
- The `bar()` coroutine resumes and continues executing and eventually reaches the end of the statement containing the `co_await` expression at which point it calls the destructor of the temporary `task` object returned from `foo()`.
- The `task` destructor then calls the `.destroy()` method on `foo()`'s coroutine handle which then destroys the coroutine frame along with the promise object and copies of any arguments.

Ok, so that seems like a lot of steps for a simple call.

To help understand this in a bit more depth, let's look at how a naive implementation of this `task` class would look when implemented using the the Coroutines TS design (which didn't support symmetric transfer).

Outline of a `task` implementation

The outline of the class looks something like this:

```
class task {
public:
  class promise_type { /* see below */ };

  task(task&& t) noexcept
  : coro_(std::exchange(t.coro_, {}))
  {}

  ~task() {
    if (coro_)
      coro_.destroy();
  }

  class awaiter { /* see below */ };

  awaiter operator co_await() && noexcept;

private:
  explicit task(std::coroutine_handle<promise_type> h) noexcept
  : coro_(h)
  {}

  std::coroutine_handle<promise_type> coro_;
};
```

A `task` has exclusive ownership of the `std::coroutine_handle` that corresponds to the coroutine frame created during the invocation of the coroutine. The `task` object is an RAII object that ensures that `.destroy()` is called on the `std::coroutine_handle` when the `task` object goes out of scope.

So now let's expand on the `promise_type`.

Implementing `task::promise_type`

From the previous post we know that the `promise_type` member defines the type of the **Promise** object that is created within the coroutine frame and that controls the behaviour of the coroutine.

First, we need to implement the `get_return_object()` to construct the `task` object to return when the coroutine is invoked. This method just needs to initialise the task with the `std::coroutine_handle` of the newly create coroutine frame.

We can use the `std::coroutine_handle::from_promise()` method to manufacture one of these handles from the promise object.

```
class task::promise_type {
public:
  task get_return_object() noexcept {
    return task{std::coroutine_handle<promise_type>::from_promise(*this)};
  }
```

Next, we want the coroutine to initially suspend at the open curly brace so that we can later resume the coroutine from this point when the returned `task` is awaited.

There are several benefits of starting the coroutine lazily:

1. It means that we can attach the continuation's `std::coroutine_handle` before starting execution of the coroutine. This means we don't need to use thread-synchronisation to arbitrate the race between attaching the continuation later and the coroutine running to completion.
2. It means that the `task` destructor can unconditionally destroy the coroutine frame - we don't need to worry about whether the coroutine is potentially executing on another thread since the coroutine will not start executing until we await it, and while it is executing the calling coroutine is suspended and so won't attempt to call the task destructor until the coroutine finishes executing. This gives the compiler a much better chance at inlining the allocation of the coroutine frame into the frame of the caller. See P0981R0 to read more about the Heap Allocation eLision Optimisation (HALO).
3. It also improves the exception-safety of your coroutine code. If you don't immediately `co_await` the returned `task` and do something else that can throw an exception that causes the stack to unwind and the `task` destructor to run then we can safely destroy the coroutine since we know it hasn't started yet. We aren't left with the difficult choice between detaching, potentially leaving dangling references, blocking in the destructor, terminating or undefined-behaviour. This is something that I cover in a bit more detail in my CppCon 2019 talk on Structured Concurrency.

To have the coroutine initially suspend at the open curly brace we define an `initial_suspend()` method that returns the builtin `suspend_always` type.

```
std::suspend_always initial_suspend() noexcept {
  return {};
}
```

Next, we need to define the `return_void()` method, called when you execute `co_return;` or when execution runs off the end of the coroutine. This method doesn't actually need to do anything, it just needs to exist so that the compiler knows that `co_return;` is valid within this coroutine type.

```
void return_void() noexcept {}
```

We also need to add an `unhandled_exception()` method that is called if an exception escapes the body of the coroutine. For our purposes we can just treat the task coroutine bodies as `noexcept` and call `std::terminate()` if this happens.

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

Finally, when the coroutine execution reaches the closing curly brace, we want the coroutine to suspend at the final-suspend point and then resume its continuation. ie. the coroutine that is awaiting the completion of this coroutine.

To support this, we need a data-member in the promise to hold the `std::coroutine_handle` of the continuation. We also need to define the `final_suspend()` method that returns an awaitable object that will resume the continuation after the current coroutine has suspended at the final-suspend point.

It's important to delay resuming the continuation until after the current coroutine has suspended because the continuation may go on to immediately call the `task` destructor which will call `.destroy()` on the coroutine frame. The `.destroy()` method is only valid to call on a suspended coroutine and so it would be undefined-behaviour to resume the continuation before the current coroutine has suspended.

The compiler inserts code to evaluate the statement `co_await promise.final_suspend();` at the closing curly brace.

It's important to note that the coroutine is not yet in a suspended state when the `final_suspend()` method is invoked. We need to wait until the `await_suspend()` method on the returned awaitable is called before the coroutine is suspended.

```
  struct final_awaiter {
    bool await_ready() noexcept {
      return false;
    }

    void await_suspend(std::coroutine_handle<promise_type> h) noexcept {
      // The coroutine is now suspended at the final-suspend point.
      // Lookup its continuation in the promise and resume it.
      h.promise().continuation.resume();
    }

    void await_resume() noexcept {}
  };

  final_awaiter final_suspend() noexcept {
    return {};
  }

  std::coroutine_handle<> continuation;
};
```

Ok, so that's the complete `promise_type`. The final piece we need to implement is the `task::operator co_await()`.

Implementing `task::operator co_await()`

You may remember from the Understanding operator co_await() post that when evaluating a `co_await` expression, the compiler will generate a call to `operator co_await()`, if one is defined, and then the object returned must have the `await_ready()`, `await_suspend()` and `await_resume()` methods defined.

When a coroutine awaits a `task` we want the awaiting coroutine to always suspend and then, once it has suspended, store the awaiting coroutine's handle in the promise of the coroutine we are about to resume and then call `.resume()` on the `task`'s `std::coroutine_handle` to start executing the task.

