Asymmetric Transfer

C++ Coroutines: Understanding operator co_await

Nov 17, 2017

In the previous post on Coroutine Theory I described the high-level differences between functions and coroutines but without going into any detail on syntax and semantics of coroutines as described by the C++ Coroutines TS (N4680).

The key new facility that the Coroutines TS adds to the C++ language is the ability to suspend a coroutine, allowing it to be later resumed. The mechanism the TS provides for doing this is via the new `co_await` operator.

Understanding how the `co_await` operator works can help to demystify the behaviour of coroutines and how they are suspended and resumed. In this post I will be explaining the mechanics of the `co_await` operator and introduce the related **Awaitable** and **Awaiter** type concepts.

But before I dive into `co_await` I want to give a brief overview of the Coroutines TS to provide some context.

What does the Coroutines TS give us?

- Three new language keywords: `co_await`, `co_yield` and `co_return`
- Several new types in the `std::experimental` namespace:
  - `coroutine_handle<P>`
  - `coroutine_traits<Ts...>`
  - `suspend_always`
  - `suspend_never`
- A general mechanism that library writers can use to interact with coroutines and customise their behaviour.
- A language facility that makes writing asynchronous code a whole lot easier!

The facilities the C++ Coroutines TS provides in the language can be thought of as a *low-level assembly-language* for coroutines. These facilities can be difficult to use directly in a safe way and are mainly intended to be used by library-writers to build higher-level abstractions that application developers can work with safely.

The plan is to deliver these new low-level facilities into an upcoming language standard (hopefully C++20) along with some accompanying higher-level types in the standard library

that wrap these low-level building-blocks and make coroutines more accessible in a safe way for application developers.

Compiler <-> Library interaction

Interestingly, the Coroutines TS does not actually define the semantics of a coroutine. It does not define how to produce the value returned to the caller. It does not define what to do with the return value passed to the `co_return` statement or how to handle an exception that propagates out of the coroutine. It does not define what thread the coroutine should be resumed on.

Instead, it specifies a general mechanism for library code to customise the behaviour of the coroutine by implementing types that conform to a specific interface. The compiler then generates code that calls methods on instances of types provided by the library. This approach is similar to the way that a library-writer can customise the behaviour of a range-based for-loop by defining the `begin()` / `end()` methods and an `iterator` type.

The fact that the Coroutines TS doesn't prescribe any particular semantics to the mechanics of a coroutine makes it a powerful tool. It allows library writers to define many different kinds of coroutines, for all sorts of different purposes.

For example, you can define a coroutine that produces a single value asynchronously, or a coroutine that produces a sequence of values lazily, or a coroutine that simplifies control-flow for consuming `optional<T>` values by early-exiting if a `nullopt` value is encountered.

There are two kinds of interfaces that are defined by the coroutines TS: The **Promise** interface and the **Awaitable** interface.

The **Promise** interface specifies methods for customising the behaviour of the coroutine itself. The library-writer is able to customise what happens when the coroutine is called, what happens when the coroutine returns (either by normal means or via an unhandled exception) and customise the behaviour of any `co_await` or `co_yield` expression within the coroutine.

The **Awaitable** interface specifies methods that control the semantics of a `co_await` expression. When a value is `co_await`ed, the code is translated into a series of calls to methods on the awaitable object that allow it to specify: whether to suspend the current coroutine, execute some logic after it has suspended to schedule the coroutine for later resumption, and execute some logic after the coroutine resumes to produce the result of the `co_await` expression.

I'll be covering details of the **Promise** interface in a future post, but for now let's look at the **Awaitable** interface.

Awaiters and Awaitables: Explaining `operator co_await`

The `co_await` operator is a new unary operator that can be applied to a value. For example: `co_await someValue`.

The `co_await` operator can only be used within the context of a coroutine. This is somewhat of a tautology though, since any function body containing use of the `co_await` operator, by definition, will be compiled as a coroutine.

