Creating a task completion source for a C++ coroutine: Producing a result

devblogs.microsoft.com/oldnewthing/20210323-00

March 23, 2021

Raymond Chen

We've been looking at creating different types of awaitable synchronization objects. This time, we'll create something analogous to what C# calls a `TaskCompletionSource` and what PPL calls a `task_completion_event`. For lack of a better name, I'm going to call it a `result_holder`.

A `result_holder` is an object that you can put a result into, and you can `co_await` it to wait for the result to appear. Once a result has been set, it can be retrieved multiple times. You can use this sort of object for one-time initialization, or if you want to cache the results of earlier calculations.

First, we need to teach our library about coroutines that return values. Up until now, the result of a `co_await` had always been `void`.

```
template<typename State>
class awaitable_state
{
    ...

    void get_result() const noexcept { }

    auto await_resume(
        impl::node<extra_await_data>& node) noexcept
    {
        node.handle = nullptr;
        return parent().get_result();
    }
    ...
};

template<typename State>
class awaitable_sync_object
{
    ...

    struct awaiter
    {
        ...

        auto await_resume()
        { return s.await_resume(node); }

        ...
    }
};
```

We allow the CRTP client to implement a method `get_result`, and whatever that method returns is the result of the `co_await`. By default, it's just `void`, but we're going to override it in our `result_holder`.

```
template<typename T>
struct result_holder_state :
    async_helpers::awaitable_state<result_holder_state<T>>
{
    std::atomic<bool> ready{ false };

    union optional
    {
        optional() {}
        ~optional() {}

        T value;
    } result;

    result_holder_state() {}
    result_holder_state(result_holder_state const&) = delete;
    void operator=(result_holder_state const&) = delete;

    ~result_holder_state()
    {
        if (ready.load(std::memory_order_relaxed)) {
            result.value.~T();
        }
    }
```

We build our own equivalent of `std::optional<T>` that supports querying atomically whether a value has been set. The atomic boolean `ready` becomes `true` when a value is set, and the union `result` holds the value if so. We use a union because unions do not construct or destruct their members by default. But it means that we must remember to do the construction and destruction ourselves.

This is not a general-purpose atomic `optional` because it supports only one-way transitions: You can go from unset to set, but once set, it's stuck forever. This limitation allows the discriminant (`ready`) to be atomic.

```
    using typename result_holder_state::extra_await_data;
    using typename result_holder_state::node_list;
```

Since our state type is now a template type, we have to tell the compiler which identifiers are dependent names. We may as well just import them to save ourselves some typing.

```
    bool fast_claim(extra_await_data const&) noexcept
    {
        return ready.load(std::memory_order_acquire);
    }

    bool claim(extra_await_data const&) noexcept
    {
        return ready.load(std::memory_order_relaxed);
    }
```

If someone tries to `co_await`, we let the await complete immediately if the value is already ready.

```
    void set_result(node_list& list, T v)
    {
        if (!ready.load(std::memory_order_relaxed)) {
            new (std::addressof(result.value))
                T{ std::move<T>(v) };
            ready.store(true, std::memory_order_release);
            this->resume_all(list);
        }
    }
```

To set the result, we first check that we don't have a result. If so, then we do nothing. You can set the result only once. Otherwise, we would have a race condition if one coroutine fetches the value while another is changing it.

If this is the first time anyone is setting the result, then we move the value into our private storage, using the placement new constructor. We provide the storage address via `std::addressof` to protect against the possibility that `T` has an overloaded `operator&`.

Only after the value is definitely set into our private storage do we mark the value as `ready`, and we do it with release semantics so that the effects of the constructor are fully visible before telling everybody that it's ready to be read.

It's also important to be aware that the constructor of `T` may throw an exception. In that case, the storage is destructed back to its uninitialized state, and the exception escapes. Another reason it's important not to set `ready` or to add coroutines to the `list` before the value is definitely constructed.

```
    T get_result()
    {
        return result.value;
    }
};
```

And here's where we override `get_result` so that the result of a `co_await` is the captured value.

We technically need an acquire fence here to ensure that all the changes to `value` made by the `set_result` are visible to the current. We get away without one because we put an acquire fence in `await_ready`!

```
template<typename T>
struct result_holder
    : async_helpers::awaitable_sync_object<
        result_holder_state<T>>
{
    using typename result_holder::state;

    void set_result(T result) const noexcept
    {
        this->action_impl(&state::set_result,
            std::move(result));
    }
};
```

The object itself is not particularly exciting. Setting the result on the main object moves the value into the state.

Now you have an object that you can put results into, and `co_await` ing it will wait until results appear.

```
result_holder<int> universe;

// coroutine 1:
auto answer = co_await universe;

// coroutine 2:
universe.set_result(42);
```

But this code is still broken.

We'll look more closely next time.

Raymond Chen

Follow