Creating other types of synchronization objects that can be used with co_await, part 2: The basic library

devblogs.microsoft.com/oldnewthing/20210310-00

March 10, 2021

Raymond Chen

Last time, I teased a library for building awaitable synchronization objects. It builds on the code had earlier written for one-shot events by distilling the pattern to its essence and then rebuilding it in a more generic way.

We start with the linked list nodes which are used to keep track of coroutines who are waiting for the synchronization object.

```
namespace async_helpers::impl
{
    struct node_base
    {
        node_base* next;
        node_base* prev;
    };

    struct node_handle : node_base
    {
        std::experimental::coroutine_handle<> handle{};
    };

    template<typename Extra>
    struct node : node_handle
    {
        template<typename... Args>
        node(Args&&... args) :
            extra(std::forward<Args>(args)...) {}

        Extra extra;
    };
}
```

The `node_base` is the node for a doubly-linked list. Derived from that is a `node_handle` that also carries a coroutine handle payload, and further derived from that is a `node<Extra>` which allows us to attach other payload to the node.

Next comes the linked list of nodes:

```
namespace async_helpers
{
    struct node_list : impl::node_base
    {
        node_list() : node_base{ this, this } {}
        node_list(node_list const&) = delete;
        void operator=(node_list const&) = delete;

        bool empty() const noexcept
        {
            return next == this;
        }

        void append_node(impl::node_base& node) noexcept
        {
            node.next = this;
            node.prev = prev;
            prev->next = std::addressof(node);
            prev = std::addressof(node);
        }

        bool append_list(node_list& other) noexcept
        {
            if (other.empty()) return false;
            other.prev->next = next;
            next->prev = other.prev;
            next = other.next;
            next->prev = this;
            other.next = other.prev = std::addressof(other);
            return true;
        }

        impl::node_base* peek_head() const noexcept
        {
            return empty() ? nullptr : next;
        }

        impl::node_base* try_remove_head() noexcept
        {
            if (empty()) return nullptr;
            auto node = next;
            next = node->next;
            next->prev = this;
            return node;
        }
    };
}
```

These are unexciting operations on doubly-linked lists. We have to write them ourselves because the C++ standard library uses non-intrusive linked lists, but we want intrusive linked lists to avoid the error conditions associated with memory allocation failures.

The node list is represented by a sentinel node that marks the beginning/end of the list. There is a little quirkiness in that I use `std::addressof` instead of the `&` operator, out of an abundance of caution, in case somebody decides to overload the `&` operator.

```
namespace async_helpers
{
    // Specialize if you need extra data for co_await.
    template<typename State> struct extra_await_data {};
}
```

We provide a traits class that lets you attach extra information to your `awaiter` to record additional context about awaiting. This will come in handy when we try to implement reader/writer locks.

The real excitement is in our `state` object. The intended usage is that you derive your full state object from the basic one, and the basic one uses CRTP to allow your full state object to customize certain operations. Last time, we saw how we could use this pattern for a one-shot event.

We'll look at the `awaitable_state` a little bit at a time.

```
namespace async_helpers
{
    template<typename State>
    class awaitable_state
    {
        std::mutex mutex;
        node_list sentinel;

        State& parent() { return static_cast<State&>(*this); }

        auto& extra_node(impl::node_base& node)
        { static_cast<impl::node<extra_await_data>&>(node); }
```

An awaitable object's state consists of a list of waiting coroutines, protected by a mutex. The helper function `parent()` helps with CRPT by downcasting the `awaitable_state` to the full `State` object. Similarly, `extra_node` takes a `node_base` and downcasts it all the way to a `node<extra_await_data>` so we can access the full node complete with extra data.

```
    public:
        using extra_await_data = extra_await_data<State>;
        using node_list = async_helpers::node_list;
```

For convenience, we give names to the extra data and the node list itself.

```
bool fast_claim(extra_await_data const&) const noexcept
{ return false; }
```

The `fast_claim` method is optional. If the derived class doesn't implement it, then we provide a default implementation that says "Nope, must do it the slow way."

```
extra_await_data* peek_head() const noexcept
{
    auto node = sentinel.peek_head();
    return node ? &extra_node(node)->extra : nullptr;
}
```

The derived class can use `peek_head` to peek at the extra data associated with the coroutine at the head of the wait list. If the wait list is empty, then the method returns `nullptr`. This method may be called only from within an action, since it requires that the lock be held.