A type that supports the `co_await` operator is called an **Awaitable** type.

Note that whether or not the `co_await` operator can be applied to a type can depend on the context in which the `co_await` expression appears. The promise type used for a coroutine can alter the meaning of a `co_await` expression within the coroutine via its `await_transform` method (more on this later).

To be more specific where required I like to use the term **Normally Awaitable** to describe a type that supports the `co_await` operator in a coroutine context whose promise type does not have an `await_transform` member. And I like to use the term **Contextually Awaitable** to describe a type that only supports the `co_await` operator in the context of certain types of coroutines due to the presence of an `await_transform` method in the coroutine's promise type. (I'm open to better suggestions for these names here...)

An **Awaiter** type is a type that implements the three special methods that are called as part of a `co_await` expression: `await_ready`, `await_suspend` and `await_resume`.

Note that I have shamelessly "borrowed" the term 'Awaiter' here from the C# `async` keyword's mechanics that is implemented in terms of a `GetAwaiter()` method which returns an object with an interface that is eerily similar to the C++ concept of an **Awaiter**. See this post for more details on C# awaiters.

Note that a type can be both an **Awaitable** type and an **Awaiter** type.

When the compiler sees a `co_await <expr>` expression there are actually a number of possible things it could be translated to depending on the types involved.

Obtaining the Awaiter

The first thing the compiler does is generate code to obtain the **Awaiter** object for the awaited value. There are a number of steps to obtaining the awaiter object which are set out in N4680 section 5.3.8(3).

Let's assume that the promise object for the awaiting coroutine has type, `P`, and that `promise` is an l-value reference to the promise object for the current coroutine.

If the promise type, `P`, has a member named `await_transform` then `<expr>` is first passed into a call to `promise.await_transform(<expr>)` to obtain the **Awaitable** value, `awaitable`. Otherwise, if the promise type does not have an `await_transform` member then we use the result of evaluating `<expr>` directly as the **Awaitable** object, `awaitable`.

Then, if the **Awaitable** object, `awaitable`, has an applicable `operator co_await()` overload then this is called to obtain the **Awaiter** object. Otherwise the object, `awaitable`, is used as the awaiter object.

If we were to encode these rules into the functions `get_awaitable()` and `get_awaiter()`, they might look something like this:

```
template<typename P, typename T>
decltype(auto) get_awaitable(P& promise, T&& expr)
{
  if constexpr (has_any_await_transform_member_v<P>)
    return promise.await_transform(static_cast<T&&>(expr));
  else
    return static_cast<T&&>(expr);
}

template<typename Awaitable>
decltype(auto) get_awaiter(Awaitable&& awaitable)
{
  if constexpr (has_member_operator_co_await_v<Awaitable>)
    return static_cast<Awaitable&&>(awaitable).operator co_await();
  else if constexpr (has_non_member_operator_co_await_v<Awaitable&&>)
    return operator co_await(static_cast<Awaitable&&>(awaitable));
  else
    return static_cast<Awaitable&&>(awaitable);
}
```

Awaiting the Awaiter

So, assuming we have encapsulated the logic for turning the `<expr>` result into an **Awaiter** object into the above functions then the semantics of `co_await <expr>` can be translated (roughly) as follows:

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

```
    using await_suspend_result_t =
      decltype(awaiter.await_suspend(handle_t::from_promise(p)));

    <suspend-coroutine>

    if constexpr (std::is_void_v<await_suspend_result_t>)
    {
      awaiter.await_suspend(handle_t::from_promise(p));
      <return-to-caller-or-resumer>
    }
    else
    {
      static_assert(
         std::is_same_v<await_suspend_result_t, bool>,
         "await_suspend() must return 'void' or 'bool'.");

      if (awaiter.await_suspend(handle_t::from_promise(p)))
      {
        <return-to-caller-or-resumer>
      }
    }

    <resume-point>
  }

  return awaiter.await_resume();
}
```

The `void`-returning version of `await_suspend()` unconditionally transfers execution back to the caller/resumer of the coroutine when the call to `await_suspend()` returns, whereas the `bool`-returning version allows the awaiter object to conditionally resume the coroutine immediately without returning to the caller/resumer.