Thus the relatively straight forward code:

```
class task::awaiter {
public:
  bool await_ready() noexcept {
    return false;
  }
```

```
  void await_suspend(std::coroutine_handle<> continuation) noexcept {
    // Store the continuation in the task's promise so that the final_suspend()
    // knows to resume this coroutine when the task completes.
    coro_.promise().continuation = continuation;

    // Then we resume the task's coroutine, which is currently suspended
    // at the initial-suspend-point (ie. at the open curly brace).
    coro_.resume();
  }

  void await_resume() noexcept {}

private:
  explicit awaiter(std::coroutine_handle<task::promise_type> h) noexcept
  : coro_(h)
  {}

  std::coroutine_handle<task::promise_type> coro_;
};

task::awaiter task::operator co_await() && noexcept {
  return awaiter{coro_};
}
```

And thus completes the code necessary for a functional `task` type.

You can see the complete set of code in Compiler Explorer here: https://godbolt.org/z/-Kw6Nf

The stack-overflow problem

The limitation of this implementation arises, however, when you start writing loops within your coroutines and you `co_await` tasks that can potentially complete synchronously within the body of that loop.

For example:

```
task completes_synchronously() {
  co_return;
}

task loop_synchronously(int count) {
  for (int i = 0; i < count; ++i) {
```

```
    co_await completes_synchronously();
  }
}
```

With the naive `task` implementation described above, the `loop_synchronously()` function will (probably) work fine when `count` is 10, 1000, or even 100'000. But there will be a value that you can pass that will eventually cause this coroutine to start crashing.

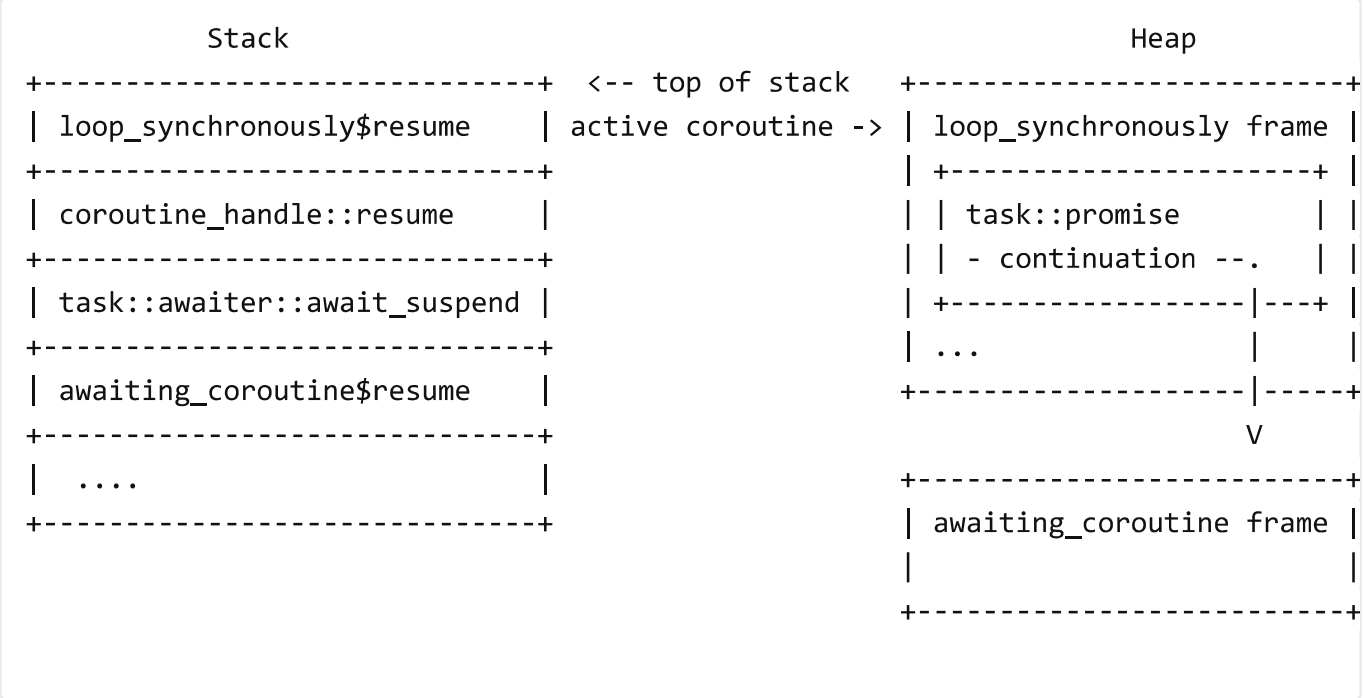
For example, see: https://godbolt.org/z/gy5Q8q which crashes when `count` is 1'000'000.

The reason that this is crashing is because of stack-overflow.

To understand why this code is causing a stack-overflow we need to take a look at what is happening when this code is executing. In particular, what is happening to the stack-frames.

When the `loop_synchronously()` coroutine first starts executing it will be because some other coroutine `co_await`ed the `task` returned. This will in turn suspend the awaiting coroutine and call `task::awaiter::await_suspend()` which will call `resume()` on the task's `std::coroutine_handle`.

Thus the stack will look something like this when `loop_synchronously()` starts:



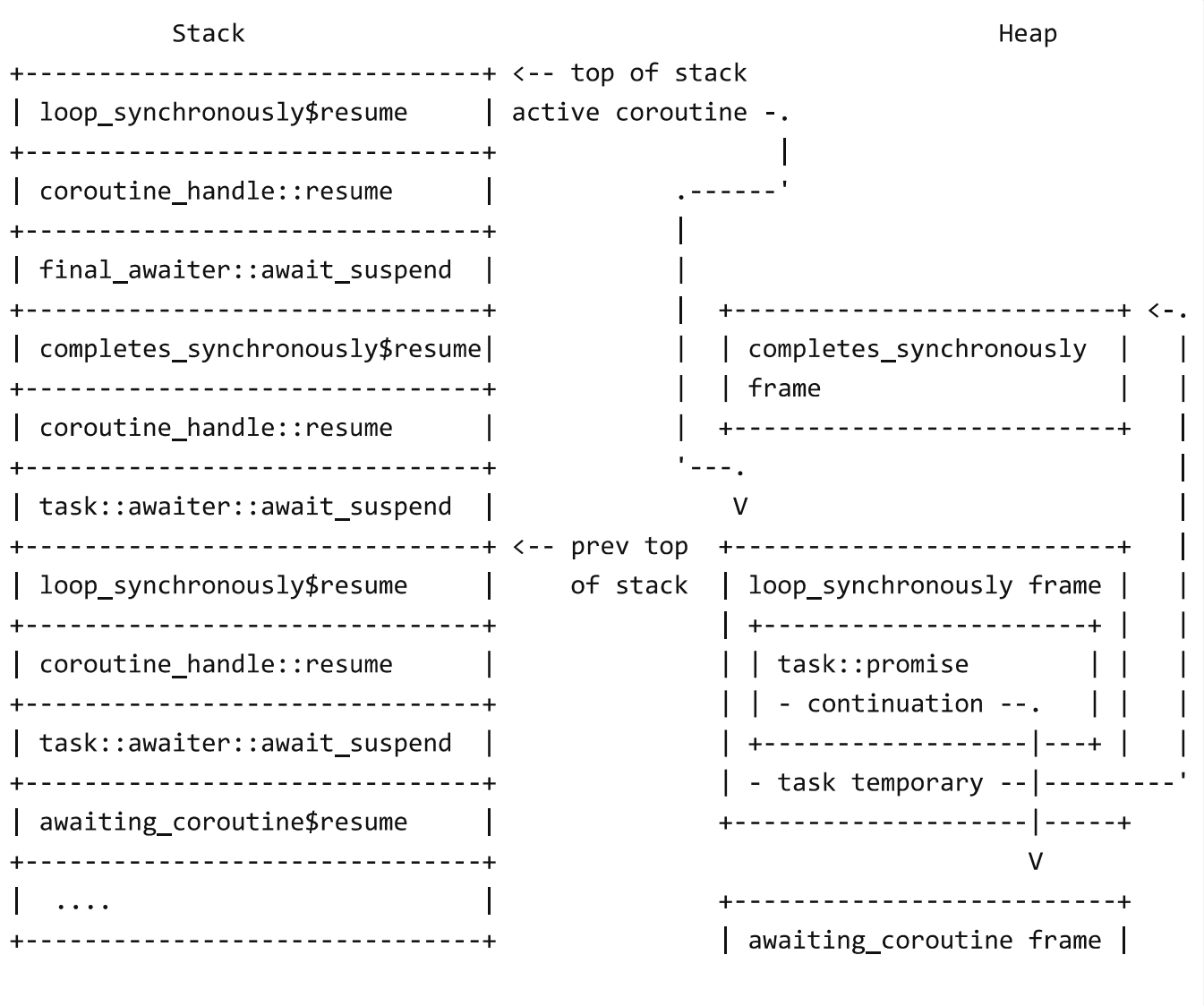
> *Note: When a coroutine function is compiled the compiler typically splits it into two parts:*
>
> 1. *the "ramp function" which deals with the construction of the coroutine frame, parameter copying, promise construction and producing the return-value, and*
> 2. *the "coroutine body" which contains the user-authored logic from the body of the coroutine.*

> *I use the `$resume` suffix to refer to the "coroutine body" part of the coroutine.*
>
> *A later blog post will go into more detail about this split.*

Then when `loop_synchronously()` awaits the `task` returned from `completes_synchronously()` the current coroutine is suspended and calls `task::awaiter::await_suspend()`. The `await_suspend()` method then calls `.resume()` on the coroutine handle corresponding to the `completes_synchronously()` coroutine.

This resumes the `completes_synchronously()` coroutine which then runs to completion synchronously and suspends at the final-suspend point. It then calls `task::promise::final_awaiter::await_suspend()` which calls `.resume()` on the coroutine handle corresponding to `loop_synchronously()`.

The net result of all of this is that if we look at the state of the program just after the `loop_synchronously()` coroutine is resumed and just before the temporary `task` returned by `completes_synchronously()` is destroyed at the semicolon then the stack/heap should look something like this:



```
                                                  |                          |
                                                  +--------------------------+
```

Then the next thing this will do is call the `task` destructor which will destroy the `completes_synchronously()` frame. It will then increment the `count` variable and go around the loop again, creating a new `completes_synchronously()` frame and resuming it.

In effect, what is happening here is that `loop_synchronously()` and `completes_synchronously()` end up recursively calling each other. Each time this happens we end up consuming a bit more stack-space, until eventually, after enough iterations, we overflow the stack and end up in undefined-behaviour land, typically resulting in your program promptly crashing.

Writing loops in coroutines built this way makes it very easy to write functions that perform unbounded recursion without looking like they are doing any recursion.

So, what would the solution look like under the original Coroutines TS design?

The Coroutines TS solution

Ok, so what can we do about this to avoid this kind of unbounded recursion?

With the above implementation we are using the variant of `await_suspend()` that returns `void`. In the Coroutines TS there is also a version of `await_suspend()` that returns `bool` - if it returns `true` then the coroutine is suspended and execution returns to the caller of `resume()`, otherwise if it returns `false` then the coroutine is immediately resumed, but this time without consuming any additional stack-space.

So, to avoid the unbounded mutual recursion what we want to do is make use of the `bool`-returning version of `await_suspend()` to resume the current coroutine by returning `false` from the `task::awaiter::await_suspend()` method if the task completes synchronously instead of resuming the coroutine recursively using `std::coroutine_handle::resume()`.

To implement a general solution for this there are two parts.

1. Inside the `task::awaiter::await_suspend()` method you can start executing the coroutine by calling `.resume()`. Then when the call to `.resume()` returns, check whether the coroutine has run to completion or not. If it has run to completion then we can return `false`, which indicates the awaiting coroutine should immediately resume, or we can return `true`, indicating that execution should return to the caller of `std::coroutine_handle::resume()`.

2. Inside `task::promise_type::final_awaiter::await_suspend()`, which is run when the coroutine runs to completion, we need to check whether the awaiting coroutine has (or will) return `true` from `task::awaiter::await_suspend()` and if so then resume it by calling `.resume()`. Otherwise, we need to avoid resuming the coroutine and notify `task::awaiter::await_suspend()` that it needs to return `false`.

There is an added complication, however, in that it's possible for a coroutine to start executing on the current thread then suspend and later resume and run to completion on a different thread before the call to `.resume()` returns. Thus, we need to be able to resolve the potential race between part 1 and part 2 above happening concurrently.

We will need to use a `std::atomic` value to decide the winner of the race here.