Also from within an action, the derived class can ask for one waiting coroutine to be resumed:

```
bool resume_one(node_list& list) noexcept
{
    auto node = sentinel.try_remove_head();
    if (!node) return false;
    list.append_node(*node);
    return true;
}
```

This is done by unlinking the head node from the waiting list and appending it to the resume list. We append to the tail of the resume list to preserve FIFO.

Another thing an action can do is ask for all of the waiting coroutines to be released.

```
bool resume_all(node_list& list) noexcept
{
    return list.append_list(sentinel);
}
```

This is equivalent to calling `resume_one` in a loop until it fails, but we can do it more efficiently by moving the entire list at once.

When a coroutine awaits the synchronization object, we ask `await_suspend` to do the work:

```
bool await_suspend(
    std::experimental::coroutine_handle<> handle,
    impl::node<extra_await_data>& node)
{
    auto guard = std::lock_guard(mutex);
    if (parent().claim(node.extra)) return false;
    node.handle = handle;
    sentinel.append_node(node);
    return true;
}
```

The caller provides a coroutine handle and a preallocated `node`. We enter the lock and try to claim the synchronization object. If successful, then we are done, and we return `false` to tell the coroutine machinery that the coroutine should resume immediately. Otherwise, we remember the coroutine handle and append the node to the list of waiters, returning `true` to complete the suspension.

```
void await_resume(
    impl::node<extra_await_data>& node) noexcept
{
    node.handle = nullptr;
}
```

When the coroutine resumes, we null out the coroutine handle so that we know that the coroutine is no longer suspended and that we therefore are no longer in the cancellation case. I'll discuss later why we do it in `await_resume`.

```
void destruct_node(impl::node_base& node) noexcept
{
    if (node.handle) {
        auto guard = std::lock_guard(mutex);
        node.next = node.prev->next;
        node.prev = node.next->prev;
    }
}
```

This is part of our accommodation for cancellation: In the case that a waiting coroutine is destroyed, we check if it has a pending resumption handle. If so, then we are in the cancellation case, and we unlink it from the list of waiters. There is a race here: If the coroutine has already been scheduled for resumption, our attempt to unlink it will corrupt memory because the `prev` and `next` members of the node point to already-resumed coroutines. However, as we discussed earlier, if this happens, then it means that the caller was already in a race condition, trying to destroy a coroutine as it is resuming. It is the caller's responsibility to ensure that destroying a suspended coroutine happens only when it can guarantee that the coroutine is not at risk of resumption.[1]

The next bit is the part that makes actions happen:

```
    template<typename... Params, typename... Args>
    void action_impl(
        void (State::*handler)(node_list&, Params...),
        Args&&... args)
    {
        node_list list;
        {
            auto guard = std::lock_guard(mutex);
            (parent().*handler)(list,
                std::forward<Args>(args)...);
        }
        resume_list(list);
    }
```

We create an empty node list outside the lock, take the lock, and then call the handler, forwarding all the parameters. When the handler returns, we drop the lock and resume all the coroutines it had requested to be resumed, using this `resume_list` method:

```
private:
    void resume_list(node_list& list)
    {
        auto node = list.next;
        while (node != std::addressof(list))
        {
            resume_node(std::exchange(node, node->next));
        }
    }

    void resume_node(impl::node_base* node) noexcept
    {
        extra_node(*node).handle.resume();
    }
};
```

Resuming a list consists of calling `resume_node` on each node in the list. Note that the node is owned by the coroutine, so resuming the coroutine will cause the awaiter to be destructed, and the node will disappear with it. We therefore have to make sure that we do not touch the node after resuming the coroutine. In this case, it means that we advance to the `next` node and pull out the `handle` before resuming the handle.