The `bool`-returning version of `await_suspend()` can be useful in cases where the awaiter might start an async operation that can sometimes complete synchronously. In the cases where it completes synchronously, the `await_suspend()` method can return `false` to indicate that the coroutine should be immediately resumed and continue execution.

At the `<suspend-coroutine>` point the compiler generates some code to save the current state of the coroutine and prepare it for resumption. This includes storing the location of the `<resume-point>` as well as spilling any values currently held in registers into the coroutine frame memory.

The current coroutine is considered suspended after the `<suspend-coroutine>` operation completes. The first point at which you can observe the suspended coroutine is inside the

call to `await_suspend()`. Once the coroutine is suspended it is then able to be resumed or destroyed.

It is the responsibility of the `await_suspend()` method to schedule the coroutine for resumption (or destruction) at some point in the future once the operation has completed. Note that returning `false` from `await_suspend()` counts as scheduling the coroutine for immediate resumption on the current thread.

The purpose of the `await_ready()` method is to allow you to avoid the cost of the `<suspend-coroutine>` operation in cases where it is known that the operation will complete synchronously without needing to suspend.

At the `<return-to-caller-or-resumer>` point execution is transferred back to the caller or resumer, popping the local stack frame but keeping the coroutine frame alive.

When (or if) the suspended coroutine is eventually resumed then the execution resumes at the `<resume-point>`. ie. immediately before the `await_resume()` method is called to obtain the result of the operation.

The return-value of the `await_resume()` method call becomes the result of the `co_await` expression. The `await_resume()` method can also throw an exception in which case the exception propagates out of the `co_await` expression.

Note that if an exception propagates out of the `await_suspend()` call then the coroutine is automatically resumed and the exception propagates out of the `co_await` expression without calling `await_resume()`.

Coroutine Handles

You may have noticed the use of the `coroutine_handle<P>` type that is passed to the `await_suspend()` call of a `co_await` expression.

This type represents a non-owning handle to the coroutine frame and can be used to resume execution of the coroutine or to destroy the coroutine frame. It can also be used to get access to the coroutine's promise object.

The `coroutine_handle` type has the following (abbreviated) interface:

```
namespace std::experimental
{
  template<typename Promise>
  struct coroutine_handle;

  template<>
  struct coroutine_handle<void>
  {
    bool done() const;
```

```
    void resume();
    void destroy();

    void* address() const;
    static coroutine_handle from_address(void* address);
  };

  template<typename Promise>
  struct coroutine_handle : coroutine_handle<void>
  {
    Promise& promise() const;
    static coroutine_handle from_promise(Promise& promise);

    static coroutine_handle from_address(void* address);
  };
}
```

When implementing **Awaitable** types, they key method you'll be using on `coroutine_handle` will be `.resume()`, which should be called when the operation has completed and you want to resume execution of the awaiting coroutine. Calling `.resume()` on a `coroutine_handle` reactivates a suspended coroutine at the `<resume-point>`. The call to `.resume()` will return when the coroutine next hits a `<return-to-caller-or-resumer>` point.

The `.destroy()` method destroys the coroutine frame, calling the destructors of any in-scope variables and freeing memory used by the coroutine frame. You should generally not need to (and indeed should really avoid) calling `.destroy()` unless you are a library writer implementing the coroutine promise type. Normally, coroutine frames will be owned by some kind of RAII type returned from the call to the coroutine. So calling `.destroy()` without cooperation with the RAII object could lead to a double-destruction bug.

