Coroutine Theory

Sep 25, 2017

This is the first of a series of posts on the C++ Coroutines TS, a new language feature that is currently on track for inclusion into the C++20 language standard.

In this series I will cover how the underlying mechanics of C++ Coroutines work as well as show how they can be used to build useful higher-level abstractions such as those provided by the cppcoro library.

In this post I will describe the differences between functions and coroutines and provide a bit of theory about the operations they support. The aim of this post is introduce some foundational concepts that will help frame the way you think about C++ Coroutines.

Coroutines are Functions are Coroutines

A coroutine is a generalisation of a function that allows the function to be suspended and then later resumed.

I will explain what this means in a bit more detail, but before I do I want to first review how a "normal" C++ function works.

"Normal" Functions

A normal function can be thought of as having two operations: **Call** and **Return** (Note that I'm lumping "throwing an exception" here broadly under the **Return** operation).

The **Call** operation creates an activation frame, suspends execution of the calling function and transfers execution to the start of the function being called.

The **Return** operation passes the return-value to the caller, destroys the activation frame and then resumes execution of the caller just after the point at which it called the function.

Let's analyse these semantics a little more...

Activation Frames

So what is this 'activation frame' thing?

You can think of the activation frame as the block of memory that holds the current state of a particular invocation of a function. This state includes the values of any parameters that were passed to it and the values of any local variables.

For "normal" functions, the activation frame also includes the return-address - the address of the instruction to transfer execution to upon returning from the function - and the address of the activation frame for the invocation of the calling function. You can think of these pieces of information together as describing the 'continuation' of the function-call. ie. they describe which invocation of which function should continue executing at which point when this function completes.

With "normal" functions, all activation frames have strictly nested lifetimes. This strict nesting allows use of a highly efficient memory allocation data-structure for allocating and freeing the activation frames for each of the function calls. This data-structure is commonly referred to as "the stack".

When an activation frame is allocated on this stack data structure it is often called a "stack frame".

This stack data-structure is so common that most (all?) CPU architectures have a dedicated register for holding a pointer to the top of the stack (eg. in X64 it is the `rsp` register).

To allocate space for a new activation frame, you just increment this register by the frame-size. To free space for an activation frame, you just decrement this register by the frame-size.

The 'Call' Operation

When a function calls another function, the caller must first prepare itself for suspension.

This 'suspend' step typically involves saving to memory any values that are currently held in CPU registers so that those values can later be restored if required when the function resumes execution. Depending on the calling convention of the function, the caller and callee may coordinate on who saves these register values, but you can still think of them as being performed as part of the **Call** operation.

The caller also stores the values of any parameters passed to the called function into the new activation frame where they can be accessed by the function.

Finally, the caller writes the address of the resumption-point of the caller to the new activation frame and transfers execution to the start of the called function.

In the X86/X64 architecture this final operation has its own instruction, the `call` instruction, that writes the address of the next instruction onto the stack, increments the stack register by the size of the address and then jumps to the address specified in the instruction's operand.

The 'Return' Operation

When a function returns via a `return`-statement, the function first stores the return value (if any) where the caller can access it. This could either be in the caller's activation frame or the function's activation frame (the distinction can get a bit blurry for parameters and return values that cross the boundary between two activation frames).

Then the function destroys the activation frame by:

- Destroying any local variables in-scope at the return-point.
- Destroying any parameter objects
- Freeing memory used by the activation-frame

And finally, it resumes execution of the caller by:

- Restoring the activation frame of the caller by setting the stack register to point to the activation frame of the caller and restoring any registers that might have been clobbered by the function.
- Jumping to the resume-point of the caller that was stored during the 'Call' operation.

Note that as with the 'Call' operation, some calling conventions may split the repsonsibilities of the 'Return' operation across both the caller and callee function's instructions.

Coroutines

Coroutines generalise the operations of a function by separating out some of the steps performed in the **Call** and **Return** operations into three extra operations: **Suspend**, **Resume** and **Destroy**.

The **Suspend** operation suspends execution of the coroutine at the current point within the function and transfers execution back to the caller or resumer without destroying the activation frame. Any objects in-scope at the point of suspension remain alive after the coroutine execution is suspended.

Note that, like the **Return** operation of a function, a coroutine can only be suspended from within the coroutine itself at well-defined suspend-points.

The **Resume** operation resumes execution of a suspended coroutine at the point at which it was suspended. This reactivates the coroutine's activation frame.

The **Destroy** operation destroys the activation frame without resuming execution of the coroutine. Any objects that were in-scope at the suspend point will be destroyed. Memory used to store the activation frame is freed.