Now for the code. We can make the following modifications:

```
class task::promise_type {
  ...

  std::coroutine_handle<> continuation;
  std::atomic<bool> ready = false;
};

bool task::awaiter::await_suspend(
    std::coroutine_handle<> continuation) noexcept {
  promise_type& promise = coro_.promise();
  promise.continuation = continuation;
  coro_.resume();
  return !promise.ready.exchange(true, std::memory_order_acq_rel);
}

void task::promise_type::final_awaiter::await_suspend(
    std::coroutine_handle<promise_type> h) noexcept {
  promise_type& promise = h.promise();
  if (promise.ready.exchange(true, std::memory_order_acq_rel)) {
    // The coroutine did not complete synchronously, resume it here.
    promise.continuation.resume();
  }
}
```

See the updated example on Compiler Explorer: https://godbolt.org/z/7fm8Za Note how it no longer crashes when executing the `count == 1'000'000` case.

This turns out to be the approach that the `cppcoro::task<T>` implementation took to avoid the unbounded recursion problem (and still does for some platforms) and it has worked

reasonably well.

Woohoo! Problem solved, right? Ship it! Right...?

The problems

While the above solution does solve the recursion problem it has a couple of drawbacks.

Firstly, it introduces the need for `std::atomic` operations which can be quite costly. There is an atomic exchange on the caller when suspending the awaiting coroutine, and another atomic exchange on the callee when it runs to completion. If your application only ever executes on a single thread then you are paying the cost of the atomic operations for synchronising threads even though it's never needed.

Secondly, it introduces additional branches. One in the caller, which needs to decide whether to suspend or immediately resume the coroutine, and one in the callee, which needs to decide whether to resume the continuation or suspend.

Note that the cost of this extra branch, and possibly even the atomic operations, would often be dwarfed by the cost of the business logic present in the coroutine. However, coroutines have been advertised as a zero cost abstraction and there have even been people using coroutines to suspend execution of a function to avoid waiting for an L1-cache-miss (see Gor's great CppCon talk on nanocoroutines for more details on this).

Thirdly, and probably most importantly, it introduces some non-determinism in the execution-context that the awaiting coroutine resumes on.

Let's say I have the following code:

```
cppcoro::static_thread_pool tp;

task foo()
{
  std::cout << "foo1 " << std::this_thread::get_id() << "\n";
  // Suspend coroutine and reschedule onto thread-pool thread.
  co_await tp.schedule();
  std::cout << "foo2 " << std::this_thread::get_id() << "\n";
}

task bar()
{
  std::cout << "bar1 " << std::this_thread::get_id() << "\n";
  co_await foo();
```

```
  std::cout << "bar2" << std::this_thread::get_id() << "\n";
}
```

With the original implementation we were guaranteed that the code that runs after `co_await foo()` would run inline on the same thread that `foo()` completed on.

For example, one possible output would have been:

```
bar1 1234
foo1 1234
foo2 3456
bar2 3456
```

However, with the changes to use atomics, it's possible the completion of `foo()` may race with the suspension of `bar()` and this can, in some cases, mean that the code after `co_await foo()` might run on the original thread that `bar()` started executing on.

For example, the following output might now also be possible:

```
bar1 1234
foo1 1234
foo2 3456
bar2 1234
```

For many use-cases this behaviour may not make a difference. However, for algorithms whose purpose is to transition execution context this can be problematic.

For example, the `via()` algorithm awaits some Awaitable and then produces it on the specified scheduler's execution context. A simplified version of this algorithm is shown below.

```
template<typename Awaitable, typename Scheduler>
task<await_result_t<Awaitable>> via(Awaitable a, Scheduler s)
{
  auto result = co_await std::move(a);
  co_await s.schedule();
  co_return result;
}

task<T> get_value();
void consume(const T&);

task<void> consumer(static_thread_pool::scheduler s)
{
```

```
  T result = co_await via(get_value(), s);
  consume(result);
}
```

With the original version the call to `consume()` is always guaranteed to be executed on the thread-pool, `s`. However, with the revised version that uses atomics it's possible that `consume()` might either be executed on a thread associated with the scheduler, `s`, or on whatever thread the `consumer()` coroutine started execution on.

So how do we solve the stack-overflow problem without the overhead of the atomic operations, extra branches and the non-deterministic resumption context?

Enter "symmetric transfer"!

The paper P0913R0 "Add symmetric coroutine control transfer" by Gor Nishanov (2018) proposed a solution to this problem by providing a facility which allows one coroutine to suspend and then resume another coroutine symmetrically without consuming any additional stack-space.

This paper proposed two key changes:

- Allow returning a `std::coroutine_handle<T>` from `await_suspend()` as a way of indicating that execution should be symmetrically transferred to the coroutine identified by the returned handle.
- Add a `std::experimental::noop_coroutine()` function that returns a special `std::coroutine_handle` that can be returned from `await_suspend()` to suspend the current coroutine and return from the call to `.resume()` instead of transferring execution to another coroutine.

So what do we mean by "symmetric transfer"?

When you resume a coroutine by calling `.resume()` on it's `std::coroutine_handle` the caller of `.resume()` remains active on the stack while the resumed coroutine executes. When this coroutine next suspends and the call to `await_suspend()` for that suspend-point returns either `void` (indicating unconditional suspend) or `true` (indicating conditional suspend) then call to `.resume()` will return.

This can be thought of as an "asymmetric transfer" of execution to the coroutine and behaves just like an ordinary function call. The caller of `.resume()` can be any function (which may or may not be a coroutine). When that coroutine suspends and returns either `true` or `void` from `await_suspend()` then execution will return from the call to `.resume()` and

Every time we resume a coroutine by calling `.resume()` we create a new stack-frame for the execution of that coroutine.

However, with "symmetric transfer" we are simply suspending one coroutine and resuming another coroutine. There is no implicit caller/callee relationship between the two coroutines - when a coroutine suspends it can transfer execution to any suspended coroutine (including itself) and does not necessarily have to transfer execution back to the previous coroutine when it next suspends or completes.

Let's look at what the compiler lowers a `co_await` expression to when the awaiter makes use of symmetric-transfer:

```
{
  decltype(auto) value = <expr>;
  decltype(auto) awaitable =
      get_awaitable(promise, static_cast<decltype(value)&&>(value));
  decltype(auto) awaiter =
      get_awaiter(static_cast<decltype(awaitable)&&>(awaitable));
  if (!awaiter.await_ready())
  {
    using handle_t = std::coroutine_handle<P>;

    //<suspend-coroutine>

    auto h = awaiter.await_suspend(handle_t::from_promise(p));
    h.resume();
    //<return-to-caller-or-resumer>

    //<resume-point>
  }

  return awaiter.await_resume();
}
```

Let's zoom in on the key part that differs from other `co_await` forms:

```
auto h = awaiter.await_suspend(handle_t::from_promise(p));
h.resume();
//<return-to-caller-or-resumer>
```

Once the coroutine state-machine is lowered (a topic for another post), the `<return-to-caller-or-resumer>` part basically becomes a `return;` statement which causes the call to `.resume()` that last resumed the coroutine to return to its caller.