The `.promise()` method returns a reference to the coroutine's promise object. However, like `.destroy()`, it is generally only useful if you are authoring coroutine promise types. You should consider the coroutine's promise object as an internal implementation detail of the coroutine. For most **Normally Awaitable** types you should use `coroutine_handle<void>` as the parameter type to the `await_suspend()` method instead of `coroutine_handle<Promise>`.

The `coroutine_handle<P>::from_promise(P& promise)` function allows reconstructing the coroutine handle from a reference to the coroutine's promise object. Note that you must ensure that the type, `P`, exactly matches the concrete promise type used for the coroutine frame; attempting to construct a `coroutine_handle<Base>` when the concrete promise type is `Derived` can lead to undefined behaviour.

The `.address()` / `from_address()` functions allow converting a coroutine handle to/from a `void*` pointer. This is primarily intended to allow passing as a 'context' parameter into existing C-style APIs, so you might find it useful in implementing **Awaitable** types in some circumstances. However, in most cases I've found it necessary to pass additional information through to callbacks in this 'context' parameter so I generally end up storing the `coroutine_handle` in a struct and passing a pointer to the struct in the 'context' parameter rather than using the `.address()` return-value.

Synchronisation-free async code

One of the powerful design-features of the `co_await` operator is the ability to execute code after the coroutine has been suspended but before execution is returned to the caller/resumer.

This allows an Awaiter object to initiate an async operation after the coroutine is already suspended, passing the `coroutine_handle` of the suspended coroutine to the operation which it can safely resume when the operation completes (potentially on another thread) without any additional synchronisation required.

For example, by starting an async-read operation inside `await_suspend()` when the coroutine is already suspended means that we can just resume the coroutine when the operation completes without needing any thread-synchronisation to coordinate the thread that started the operation and the thread that completed the operation.

```
Time     Thread 1                              Thread 2
  |      --------                              --------
  |      ....                                  Call OS - Wait for I/O event
  |      Call await_ready()                       |
  |      <supend-point>                           |
  |      Call await_suspend(handle)               |
  |        Store handle in operation              |
  V        Start AsyncFileRead ---+               V
                                  +----->    <AsyncFileRead Completion Event>
                                             Load coroutine_handle from operati
                                             Call handle.resume()
                                               <resume-point>
                                               Call to await_resume()
                                               execution continues....
           Call to AsyncFileRead returns
         Call to await_suspend() returns
         <return-to-caller/resumer>
```

One thing to be *very* careful of when taking advantage of this approach is that as soon as you have started the operation which publishes the coroutine handle to other threads then

another thread may resume the coroutine on another thread before `await_suspend()` returns and may continue executing concurrently with the rest of the `await_suspend()` method.

The first thing the coroutine will do when it resumes is call `await_resume()` to get the result and then often it will immediately destruct the **Awaiter** object (ie. the `this` pointer of the `await_suspend()` call). The coroutine could then potentially run to completion, destructing the coroutine and promise object, all before `await_suspend()` returns.

So within the `await_suspend()` method, once it's possible for the coroutine to be resumed concurrently on another thread, you need to make sure that you avoid accessing `this` or the coroutine's `.promise()` object because both could already be destroyed. In general, the only things that are safe to access after the operation is started and the coroutine is scheduled for resumption are local variables within `await_suspend()`.

Comparison to Stackful Coroutines

I want to take a quick detour to compare this ability of the Coroutines TS stackless coroutines to execute logic after the coroutine is suspended with some existing common stackful coroutine facilities such as Win32 fibers or boost::context.

With many of the stackful coroutine frameworks, the suspend operation of a coroutine is combined with the resumption of another coroutine into a 'context-switch' operation. With this 'context-switch' operation there is typically no opportunity to execute logic after suspending the current coroutine but before transferring execution to another coroutine.

This means that if we want to implement a similar async-file-read operation on top of stackful coroutines then we have to start the operation *before* suspending the coroutine. It is therefore possible that the operation could complete on another thread before the coroutine is suspended and is eligible for resumption. This potential race between the operation completing on another thread and the coroutine suspending requires some kind of thread synchronisation to arbitrate and decide on the winner.