Coroutine activation frames

Since coroutines can be suspended without destroying the activation frame, we can no longer guarantee that activation frame lifetimes will be strictly nested. This means that

activation frames cannot in general be allocated using a stack data-structure and so may need to be stored on the heap instead.

There are some provisions in the C++ Coroutines TS to allow the memory for the coroutine frame to be allocated from the activation frame of the caller if the compiler can prove that the lifetime of the coroutine is indeed strictly nested within the lifetime of the caller. This can avoid heap allocations in many cases provided you have a sufficiently smart compiler.

With coroutines there are some parts of the activation frame that need to be preserved across coroutine suspension and there are some parts that only need to be kept around while the coroutine is executing. For example, the lifetime of a variable with a scope that does not span any coroutine suspend-points can potentially be stored on the stack.

You can logically think of the activation frame of a coroutine as being comprised of two parts: the 'coroutine frame' and the 'stack frame'.

The 'coroutine frame' holds part of the coroutine's activation frame that persists while the coroutine is suspended and the 'stack frame' part only exists while the coroutine is executing and is freed when the coroutine suspends and transfers execution back to the caller/resumer.

The 'Suspend' operation

The **Suspend** operation of a coroutine allows the coroutine to suspend execution in the middle of the function and transfer execution back to the caller or resumer of the coroutine.

There are certain points within the body of a coroutine that are designated as suspend-points. In the C++ Coroutines TS, these suspend-points are identified by usages of the `co_await` or `co_yield` keywords.

When a coroutine hits one of these suspend-points it first prepares the coroutine for resumption by:

- Ensuring any values held in registers are written to the coroutine frame
- Writing a value to the coroutine frame that indicates which suspend-point the coroutine is being suspended at. This allows a subsequent **Resume** operation to know where to resume execution of the coroutine or so a subsequent **Destroy** to know what values were in-scope and need to be destroyed.

Once the coroutine has been prepared for resumption, the coroutine is considered 'suspended'.

The coroutine then has the opportunity to execute some additional logic before execution is transferred back to the caller/resumer. This additional logic is given access to a handle to the coroutine-frame that can be used to later resume or destroy it.

This ability to execute logic after the coroutine enters the 'suspended' state allows the coroutine to be scheduled for resumption without the need for synchronisation that would

otherwise be required if the coroutine was scheduled for resumption prior to entering the 'suspended' state due to the potential for suspension and resumption of the coroutine to race. I'll go into this in more detail in future posts.

The coroutine can then choose to either immediately resume/continue execution of the coroutine or can choose to transfer execution back to the caller/resumer.

If execution is transferred to the caller/resumer the stack-frame part of the coroutine's activation frame is freed and popped off the stack.

The 'Resume' operation

The **Resume** operation can be performed on a coroutine that is currently in the 'suspended' state.

When a function wants to resume a coroutine it needs to effectively 'call' into the middle of a particular invocation of the function. The way the resumer identifies the particular invocation to resume is by calling the `void resume()` method on the coroutine-frame handle provided to the corresponding **Suspend** operation.

Just like a normal function call, this call to `resume()` will allocate a new stack-frame and store the return-address of the caller in the stack-frame before transferring execution to the function.

However, instead of transferring execution to the start of the function it will transfer execution to the point in the function at which it was last suspended. It does this by loading the resume-point from the coroutine-frame and jumping to that point.

When the coroutine next suspends or runs to completion this call to `resume()` will return and resume execution of the calling function.

The 'Destroy' operation

The **Destroy** operation destroys the coroutine frame without resuming execution of the coroutine.

This operation can only be performed on a suspended coroutine.

The **Destroy** operation acts much like the **Resume** operation in that it re-activates the coroutine's activation frame, including allocating a new stack-frame and storing the return-address of the caller of the **Destroy** operation.

However, instead of transferring execution to the coroutine body at the last suspend-point it instead transfers execution to an alternative code-path that calls the destructors of all local variables in-scope at the suspend-point before then freeing the memory used by the coroutine frame.

Similar to the **Resume** operation, the **Destroy** operation identifies the particular activation-frame to destroy by calling the `void destroy()` method on the coroutine-frame handle provided during the corresponding **Suspend** operation.

The 'Call' operation of a coroutine

The **Call** operation of a coroutine is much the same as the call operation of a normal function. In fact, from the perspective of the caller there is no difference.

However, rather than execution only returning to the caller when the function has run to completion, with a coroutine the call operation will instead resume execution of the caller when the coroutine reaches its first suspend-point.

When performing the **Call** operation on a coroutine, the caller allocates a new stack-frame, writes the parameters to the stack-frame, writes the return-address to the stack-frame and transfers execution to the coroutine. This is exactly the same as calling a normal function.

