Ordering asynchronous updates with coroutines, part 1: Mutual exclusion

devblogs.microsoft.com/oldnewthing/20210906-00

September 6, 2021

Raymond Chen

I opined some time ago on the perils of holding a lock across a coroutine suspension point. But say you have a bunch of asynchronous activity that you want to serialize. How can you do that without a lock?

There are a few different scenarios in which you may need to protect asynchronous activity. We'll look at them over the next few days.

We'll start with the case of wanting to serialize a collection of actions that are unrelated to each other, but which access a shared resource. For this, you can use an awaitable event.

```
awaitable_event s_mutex;

IAsyncAction DoThing1Async()
{
    co_await s_mutex;
    auto done = wil::scope_exit([&] { s_mutex.set(); });

    auto checkpoint = shared_database.create_checkpoint();
    co_await shared_database.Part1();
    co_await shared_database.Part2();
    co_await shared_database.Part3();
    checkpoint.reset(); // made it to the end; commit the changes
}

IAsyncAction DoThing2Async()
{
    co_await s_mutex;
    auto done = wil::scope_exit([&] { s_mutex.set(); });

    auto checkpoint = shared_database.create_checkpoint();
    co_await shared_database.Step1();
    co_await shared_database.Step2();
    checkpoint.reset(); // made it to the end; commit the changes
}
```

There are two different things that can be done, and we want to make sure only one thing happens at a time, say, because we want each of the things to operate atomically on the database (with rollback on failure). We protect the database access with one of our fancy `awaitable_event` objects and make sure to set the event when we're done.

If you are using C++/WinRT, you don't have a ready-made awaitable event, but you can fake it with a kernel event.

```
winrt::handle s_mutex(winrt::check_handle(CreateEvent(nullptr, false, false,
nullptr)));

IAsyncAction DoThing1Async()
{
    co_await winrt::resume_on_signal(s_mutex.get());
    auto done = wil::scope_exit([&] { SetEvent(s_mutex.get()); });

    auto checkpoint = shared_database.create_checkpoint();
    co_await shared_database.Part1();
    co_await shared_database.Part2();
    co_await shared_database.Part3();
    checkpoint.reset(); // made it to the end; commit the changes
}

IAsyncAction DoThing2Async()
{
    co_await winrt::resume_on_signal(s_mutex.get());
    auto done = wil::scope_exit([&] { SetEvent(s_mutex.get()); });

    auto checkpoint = shared_database.create_checkpoint();
    co_await shared_database.Step1();
    co_await shared_database.Step2();
    checkpoint.reset(); // made it to the end; commit the changes
}
```

The C++/WinRT `resume_on_signal` function returns an awaitable that completes when the kernel event is signaled. Immediately after we claim the event, we use an RAII type to make sure that the event gets signaled when we're done.

Okay, that's the easy pattern. It's a straightforward translation of synchronous code to a coroutine. But there are cases where you don't want to make the operations wait for each other, such as the case where a subsequent operation invalidates a previous one. We'll look at one of those patterns next time.

Raymond Chen

Follow