There are probably ways around this by using a trampoline context that can start the operation on behalf of the initiating context after the initiating context has been suspended. However this would require extra infrastructure and an extra context-switch to make it work and it's possible that the overhead this introduces would be greater than the cost of the synchronisation it's trying to avoid.

Avoiding memory allocations

Async operations often need to store some per-operation state that keeps track of the progress of the operation. This state typically needs to last for the duration of the operation and should only be freed once the operation has completed.

For example, calling async Win32 I/O functions requires you to allocate and pass a pointer to an `OVERLAPPED` structure. The caller is responsible for ensuring this pointer remains valid until the operation completes.

With traditional callback-based APIs this state would typically need to be allocated on the heap to ensure it has the appropriate lifetime. If you were performing many operations, you may need to allocate and free this state for each operation. If performance is an issue then a custom allocator may be used that allocates these state objects from a pool.

However, when we are using coroutines we can avoid the need to heap-allocate storage for the operation state by taking advantage of the fact that local variables within the coroutine frame will be kept alive while the coroutine is suspended.

By placing the per-operation state in the **Awaiter** object we can effectively "borrow" memory from the coroutine frame for storing the per-operation state for the duration of the `co_await` expression. Once the operation completes, the coroutine is resumed and the **Awaiter** object is destroyed, freeing that memory in the coroutine frame for use by other local variables.

Ultimately, the coroutine frame may still be allocated on the heap. However, once allocated, a coroutine frame can be used to execute many asynchronous operations with only that single heap allocation.

If you think about it, the coroutine frame acts as a kind of really high-performance arena memory allocator. The compiler figures out at compile time the total arena size it needs for all local variables and is then able to allocate this memory out to local variables as required with zero overhead! Try beating that with a custom allocator ;)

An example: Implementing a simple thread-synchronisation primitive

Now that we've covered a lot of the mechanics of the `co_await` operator, I want to show how to put some of this knowledge into practice by implementing a basic awaitable synchronisation primitive: An asynchronous manual-reset event.

The basic requirements of this event is that it needs to be **Awaitable** by multiple concurrently executing coroutines and when awaited needs to suspend the awaiting coroutine until some thread calls the `.set()` method, at which point any awaiting coroutines are resumed. If some thread has already called `.set()` then the coroutine should continue without suspending.

Ideally we'd also like to make it `noexcept`, require no heap allocations and have a lock-free implementation.

Edit 2017/11/23: Added example usage for `async_manual_reset_event`

Example usage should look something like this:

```
T value;
async_manual_reset_event event;

// A single call to produce a value
void producer()
{
  value = some_long_running_computation();

  // Publish the value by setting the event.
  event.set();
}

// Supports multiple concurrent consumers
task<> consumer()
{
  // Wait until the event is signalled by call to event.set()
  // in the producer() function.
  co_await event;

  // Now it's safe to consume 'value'
  // This is guaranteed to 'happen after' assignment to 'value'
  std::cout << value << std::endl;
}
```

Let's first think about the possible states this event can be in: 'not set' and 'set'.

When it's in the 'not set' state there is a (possibly empty) list of waiting coroutines that are waiting for it to become 'set'.

When it's in the 'set' state there won't be any waiting coroutines as coroutines that `co_await` the event in this state can continue without suspending.

This state can actually be represented in a single `std::atomic<void*>`.

- Reserve a special pointer value for the 'set' state. In this case we'll use the `this` pointer of the event since we know that can't be the same address as any of the list items.
- Otherwise the event is in the 'not set' state and the value is a pointer to the head of a singly linked-list of awaiting coroutine structures.

We can avoid extra calls to allocate nodes for the linked-list on the heap by storing the nodes within an 'awaiter' object that is placed within the coroutine frame.