The first thing the coroutine does is then allocate a coroutine-frame on the heap and copy/move the parameters from the stack-frame into the coroutine-frame so that the lifetime of the parameters extends beyond the first suspend-point.

The 'Return' operation of a coroutine

The **Return** operation of a coroutine is a little different from that of a normal function.

When a coroutine executes a `return`-statement (`co_return` according to the TS) operation it stores the return-value somewhere (exactly where this is stored can be customised by the coroutine) and then destructs any in-scope local variables (but not parameters).

The coroutine then has the opportunity to execute some additional logic before transferring execution back to the caller/resumer.

This additional logic might perform some operation to publish the return value, or it might resume another coroutine that was waiting for the result. It's completely customisable.

The coroutine then performs either a **Suspend** operation (keeping the coroutine-frame alive) or a **Destroy** operation (destroying the coroutine-frame).

Execution is then transferred back to the caller/resumer as per the **Suspend**/**Destroy** operation semantics, popping the stack-frame component of the activation-frame off the stack.

It is important to note that the return-value passed to the **Return** operation is not the same as the return-value returned from a **Call** operation as the return operation may be executed long after the caller resumed from the initial **Call** operation.

An illustration

To help put these concepts into pictures, I want to walk through a simple example of what happens when a coroutine is called, suspends and is later resumed.

So let's say we have a function (or coroutine), `f()` that calls a coroutine, `x(int a)`.

Before the call we have a situation that looks a bit like this:

```
STACK                     REGISTERS              HEAP


                          +------+
+---------------+ <------ | rsp  |
|  f()          |         +------+
+---------------+
| ...           |
|               |
```

Then when `x(42)` is called, it first creates a stack frame for `x()`, as with normal functions.

```
STACK                     REGISTERS              HEAP
+----------------+ <-+
|  x()           |   |
| a  = 42        |   |
| ret= f()+0x123 |   |    +------+
+----------------+   +--- | rsp  |
|  f()           |        +------+
+----------------+
| ...            |
|                |
```

Then, once the coroutine `x()` has allocated memory for the coroutine frame on the heap and copied/moved parameter values into the coroutine frame we'll end up with something that looks like the next diagram. Note that the compiler will typically hold the address of the coroutine frame in a separate register to the stack pointer (eg. MSVC stores this in the `rbp` register).

```
STACK                     REGISTERS              HEAP
+----------------+ <-+
|  x()           |   |
| a  = 42        |   |                   +-->  +-----------+
| ret= f()+0x123 |   |    +------+       |     |  x()      |
+----------------+   +--- | rsp  |       |     | a =  42   |
|  f()           |        +------+       |     +-----------+
+----------------+        | rbp  | ------+
```

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

If the coroutine `x()` then calls another normal function `g()` it will look something like this.

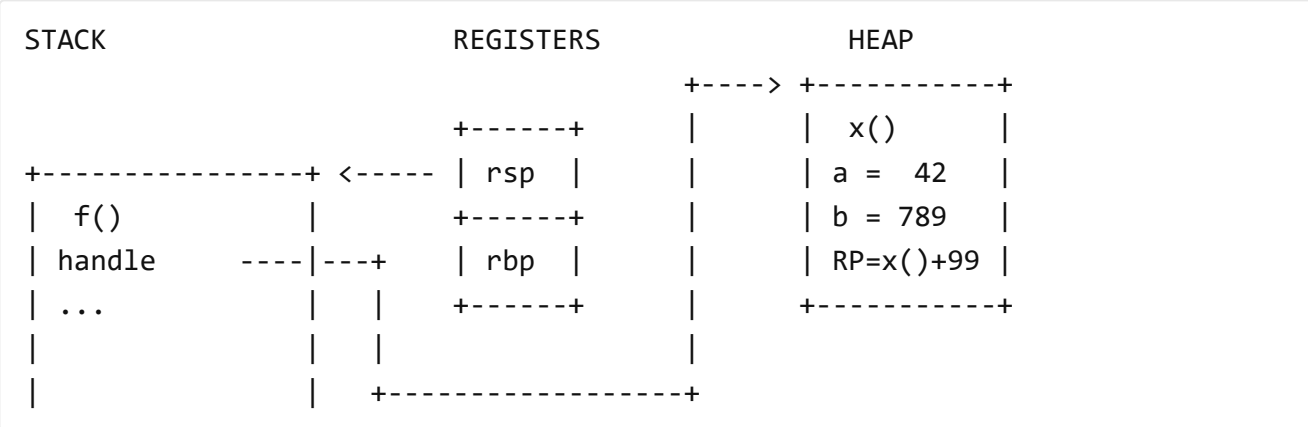
```
STACK                    REGISTERS              HEAP
+---------------+ <-+
|  g()          |   |
| ret= x()+0x45 |   |
+---------------+   |
|  x()          |   |
| coroframe     | --|--------------------+
| a  = 42       |   |                    +-->  +-----------+
| ret= f()+0x123|   |    +------+              |  x()      |
+---------------+   +--- | rsp  |              | a =  42   |
|  f()          |        +------+              +-----------+
+---------------+        | rbp  |
| ...           |        +------+
|               |
```