This means that we have the situation where we have a call to another function with the same signature, `std::coroutine_handle::resume()`, followed by a `return;` from the current function which is itself the body of a `std::coroutine_handle::resume()` call.

Some compilers, when optimisations are enabled, are able to apply an optimisation that turns calls to other functions the tail-position (ie. just before returning) into tail-calls as long as some conditions are met.

It just so happens that this kind of tail-call optimisation is exactly the kind of thing we want to be able to do to avoid the stack-overflow problem we were encountering before. But instead of being at the mercy of the optimiser as to whether or not the tail-call transformation is perfromed, we want to be able to guarantee that the tail-call transformation occurs, even when optimisations are not enabled.

But first let's dig into what we mean by tail-calls.

Tail-calls

A tail-call is one where the current stack-frame is popped before the call and the current function's return address becomes the return-address for the callee. ie. the callee will return directly the the caller of this function.

On X86/X64 architectures this generally means that the compiler will generate code that first pops the current stack-frame and then uses a `jmp` instruction to jump to the called function's entry-point instead of using a `call` instruction and then popping the current stack-frame after the `call` returns.

This optimisation is generally only possible to do in limited circumstances, however.

In particular, it requires that:

- the calling convention supports tail-calls and is the same for the caller and callee;
- the return-type is the same;
- there are no non-trivial destructors that need to be run after the call before returning to the caller; and
- the call is not inside a try/catch block.

The shape of the symmetric-transfer form of `co_await` has actually been designed specifically to allow coroutines to satisfy all of these requirements. Let's look at them individually.

Calling convention When the compiler lowers a coroutine into machine code it actually splits the coroutine up into two parts: the ramp (which allocates and initialises the coroutine frame) and the body (which contains the state-machine for the user-authored coroutine body).

The function signature of the coroutine (and thus any user-specified calling-convention) affects only the ramp part, whereas the body part is under the control of the compiler and is never directly called by any user-code - only by the ramp function and by `std::coroutine_handle::resume()`.

The calling-convention of the coroutine body part is not user-visible and is entirely up to the compiler and thus it can choose an appropriate calling convention that supports tail-calls and that is used by all coroutine bodies.

Return type is the same The return-type for both the source and target coroutine's `.resume()` method is `void` so this requirement is trivially satisfied.

No non-trivial destructors When performing a tail-call we need to be able to free the current stack-frame before calling the target function and this requires the lifetime of all stack-allocated objects to have ended prior to the call.

Normally, this would be problematic as soon as there are any objects with non-trivial destructors in-scope as the lifetime of those objects would not yet have ended and those objects would have been allocated on the stack.

However, when a coroutine suspends it does so without exiting any scopes and the way it achieves this is by placing any objects whose lifetime spans a suspend-point in the coroutine frame rather than allocating them on the stack.

Local variables with lifetimes that do not span a suspend-point may be allocated on the stack, but the lifetime of these objects will have already ended and their destructors will have been called before the coroutine next suspends.

Thus there should be no non-trivial destructors for stack-allocated objects that need to be run after the return of the tail-call.

Call not inside a try/catch block This one is a little tricker as within every coroutine there is an implicit try/catch block that encloses the user-authored body of the coroutine.

From the specification, we see that the coroutine is defined as:

```
{
  promise_type promise;
  co_await promise.initial_suspend();
  try { F; }
  catch (...) { promise.unhandled_exception(); }
final_suspend:
  co_await promise.final_suspend();
}
```

Where `F` is the user-authored part of the coroutine body.

Thus every user-authored `co_await` expression (other than initial/final_suspend) exists within the context of a try/catch block.

However, implementations work around this by actually executing the call to `.resume()` *outside* of the context of the try-block.

I hope to be able to go into this aspect in more detail in another blog post that goes into the details of the lowering of a coroutine to machine-code (this post is already long enough).

> *Note, however, that the current wording in the C++ specification is not clear on requiring implementations to do this and it is only a non-normative note that hints that this is something that might be required. Hopefully we'll be able to fix the specification in the future.*

So we see that coroutines performing a symmetric-transfer generally satisfy all of the requirements for being able to perform a tail-call. The compiler guarantees that this will always be a tail-call, regardless of whether optimisations are enabled or not.

This means that by using the `std::coroutine_handle`-returning flavour of `await_suspend()` we can suspend the current coroutine and transfer execution to another coroutine without consuming extra stack-space.

This allows us to write coroutines that mutually and recursively resume each other to an arbitrary depth without fear of overflowing the stack.

This is exactly what we need to fix our `task` implementation.

`task` revisited

So with the new "symmetric transfer" capability under our belt let's go back and fix our `task` type implementation.

To do this we need to make changes to the two `await_suspend()` methods in our implementation:

- First so that when we await the task that we perform a symmetric-transfer to resume the task's coroutine.
- Second so that when the task's coroutine completes that it performs a symmetric transfer to resume the awaiting coroutine.

To address the await direction we need to change the `task::awaiter` method from this:

```
void task::awaiter::await_suspend(
    std::coroutine_handle<> continuation) noexcept {
  // Store the continuation in the task's promise so that the final_suspend()
  // knows to resume this coroutine when the task completes.
  coro_.promise().continuation = continuation;

  // Then we resume the task's coroutine, which is currently suspended
  // at the initial-suspend-point (ie. at the open curly brace).
  coro_.resume();
}
```

to this:

```
std::coroutine_handle<> task::awaiter::await_suspend(
    std::coroutine_handle<> continuation) noexcept {
  // Store the continuation in the task's promise so that the final_suspend()
  // knows to resume this coroutine when the task completes.
  coro_.promise().continuation = continuation;

  // Then we tail-resume the task's coroutine, which is currently suspended
  // at the initial-suspend-point (ie. at the open curly brace), by returning
  // its handle from await_suspend().
  return coro_;
}
```

And to address the return-path we need to update the `task::promise_type::final_awaiter` method from this:

```
void task::promise_type::final_awaiter::await_suspend(
    std::coroutine_handle<promise_type> h) noexcept {
  // The coroutine is now suspended at the final-suspend point.
  // Lookup its continuation in the promise and resume it.
  h.promise().continuation.resume();
}
```

to this:

```
std::coroutine_handle<> task::promise_type::final_awaiter::await_suspend(
    std::coroutine_handle<promise_type> h) noexcept {
  // The coroutine is now suspended at the final-suspend point.
  // Lookup its continuation in the promise and resume it symmetrically.
  return h.promise().continuation;
}
```

And now we have a `task` implementation that doesn't suffer from the stack-overflow problem that the `void`-returning `await_suspend` flavour had and that doesn't have the non-deterministic resumption context problem of the `bool`-returning `await_suspend` flavour had.