So let's start with a class interface that looks something like this:

```
class async_manual_reset_event
{
public:

  async_manual_reset_event(bool initiallySet = false) noexcept;

  // No copying/moving
  async_manual_reset_event(const async_manual_reset_event&) = delete;
  async_manual_reset_event(async_manual_reset_event&&) = delete;
  async_manual_reset_event& operator=(const async_manual_reset_event&) = delet
  async_manual_reset_event& operator=(async_manual_reset_event&&) = delete;

  bool is_set() const noexcept;

  struct awaiter;
  awaiter operator co_await() const noexcept;

  void set() noexcept;
  void reset() noexcept;

private:

  friend struct awaiter;

  // - 'this' => set state
  // - otherwise => not set, head of linked list of awaiter*.
  mutable std::atomic<void*> m_state;

};
```

Here we have a fairly straight-forward and simple interface. The main thing to note at this point is that it has an `operator co_await()` method that returns an, as yet, undefined type, `awaiter`.

Let's define the `awaiter` type now.

Defining the Awaiter

Firstly, it needs to know which `async_manual_reset_event` object it is going to be awaiting, so it will need a reference to the event and a constructor to initialise it.

It also needs to act as a node in a linked-list of `awaiter` values so it will need to hold a pointer to the next `awaiter` object in the list.

It also needs to store the `coroutine_handle` of the awaiting coroutine that is executing the `co_await` expression so that the event can resume the coroutine when it becomes 'set'. We

don't care what the promise type of the coroutine is so we'll just use a `coroutine_handle<>` (which is short-hand for `coroutine_handle<void>`).

Finally, it needs to implement the **Awaiter** interface, so it needs the three special methods: `await_ready`, `await_suspend` and `await_resume`. We don't need to return a value from the `co_await` expression so `await_resume` can return `void`.

Once we put all of that together, the basic class interface for `awaiter` looks like this:

```
struct async_manual_reset_event::awaiter
{
  awaiter(const async_manual_reset_event& event) noexcept
  : m_event(event)
  {}

  bool await_ready() const noexcept;
  bool await_suspend(std::experimental::coroutine_handle<> awaitingCoroutine)
  void await_resume() noexcept {}

private:

  const async_manual_reset_event& m_event;
  std::experimental::coroutine_handle<> m_awaitingCoroutine;
  awaiter* m_next;
};
```

Now, when we `co_await` an event, we don't want the awaiting coroutine to suspend if the event is already set. So we can define `await_ready()` to return `true` if the event is already set.

```
bool async_manual_reset_event::awaiter::await_ready() const noexcept
{
  return m_event.is_set();
}
```

Next, let's look at the `await_suspend()` method. This is usually where most of the magic happens in an awaitable type.

First it will need to stash the coroutine handle of the awaiting coroutine into the `m_awaitingCoroutine` member so that the event can later call `.resume()` on it.

Then once we've done that we need to try and atomically enqueue the awaiter onto the linked list of waiters. If we successfully enqueue it then we return `true` to indicate that we don't want to resume the coroutine immediately, otherwise if we find that the event has concurrently been changed to the 'set' state then we return `false` to indicate that the coroutine should be resumed immediately.

```
bool async_manual_reset_event::awaiter::await_suspend(
  std::experimental::coroutine_handle<> awaitingCoroutine) noexcept
{
  // Special m_state value that indicates the event is in the 'set' state.
  const void* const setState = &m_event;

  // Remember the handle of the awaiting coroutine.
  m_awaitingCoroutine = awaitingCoroutine;

  // Try to atomically push this awaiter onto the front of the list.
  void* oldValue = m_event.m_state.load(std::memory_order_acquire);
  do
  {
    // Resume immediately if already in 'set' state.
    if (oldValue == setState) return false;

    // Update linked list to point at current head.
    m_next = static_cast<awaiter*>(oldValue);

    // Finally, try to swap the old list head, inserting this awaiter
    // as the new list head.
  } while (!m_event.m_state.compare_exchange_weak(
              oldValue,
              this,
              std::memory_order_release,
              std::memory_order_acquire));

  // Successfully enqueued. Remain suspended.
  return true;
}
```

Note that we use ‘acquire’ memory order when loading the old state so that if we read the special ‘set’ value then we have visibility of writes that occurred prior to the call to ‘set()’.