When `g()` returns it will destroy its activation frame and restore `x()`'s activation frame. Let's say we save `g()`'s return value in a local variable `b` which is stored in the coroutine frame.

```
STACK                    REGISTERS              HEAP
+---------------+ <-+
|  x()          |   |
| a  = 42       |   |                    +-->  +-----------+
| ret= f()+0x123|   |    +------+        |     |  x()      |
+---------------+   +--- | rsp  |        |     | a =  42   |
|  f()          |        +------+        |     | b = 789   |
+---------------+        | rbp  | ------ +     +-----------+
| ...           |        +------+
|               |
```

If `x()` now hits a suspend-point and suspends execution without destroying its activation frame then execution returns to `f()`.

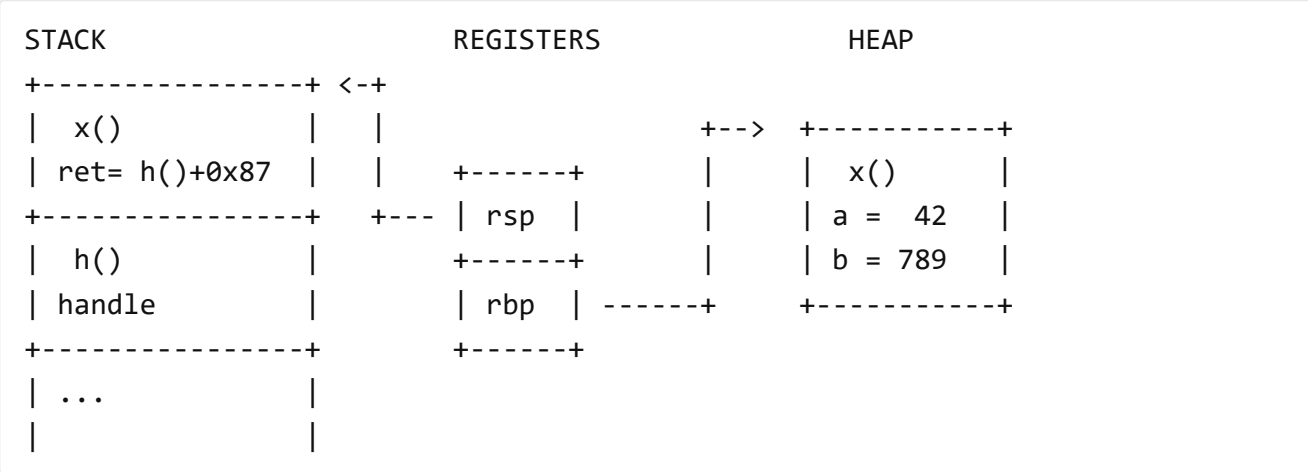
This results in the stack-frame part of `x()` being popped off the stack while leaving the coroutine-frame on the heap. When the coroutine suspends for the first time, a return-value is returned to the caller. This return value often holds a handle to the coroutine-frame that suspended that can be used to later resume it. When `x()` suspends it also stores the address of the resumption-point of `x()` in the coroutine frame (call it `RP` for resume-point).



This handle may now be passed around as a normal value between functions. At some point later, potentially from a different call-stack or even on a different thread, something (say, `h()`) will decide to resume execution of that coroutine. For example, when an async I/O operation completes.

The function that resumes the coroutine calls a `void resume(handle)` function to resume execution of the coroutine. To the caller, this looks just like any other normal call to a `void`-returning function with a single argument.

This creates a new stack-frame that records the return-address of the caller to `resume()`, activates the coroutine-frame by loading its address into a register and resumes execution of `x()` at the resume-point stored in the coroutine-frame.



In summary

I have described coroutines as being a generalisation of a function that has three additional operations - ‘Suspend’, ‘Resume’ and ‘Destroy’ - in addition to the ‘Call’ and ‘Return’ operations provided by “normal” functions.

I hope that this provides some useful mental framing for how to think of coroutines and their control-flow.

In the next post I will go through the mechanics of the C++ Coroutines TS language extensions and explain how the compiler translates code that you write into coroutines.

Comments

Comments are welcome in this GitHub issue

Asymmetric Transfer

Lewis Baker

lewissbaker

lewissbaker

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