Visualising the stack

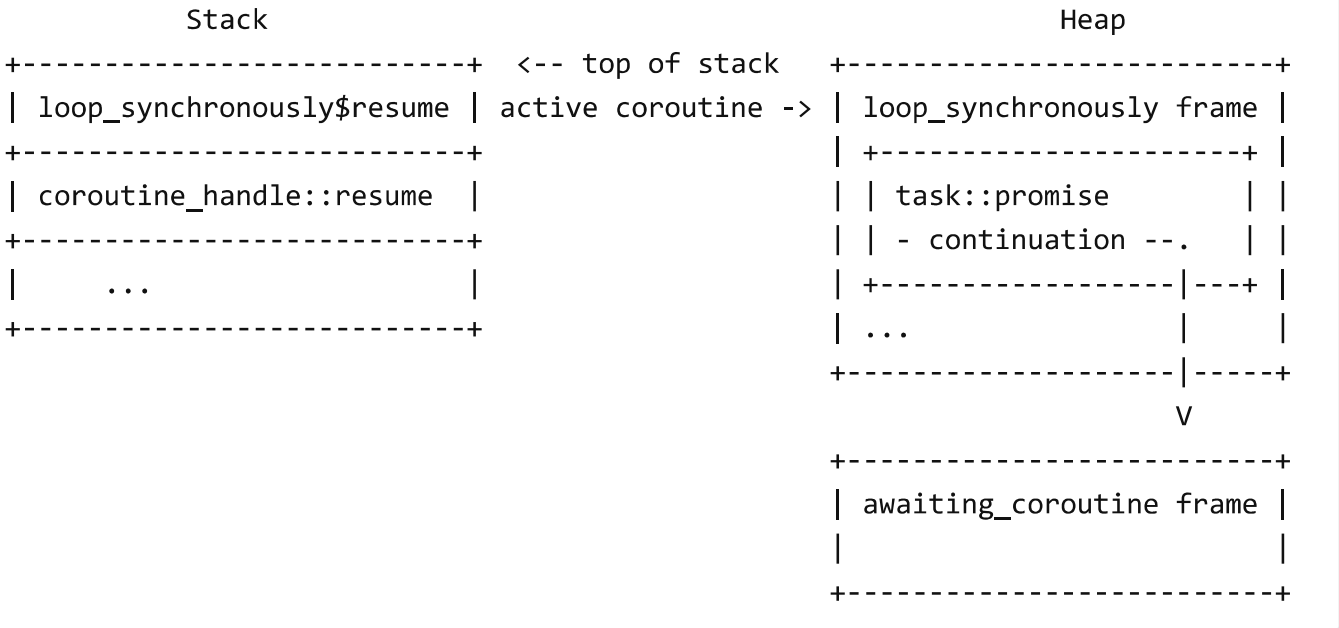
Let's now go back and have a look at our original example:

```
task completes_synchronously() {
  co_return;
}

task loop_synchronously(int count) {
  for (int i = 0; i < count; ++i) {
    co_await completes_synchronously();
  }
}
```

When the `loop_synchronously()` coroutine first starts executing it will be because some other coroutine `co_await`ed the `task` returned. This will have been launched by symmetric transfer from some other coroutine, which would have been resumed by a call to `std::coroutine_handle::resume()`.

Thus the stack will look something like this when `loop_synchronously()` starts:

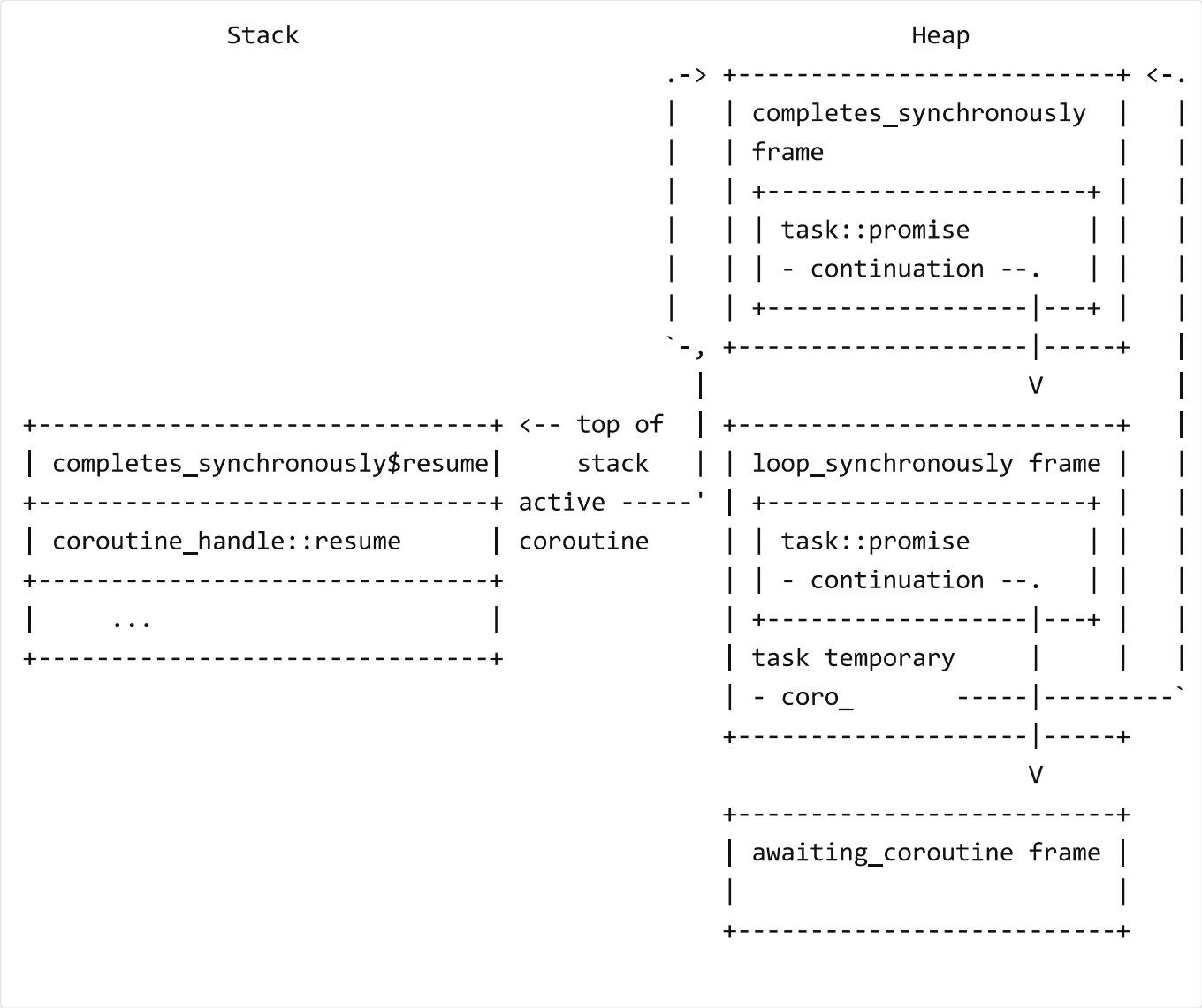


Now, when it executes `co_await completes_synchronously()` it will perform a symmetric transfer to `completes_synchronously` coroutine.

It does this by:

- calling the `task::operator co_await()` which then returns the `task::awaiter` object
- then suspends and calls `task::awaiter::await_suspend()` which then returns the `coroutine_handle` of the `completes_synchronously` coroutine.
- then performs a tail-call / jump to `completes_synchronously` coroutine. This pops the `loop_synchronously` frame before activing the `completes_synchronously` frame.

If we now look at the stack just after `completes_synchronously` is resumed it will now look like this:



Note that the number of stack-frames has not grown here.

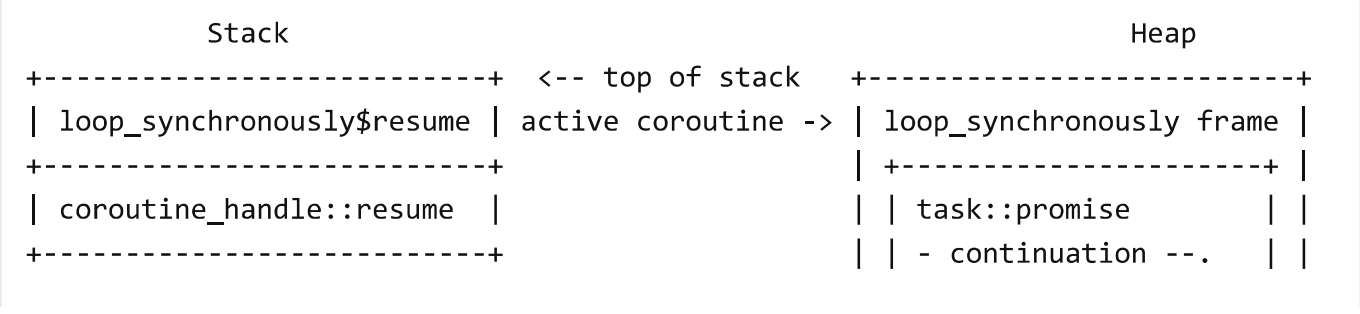
After the `completes_synchronously` coroutine completes and execution reaches the closing curly brace it will evaluate `co_await promise.final_suspend()`.

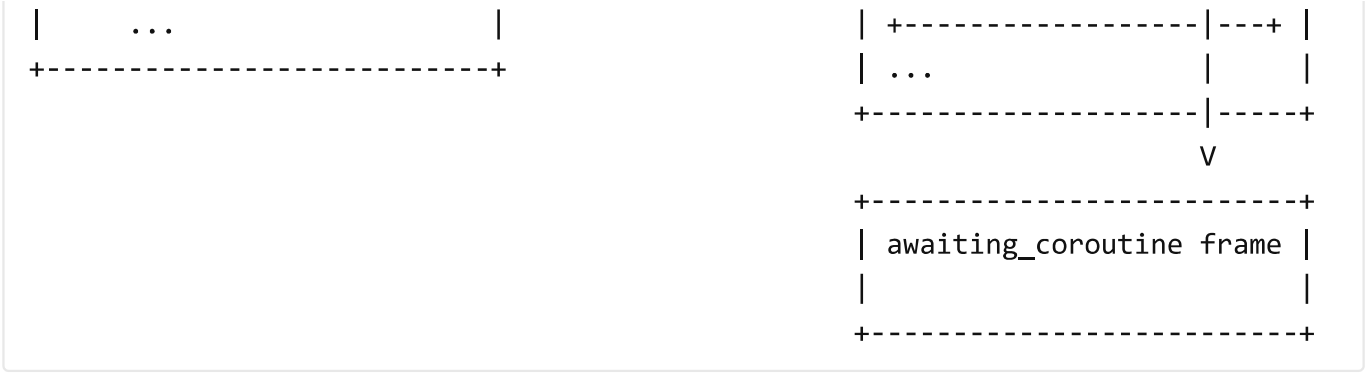
This will suspend the coroutine and call `final_awaiter::await_suspend()` which return the continuation's `std::coroutine_handle` (ie. the handle that points to the

`loop_synchronously` coroutine). This will then do a symmetric transfer/tail-call to resume the `loop_synchronously` coroutine.

If we look at the stack just after `loop_synchronously` is resumed then it will look something like this:

The first thing the `loop_synchronously` coroutine is going to do once resumed is to call the destructor of the temporary `task` that was returned from the call to `completes_synchronously` when execution reaches the semicolon. This will destroy the coroutine-frame, freeing its memory and leaving us with the following situtation:





We are now back to executing the `loop_synchronously` coroutine and we now have the same number of stack-frames and coroutine-frames as we started, and will do so each time we go around the loop.

Thus we can perform as many iterations of the loop as we want and will only use a constant amount of storage space.

For a full example of the symmetric-transfer version of the `task` type see the following Compiler Explorer link: https://godbolt.org/z/9baieF.

Symmetric Transfer as the Universal Form of await_suspend

Now that we see the power and importance of the symmetric-transfer form of the awaitable concept, I want to show you that this form is actually the universal form, which can theoretically replace the `void` and `bool`-returning forms of `await_suspend()`.

