C++ coroutines: The problem of the DispatcherQueue task that runs too soon, part 3

devblogs.microsoft.com/oldnewthing/20191225-00

December 25, 2019

Raymond Chen

Last time, we fixed a race condition in C++/WinRT's `resume_foreground(Dispatcher-Queue)` function when it tries to resume execution on a dispatcher queue. We did this by having the queued task wait until `await_ suspend` was finished before allowing the coroutine to resume. The hard part was finding a place to put the synchronization object, and we ended up putting it in the queued task's lambda.

But it turns out there's another place we can put it, and it was in front of us the whole time.

We can put it in the awaiter.

```
auto resume_foreground(DispatcherQueue const& dispatcher)
{
  struct awaitable
  {
    DispatcherQueue m_dispatcher;
    bool m_queued = false;
    slim_event ready;

    bool await_ready()
    {
      return false;
    }

    bool await_suspend(coroutine_handle<> handle)
    {
      bool result = m_dispatcher.TryEnqueue([this, handle]
        {
          ready.wait();
          handle();
        });
      m_queued = result;
      ready.signal();
      return result;
    }

    bool await_resume()
    {
      return m_queued;
    }
  };
  return awaitable{ dispatcher };
}
```

The awaiter is destructed when the coroutine resumes, and the coroutine resumes either when the coroutine handle is invoked (which we do in the lambda), or when the `await_ ready` or `await_ suspend` methods indicate that the resumption should proceed immediately.

Therefore, members of the awaiter are good as long as we haven't done any of those things.

In the case of a successful `TryEnqueue` , the `await_ suspend` is not going to ask for immediate resumption. The resumption will occur when the lambda invokes the handler. But our lambda waits for `async_ suspend` to signal the event before invoking the handler, so the awaiter is still valid at that point.

In the case of a failed `TryEnqueue` , the lambda will never run. Instead, the coroutine resumes after `await_ suspend` returns with a value of `false` . Prior to the return, the awaiter is still valid, so we can signal the event. (Mind you, there's nobody listening for the signal.)

This is a lot simpler than trying to dig the event out of the lambda.

However, there is still quite a bit of overhead to this plan: While it's true that the `slim_event` runs entirely in user mode in the non-blocking case, it still generates a good number of memory barriers. The `slim_event::signal` method uses a memory barrier in order to ensure that `m_queued` is made visible to all threads before setting `signaled` to true. If it didn't do this, then the `wait` function could see that `signaled` is true and run ahead with the wrong value of `m_queued`.

Furthermore, after the `signal` method updates the `signaled` member, it calls `WakeByAddressAll`, which will itself erect another memory barrier to make sure it found all the waiters (even if there are none).

The memory barriers are necessary to ensure that the updated value of `m_queued` that was written by the `await_suspend` can be ready by `await_ready` in the case that the coroutine continues on the dispatcher thread.

Next time, we'll get rid of the memory barriers.

Raymond Chen

Follow