C++ coroutines: Adding COM context support to our awaiter

devblogs.microsoft.com/oldnewthing/20210429-00

April 29, 2021

Raymond Chen

You may want to have awaiters that apply custom resume behavior. For example, in Windows, you are likely to want your awaiter to preserve the COM thread context. For X11 programming, you may want to the awaiter to return to the render thread if the `co_await` was initiated from the render thread. Today we'll add the ability to customize the awaiter to our coroutine promise type.

The idea is that instead of registering just a coroutine handle for resumption, we register information that lets us resume the coroutine in some scenario-specific manner.

There are a couple of possible designs for this.

One is to register an object with a virtual `resume()` method, and we just call that `resume` method when it's time to resume the awaiting coroutine. This has the disadvantage of introducing a virtual function call, which affects speculatability as well as adding a control flow guard check due to the use of a function pointer.

Another option is to use a switch statement with one case for each type of resumption. This avoids the virtual function call, but it also means that add a new kind of awaiter requires that the base class be updated to understand it. Furthermore, it means that even if you don't use a particular awaiter, the code for it is still compiled into the promise, because the promise doesn't know at compile time which cases are going to end up being dead code at run time.

So I split the difference: I register a function pointer and a `void*` pointer. The function pointer is called with the `void*` pointer to do whatever it needs in order to resume the awaiting coroutine. This plug-in model makes the code pay-for-play: If you never use an awaiter, its code doesn't get compiled in. It also makes it easy to add new types of awaiters in the future. To avoid paying for the function pointer call, we also adopt the special convention that if the function pointer is `nullptr`, then the `void*` pointer is assumed to be the address of the awaiting coroutine. This lets us take a function pointer call out of the common case.

```
namespace async_helpers::details
{
    template<typename T>
    struct simple_promise_base
    {
        void (*m_resumer)(void*);
        std::atomic<void*> m_waiting{ cold_ptr };
        simple_promise_result_holder<T> m_holder;
```

We add a "resumer" function pointer to our promise. This is the function that knows how to resume a coroutine given the `m_waiting` pointer.

```
        void resume_waiting_coroutine(void* waiting) const
        {
            if (m_resumer)
            {
                m_resumer(waiting);
            }
            else
            {
                std::experimental::coroutine_handle<>::
                    from_address(waiting).resume();
            }
        }
```

When it comes time to resume the waiting coroutine, we check if there is a custom resumer. If so, then we call it with the `waiting` pointer and trust it to know what to do next. Otherwise, we assume that the pointer is the address of a coroutine and just resume it.

We now teach our final awaiter about this new convention for resuming the awaiting coroutine.

```
        auto final_suspend() noexcept
        {
            struct awaiter : std::experimental::suspend_always
            {
                simple_promise_base& self;
                void await_suspend(
                    std::experimental::coroutine_handle<>)
                    const noexcept
                {
                    auto waiter = self.m_waiting.exchange(completed_ptr,
                        std::memory_order_acq_rel);
                    if (waiting != abandoned_ptr) self.destroy();
                    if (waiting != running_ptr)
                        self.resume_waiting_coroutine(waiting);
                }
            };
            return awaiter{ {}, *this };
        }
```

Instead of treating the `waiting` as a coroutine pointer, we ask `resume_waiting_coroutine` to resume it in the awaiter-requested manner.

```
auto client_await_suspend(
    void* waiting,
    void (*resumer)(void*) = nullptr)
{
    m_resumer = resumer;
    assert(reinterpret_cast<uintptr_t>(waiting) >
                            reinterpret_cast<uintptr_t>(cold_ptr));
    return m_waiting.exchange(waiting,
        std::memory_order_acq_rel) == running_ptr;
}

auto cold_client_await_suspend(
    void* waiting,
    void (*resumer)(void*) = nullptr)
{
    start();
    client_await_suspend(waiting, resumer);
}
```

Suspension requires the `waiting` pointer and an optional resumer function. If there is no resumer function, then `waiting` is assumed to be a pointer to a coroutine. We assert that the `waiting` doesn't match any of our special sentinel values, so we won't get confused later.

Now we can update our awaiters:

```
template<typename T>
struct promise_awaiter
{
    promise_ptr<T> self;

    bool await_ready()
    {
        return self->client_await_ready();
    }

    auto await_suspend(std::experimental::coroutine_handle<> handle)
    {
        return self->client_await_suspend(handle.address());
    }

    T await_resume()
    {
        return self->client_await_resume();
    }
};

template<typename T>
struct cold_promise_awaiter
{
    promise_ptr<T> self;

    bool await_ready()
    {
        return self->cold_client_await_ready();
    }

    auto await_suspend(std::experimental::coroutine_handle<> handle)
    {
        return self->cold_client_await_suspend(handle.address());
    }

    T await_resume()
    {
        return self->client_await_resume();
    }
};
```

The awaiters pass the coroutine by address rather than handle.

