A simpler version of the task sequencer that doesn't promise fairness

devblogs.microsoft.com/oldnewthing/20231208-00

December 8, 2023

Raymond Chen

Some time ago, we wrote a `task_sequencer` for forcing a series of asynchronous operations to run one after another.

It turned into quite a bit of code, but that was in part because it has to preserve the ordering of tasks, and because it's trying to operate purely inside the language with coroutine handles and not rely on external libraries.

But if you're willing to build on the C++/WinRT library and you aren't concerned about fairness, then you can do it in an easier way.

```
winrt::handle g_serializerHandle{
  winrt::check_pointer(CreateEvent(nullptr,
  /* manual reset */ false, /* initial state */ true,
  nullptr)) };

struct SetEvent_scope_exit
{
    SetEvent_scope_exit(void* handle) : handle(handle) {}
    ~SetEvent_scope_exit() {
        SetEvent(handle);
    }
    void* handle;
}

winrt::IAsyncAction DoSomethingAsync()
{
    ⟦ do some stuff ⟧

    // Only one instance of this next block of code can run at a time.
    {
        co_await winrt::resume_on_signal(g_serializerHandle.get());
        auto next = SetEvent_scope_exit(g_serializerHandle.get());

        ⟦ do some more stuff, including co_await ⟧
    }
    // End of serialization region.

    ⟦ do even more stuff ⟧
}
```

We create an auto-reset event handle called `g_serializerHandle` and use it to control access to the protected code region. To enter the region, you claim the event handle, and to exit the region, you signal the event handle, which then allows someone else to enter the region. We use a custom RAII type to ensure that the event is signaled even if an exception is thrown.

This is the same as the traditional synchronous code, but we use a kernel object because that allows us to use `resume_on_signal` to await asynchronously on the event. This kernel object must be an event (or a semaphore, which is a generalization of an auto-reset event) rather than a Win32 mutex for two reasons.

First, Win32 mutexes have thread affinity, and we can't guarantee that all of our asynchronous work occurs on a single thread. Second, Win32 mutexes are reentrant, so it would allow a second attempt to claim the mutex to succeed if the attempt was made on the same thread that claimed the mutex for the work already in progress.

If you are willing to use WIL (the Windows Implementation Library), then you can use the `unique_event` and `SetEvent_scope_exit` from WIL.

```
// Default is auto-reset, unsignaled. We change it to signaled.
wil::unique_event g_serializerHandle{ wil::EventOptions::Signaled };

winrt::IAsyncAction DoSomethingAsync()
{
    ⟦ do some stuff ⟧

    // Only one instance of this next block of code can run at a time.
    {
        co_await winrt::resume_on_signal(g_serializerHandle.get());
        auto next = wil::SetEvent_scope_exit(g_serializerHandle.get());

        ⟦ do some more stuff, including co_await ⟧
    }
    // End of serialization region.

    ⟦ do even more stuff ⟧
}
```

Note that this pattern is unfair: There is no guarantee that the waiting coroutines will be released in the order of arrival. But if fairness is not important, this is a lot less work than the fully-general `task_sequencer` from last time.

Bonus chatter: Having to remember to follow up the `co_await resume_on_signal` with a `SetEvent_scope_exit` is cumbersome. You can remedy that by packing it all into a single method.

```
struct async_unfair_mutex
{
    wil::unique_event m_serializerHandle
        { wil::EventOptions::Signaled };

    wil::task<wil::event_set_scope_exit> LockAsync()
    {
        co_await winrt::resume_on_signal(m_serializerHandle.get());
        co_return wil::SetEvent_scope_exit(m_serializerHandle.get());
    }
};
```

We can’t use `winrt::IAsyncOperation<wil::event_set_scope_exit>` because `IAsyncOperation` works only with Windows Runtime types. We also can’t use `concurrency::task<wil::event_set_scope_exit>` because PPL’s `task` requires that the task return type be default-constructible (okay) and copyable (nope).