We require ‘release’ sematics if the compare-exchange succeeds so that a subsequent call to ‘set()’ will see our writes to m_awaitingCoroutine and prior writes to the coroutine state.

Filling out the rest of the event class

Now that we have defined the `awaiter` type, let’s go back and look at the implementation of the `async_manual_reset_event` methods.

First, the constructor. It needs to initialise to either the ‘not set’ state with the empty list of waiters (ie. `nullptr`) or initialise to the ‘set’ state (ie. `this`).

```
async_manual_reset_event::async_manual_reset_event(
  bool initiallySet) noexcept
: m_state(initiallySet ? this : nullptr)
{}
```

Next, the `is_set()` method is pretty straight-forward - it's 'set' if it has the special value `this`:

```
bool async_manual_reset_event::is_set() const noexcept
{
  return m_state.load(std::memory_order_acquire) == this;
}
```

Next, the `reset()` method. If it's in the 'set' state we want to transition back to the empty-list 'not set' state, otherwise leave it as it is.

```
void async_manual_reset_event::reset() noexcept
{
  void* oldValue = this;
  m_state.compare_exchange_strong(oldValue, nullptr, std::memory_order_acquire
}
```

With the `set()` method, we want to transition to the 'set' state by exchanging the current state with the special 'set' value, `this`, and then examine what the old value was. If there were any waiting coroutines then we want to resume each of them sequentially in turn before returning.

```
void async_manual_reset_event::set() noexcept
{
  // Needs to be 'release' so that subsequent 'co_await' has
  // visibility of our prior writes.
  // Needs to be 'acquire' so that we have visibility of prior
  // writes by awaiting coroutines.
  void* oldValue = m_state.exchange(this, std::memory_order_acq_rel);
  if (oldValue != this)
  {
    // Wasn't already in 'set' state.
    // Treat old value as head of a linked-list of waiters
    // which we have now acquired and need to resume.
    auto* waiters = static_cast<awaiter*>(oldValue);
    while (waiters != nullptr)
    {
      // Read m_next before resuming the coroutine as resuming
      // the coroutine will likely destroy the awaiter object.
      auto* next = waiters->m_next;
```

```
      waiters->m_awaitingCoroutine.resume();
      waiters = next;
    }
  }
}
```

Finally, we need to implement the `operator co_await()` method. This just needs to construct an `awaiter` object.

```
async_manual_reset_event::awaiter
async_manual_reset_event::operator co_await() const noexcept
{
  return awaiter{ *this };
}
```

And there we have it. An awaitable asynchronous manual-reset event that has a lock-free, memory-allocation-free, `noexcept` implementation.

If you want to have a play with the code or check out what it compiles down to under MSVC and Clang have a look at the source on godbolt.

You can also find an implementation of this class available in the cppcoro library, along with a number of other useful awaitable types such as `async_mutex` and `async_auto_reset_event`.

Closing Off

This post has looked at how the `operator co_await` is implemented and defined in terms of the **Awaitable** and **Awaiter** concepts.

It has also walked through how to implement an awaitable async thread-synchronisation primitive that takes advantage of the fact that awaiter objects are allocated on the coroutine frame to avoid additional heap allocations.

I hope this post has helped to demystify the new `co_await` operator for you.

In the next post I'll explore the **Promise** concept and how a coroutine-type author can customise the behaviour of their coroutine.

Thanks

I want to call out special thanks to Gor Nishanov for patiently and enthusiastically answering my many questions on coroutines over the last couple of years.

And also to Eric Niebler for reviewing and providing feedback on an early draft 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.