Now that we have the plug-in model set up, we can add a new kind of awaiter, which I'll call a `com_promise_awaiter`. This one ensures that we resume in the same COM context.

```
template<typename T>
struct com_promise_awaiter
{
    com_promise_awaiter(promise_ptr<T>&& ptr)
        : self(std::move(ptr))
    {
    }

    promise_ptr<T> self;
    std::experimental::coroutine_handle<> waiter;
    wil::com_ptr<IContextCallback> context;

    bool await_ready()
    {
        return self->client_await_ready();
    }

    auto await_suspend(std::experimental::coroutine_handle<> handle)
    {
        waiter = handle;
        THROW_IF_FAILED(CoGetObjectContext(IID_PPV_ARGS(&context)));
        return self->client_await_suspend(this, resume_in_context);
    }

    T await_resume()
    {
        return self->client_await_resume();
    }

    static auto as_self(void* p)
    {
        return reinterpret_cast<com_promise_awaiter*>(p);
    }

    static void resume_in_context(void* parameter)
    {
        as_self(parameter)->resume_context();
    }

    void resume_context()
    {
        ComCallData data{};
        data.pUserDefined = this;
        auto local_context = std::move(context);
        THROW_IF_FAILED(local_context->ContextCallback(
            resume_apartment_callback, &data,
            IID_ICallbackWithNoReentrancyToApplicationSTA, 5, nullptr));
    }

    static HRESULT CALLBACK resume_apartment_callback(
        ComCallData* data) noexcept
    {
```

```
            as_self(data->pUserDefined)->waiter();
            return S_OK;
        }
    };
```

Most of this code is related to COM context management and doesn't really illustrate the point that we have a plug-in model for awaiting.

The magic happens in the `await_suspend` method: We remember the handle to resume in our new `waiter` member variable, and capture the current COM context in the new `context` member variable. Once that's done, we can call `client_await_suspend`, but instead of passing the coroutine handle, we pass our own address, and also pass a custom resumer function, which we've called `resume_in_context`.

When it's time to resume the awaiting coroutine, the `resume_in_context` function recovers the original `com_promise_awaiter` and asks the `context` to resume execution in the captured context. The `resume_apartment_callback` runs in that captured context, and it resumes the coroutine.

There is a subtlety here: The awaiting coroutine resumes once we invoke `waiter()`, and at resumption, the coroutine will destruct the `com_promise_awaiter`. If we hadn't captured `context` into `local_context`, that would have resulted in the `context` being destroyed while its `ContextCallback` method was still running, a violation of one of the basic rules of programming, namely that function parameters are stable for the lifetime of the function call. (In this case, the function parameter is the implied `this` pointer.) Capturing it into a local variable prevents the `context` from being destructed when the `com_promise_awaiter` destructs, and instead waits until `ContextCallback` returns.

The above code is an illustration, and it does technically work, but there are a number of optimizations that a real program would want to perform to avoid unwanted stack build-up or to avoid the synchronous apartment-changing callback problem we discussed some time ago. Patching up these problems is an important exercise, but not really within the scope of this series on coroutines.

We can now hook up this new promise to a `com_aware_task`.

```
namespace async_helpers
{
    template<typename T>
    struct com_aware_task : details::simple_task_base<T>
    {
        using base = details::simple_task_base<T>;
        com_aware_task() = default;
        com_aware_task(details::simple_promise<T>* initial)
            : base(initial) { this->promise->start(); }

        void swap(com_task& other)
        {
            std::swap(this->promise, other.promise);
        }

        using base::operator co_await;

        auto operator co_await() &&
        {
            return details::com_promise_awaiter<T>
                { std::move(this->promise) };
        }
    };

    template<typename T>
    void swap(com_aware_task<T>& left, com_aware_task<T>& right)
    {
        left.swap(right);
    }
}

template <typename T, typename... Args>
struct std::experimental::coroutine_traits<async_helpers::com_aware_task<T>, Args...>
{
    using promise_type = async_helpers::details::simple_promise<T>;
};
```

This is identical to our `simple_task` except that its `co_await` operator uses a `com_promise_awaiter` instead of a a `promise_awaiter`.

Since the `simple_task` and `com_aware_task` differ only in their `co_await`, we can actually make them use each other's awaiter.

```
namespace async_helpers::details
{
    template<typename T>
    struct simple_task_base
    {
        ...

        auto resume_same_context() &&
        {
            return com_promise_awaiter<T>
                { std::move(promise) };
        }

        auto resume_any_context() &&
        {
            return promise_awaiter<T>
                { std::move(promise) };
        }

        ...
    };
}

namespace async_helpers
{
    template<typename T>
    struct simple_task : details::simple_task_base<T>
    {
        ...

        auto operator co_await() &&
        {
            return std::move(*this).resume_any_context();
        }
    };

    template<typename T>
    struct com_aware_task :
    {
        ...

        auto operator co_await() &&
        {
            return std::move(*this).resume_same_context();
        }
    };
}
```

This means that if you have a `simple_task` or `com_aware_task`, but you want to resume in a context different from the one that you normally get with `co_await`, you can ask explicitly for the awaiter that gives you the behavior you desire:

```
extern async_helpers::simple_task<void> Something1Async();

// resumes in any context by default
co_await Something1Async();

// same as above, but more explicit
co_await Something1Async().resume_any_context();

// forces resumption in same COM context
co_await Something1Async().resume_same_context();

extern async_helpers::com_aware_task<void> Something2Async();

// resumes in same COM context by default
co_await Something2Async();

// same as above, but more explicit
co_await Something2Async().resume_same_context();

// forces resumption in any context
co_await Something2Async().resume_any_context();
```

We'll look at synchronous waits next time.

Raymond Chen

Follow