This example resumes the coroutines synchronously. We'll work on asynchronous resumption next time.

The last piece is defining the synchronization object itself:

```
    template<typename State>
    class awaitable_sync_object
    {
        std::shared_ptr<State> shared;

    public:
        template<typename... Args>
        awaitable_sync_object(Args&&... args) :
            shared(std::make_shared<State>(
                std::forward<Args>(args)...)) {}

        template<typename Arg = typename State::extra_await_data>
        auto make_awaiter(Arg arg = {})
        { return awaiter{ *shared, std::forward<Arg>(arg) }; }

        auto operator co_await() { return awaiter{ *shared }; }

    protected:
        using state = State;

        State& get_state() const noexcept { return *shared; }

        template<typename... Args>
        void action_impl(Args&&... args) const
        {
            shared->action_impl(std::forward<Args>(args)...);
        }

        struct awaiter
        {
            template<typename... Args>
            awaiter(State& state, Args&&... args)
                : s(state), node(std::forward<Args>(args)...) {}

            State& s;
            impl::node<extra_await_data<State>> node;

            bool await_ready()
            { return s.fast_claim(node.extra); }

            bool await_suspend(
                std::experimental::coroutine_handle<> handle)
            { return s.await_suspend(handle, node); }

            void await_resume()
            { s.await_resume(node); }

            ~awaiter() { s.destruct_node(node); }
        };
    };
}
```

The `awaitable_sync_object` is a wrapper around a `shared_ptr` of the `State`. The shared state is constructed by forwarding all of the `awaitable_sync_object` constructor parameters to the `State` constructor, so that you can construct the `State` object with any default initial state. For example, semaphores may want to be initialized with an initial token count and possibly also a maximum token count.

There is also a `make_awaiter` method for creating the awaiter with optional `extra_await_data`, if you need to pass additional context for a particular `await` operation.

And the last public method is the `co_await` operator which creates our custom-defined `awaiter`.

The protected methods are for the custom synchronization object to use as part of its implementation. The `get_state()` method lets the implementation access its own state. The `action_impl` forwards its parameters to the state's `action_impl` method, which we discussed earlier.

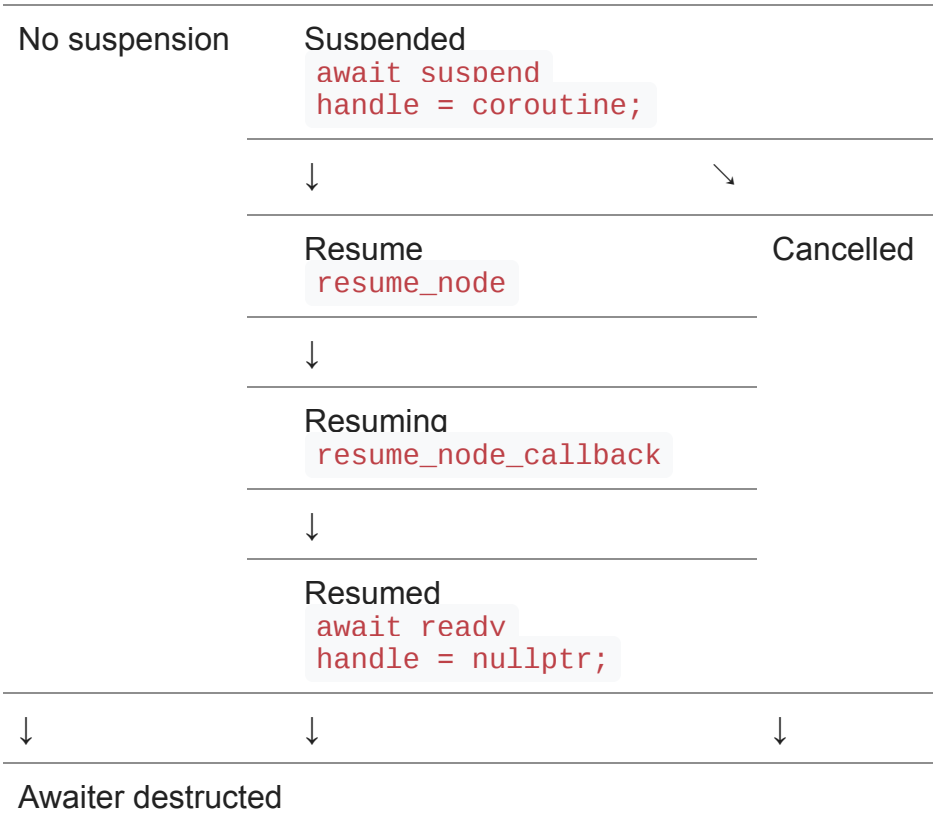
The `awaiter` is generalized so that you can construct the extra data in the node. The `await_ready` tries to do a `fast_claim`, which might allow the caller to bypass suspension if it was able to claim the object immediately. The `await_suspend` method forwards to the state, which we discussed earlier. The `await_resume` method asks the `State` to clean up, and recall that the method sets the `handle` to null to indicate that the coroutine is no longer suspended. We use that fact in the awaiter's destructor to detect the case where the coroutine is destroyed while suspended: In that case, the `handle` will still hold the coroutine's continuation (which is being abandoned), and that tells us to unlink this coroutine from the list of waiters.