But first we need to look at the other piece that the P0913R0 proposal added to the coroutines design: `std::noop_coroutine()`.

Terminating the recursion

With the symmetric-transfer form of coroutines, every time the coroutine suspends it symmetrically resumes another coroutine. This is great as long as you have another coroutine to resume, but sometimes we don't have another coroutine to execute and just need to suspend and let execution return to the caller of `std::coroutine_handle::resume()`.

Both the `void`-returning and `bool`-returning flavours of `await_suspend()` allow a coroutine to suspend and return from `std::coroutine_handle::resume()`, so how do we do that with the symmetric-transfer flavour?

The answer is by using the special builtin `std::coroutine_handle`, called the "noop coroutine handle" which is produced by the function `std::noop_coroutine()`.

The "noop coroutine handle" is named as such because its `.resume()` implementation is such that it just immediately returns. i.e. resuming the coroutine is a no-op. Typically its implementation contains a single `ret` instruction.

If the `await_suspend()` method returns the `std::noop_coroutine()` handle then instead of transferring execution to the next coroutine, it transfers execution back to the caller of `std::coroutine_handle::resume()`.

Representing the other flavours of `await_suspend()`

With this information in-hand we can now show how to represent the other flavours of `await_suspend()` using the symmetric-transfer form.

The `void`-returning form

```
void my_awaiter::await_suspend(std::coroutine_handle<> h) {
  this->coro = h;
  enqueue(this);
}
```

can also be written using both the `bool`-returning form:

```
bool my_awaiter::await_suspend(std::coroutine_handle<> h) {
  this->coro = h;
  enqueue(this);
  return true;
}
```

and can be written using the symmetric-transfer form:

```
std::noop_coroutine_handle my_awaiter::await_suspend(
    std::coroutine_handle<> h) {
  this->coro = h;
  enqueue(this);
  return std::noop_coroutine();
}
```

The `bool`-returning form:

```
bool my_awaiter::await_suspend(std::coroutine_handle<> h) {
  this->coro = h;
  if (try_start(this)) {
    // Operation will complete asynchronously.
```

```
    // Return true to transfer execution to caller of
    // coroutine_handle::resume().
    return true;
  }

  // Operation completed synchronously.
  // Return false to immediately resume the current coroutine.
  return false;
}
```

can also be written using the symmetric-transfer form:

```
std::coroutine_handle<> my_awaiter::await_suspend(std::coroutine_handle<> h) {
  this->coro = h;
  if (try_start(this)) {
    // Operation will complete asynchronously.
    // Return std::noop_coroutine() to transfer execution to caller of
    // coroutine_handle::resume().
    return std::noop_coroutine();
  }

  // Operation completed synchronously.
  // Return current coroutine's handle to immediately resume
  // the current coroutine.
  return h;
}
```

Why have all three flavours?

So why do we still have the `void` and `bool`-returning flavours of `await_suspend()` when we have the symmetric-transfer flavour?

The reason is partly historical, partly pragmatic and partly performance.

The `void`-returning version could be entirely replaced by returning the `std::noop_coroutine_handle` type from `await_suspend()` as this would be an equivalent signal to the compiler that the coroutine is unconditionally transfering execution to the caller of `std::coroutine_handle::resume()`.

That it was kept was, IMO, partly because it was already in-use prior to the introduction of symmetric-transfer and partly because the `void`-form results in less-code/less-typing for the unconditional suspend case.

The `bool`-returning version, however, can have a slight win in terms of optimisability in some cases compared to the symmetric-transfer form.

Consider the case where we have a `bool`-returning `await_suspend()` method that is defined in another translation unit. In this case the compiler can generate code in the awaiting coroutine that will suspend the current coroutine and then conditionally resume it after the call to `await_suspend()` returns by just executing the next piece of code. It knows exactly the piece of code to execute next if `await_suspend()` returns `false`.

With the symmetric-transfer flavour we still need to represent the same outcomes; either return to the caller/resume or resume the current coroutine. Instead of returning `true` or `false` we need to return `std::noop_coroutine()` or the handle to the current coroutine. We can coerce both of these handles into a `std::coroutine_handle<void>` type and return it.

However, now, because the `await_suspend()` method is defined in another translation unit the compiler can't see what coroutine the returned handle is referring to and so when it resumes the coroutine it now has to perform some more expensive indirect calls and possibly some branches to resume the coroutine, compared to a single branch for the `bool`-returning case.

Now, it's possible that we might be able to get equivalent performance out of the symmetric transfer version one day. For example, we could write our code in such a way that `await_suspend()` is defined inline but calls a `bool`-returning method that is defined out-of-line and then conditionally returns the appropriate handle.

For example:

```
struct my_awaiter {
  bool await_ready();

  // Compilers should in-theory be able to optimise this to the same
  // as the bool-returning version, but currently don't do this optimisation.
  std::coroutine_handle<> await_suspend(std::coroutine_handle<> h) {
    if (try_start(h)) {
      return std::noop_coroutine();
    } else {
      return h;
    }
  }

  void await_resume();

private:
  // This method is defined out-of-line in a separate translation unit.
```

```
  bool try_start(std::coroutine_handle<> h);
}
```

However, current compilers (c. Clang 10) are not currently able to optimise this to as efficient code as the equivalent `bool`-returning version. Having said that, you're probably not going to notice the difference unless you're awaiting this in a really tight loop.

So, for now, the general rule is:

- If you need to unconditionally return to `.resume()` caller, use the `void`-returning flavour.
- If you need to conditionally return to `.resume()` caller or resume current coroutine use the `bool`-returning flavour.
- If you need to resume another coroutine use the symmetric-transfer flavour.

Rounding out

The new symmetric transfer capability added to coroutines for C++20 makes it much easier to write coroutines that recursively resume each other without fear of running into stack-overflow. This capability is key to making efficient and safe async coroutine types, such as the `task` one presented here.

This ended up being a much longer than expected post on symmetric transfer. If you made it this far, then thanks for sticking with it! I hope you found it useful.

In the next post, I'll dive into understanding how the compiler transforms a coroutine function into a state-machine.

Thanks

Thanks to Eric Niebler and Corentin Jabot for providing feedback on drafts of this post.

Comments

Comments are welcome in this GitHub issue

Asymmetric Transfer

Lewis Baker

lewissbaker

lewissbaker

Some thoughts on programming, C++ and other things.