There is a race condition here, in the case that a coroutine is destroyed while a resumption is in flight, but as I noted earlier, this is a bug in the caller, so we don't need to defend against it particularly hard. I chose to null out the `handle` in `await_resume` because that is more likely to be optimized out by the compiler, seeing as the awaiter's destructor runs immediately after `await_resume`, which increases the likelihood that the compiler will be able to propagate the value into the awaiter's destructor and realize that the code path is dead in the case of normal resumption.

Now, we could have nulled out the coroutine handle at many points in the lifetime of the awaiter. Why did I choose to do it in `await_resume`? Let's sketch out the various possible fates of the awaiter:

Awaiter constructed
`handle = nullptr;`

↓ ↓



In the case where the coroutine doesn't suspend because it was able to claim the synchronization object, the `handle` starts out as `nullptr` and stays that way until it is destructed. I'm hoping the optimizer can observe that the `handle` is not modified through this non-suspending path. The destructor's call to `destruct_node` will therefore do nothing, since it does work only if the handle is non-null, and can consequently be completely optimized out.

In the case where the coroutine is cancelled, the coroutine handle is set to a non-null value in `await_suspend` after the coroutine has suspended. When the coroutine is cancelled, the coroutine's destructor destructs the awaiter, and at that point, the `destruct_node` observes a non-null handle and knows to unlink the node from the list of waiters.

The last case is the one down the middle, and it's the most complicated one: The synchronization object cannot be claimed, so the `handle` gets set to the coroutine handle after the coroutine suspends. Later, some code signals the synchronization object, and it resumes all of the waiters. Eventually, we get to the `await_ready` which is called inside the now-resumed coroutine, and it sets the `handle` back to null to indicate that everything is back to normal. By setting the `handle` to null at the last moment before destruction, I'm hoping the optimizer can recognize that the destructor's call to `destruct_node` will do nothing, since it does work only if the handle is non-null, and can be completely optimized out.

It looks like gcc 10.1 and the Microsoft Visual C++ 19.24 compiler both can take my hints, and they do do optimize out the `destruct_node` calls in the two cases I noted. So hooray for that.

Okay, that was a whirlwind tour of our little library for writing generic awaitable synchronization objects. I did mention that this version resumes coroutines synchronously. Next time, we'll make it resume coroutines on a thread pool.

[1] We could defend against this by having `resume_list` make the node point to itself, so that if the race occurs, we don't corrupt memory immediately. But really, all we are doing is delaying the corruption, because the resumption will try to resume a destroyed coroutine, which will corrupt memory in a different way. Maybe we should `std::terminate`.

Raymond Chen

Follow