C++ coroutines: The problem of the DispatcherQueue task that runs too soon, part 4

devblogs.microsoft.com/oldnewthing/20191226-00

December 26, 2019

Raymond Chen

Last time, we made another attempt to fix a race condition in C++/WinRT's `resume_foreground(DispatcherQueue)` 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, and we found a nice place to put the synchronization object, namely in the awaiter, but even with that fix, we introduced additional memory barriers into the hot code path.

But it turns out all this work was unnecessary. We just had to look at the problem a different way.

The purpose of storing the result of `TryEnqueue` into `m_queued` is so that `await_resume` can report whether the lambda was queued or not. But we can infer that information another way: The fact that our lambda is running means that got got queued. Because if the lambda were not queued, then it would never have run in the first place.

This allows us to simplify the awaiter by making the lambda responsible for reporting that it was queued.

```
bool await_suspend(coroutine_handle<> handle)
{
  // m_queued =
  return
    m_dispatcher.TryEnqueue([this, handle]
    {
      m_queued = true;
      handle();
    });
  // return m_queued;
}
```

There are two cases to consider:

First, the `TryEnqueue` could fail. In that case, `await_suspend` returns `false`, and `m_queued` continues to have its original value (which is also `false`). The coroutine resumes immediately on the same thread, and the `await_ready` will return `m_queued`, which is `false`. The value of `m_queued` correctly reports that the lambda was not queued.

Otherwise, `TryEnqueue` succeeded, and this is the more interesting case. Since `await_suspend` does not access any member variables after calling `TryEnqueue`, it doesn't matter whether the lambda runs before or after `await_suspend` returns.

The `await_suspend` returns `true` because the lambda was queued, and this permits the suspension of the coroutine to proceed. Nobody has updated `m_queued`, so it still has its initial value of `false`. This is an incorrect state of affairs, but that's okay: We'll fix it before anybody notices.

When the lambda runs, it sets `m_queued` to `true`. This restores balance to the universe by bringing the `m_queued` member variable to a value consistent with what actually happened. Only after repairing `m_queued` do we invoke the `handle`. The two operations (updating `m_queued` and invoking the `handle`), so we don't have a race condition between the setting of `m_queued` and its observation in `await_ready`.

You could say that we lazy-updated the `m_queued` member variable. It's not safe to update it in `await_suspend`, so we wait until the lambda. We didn't have to pass the value of `true` explicitly to the lambda, because the lambda knows that `true` is the only value it could possibly be if the lambda is running.

That wraps up our introduction to C++ coroutines. I haven't even gotten a chance to get into promises and the other infrastructure needed to *create* coroutines.[1] So far, we've just been looking at the infrastructure needed to create awaitable objects. Someday, I'll write about promises, but I'm going to take a break for a bit.

Bonus chatter: Notice how my initial instinct for fixing this problem was writing fifty-some-odd lines of code. But stopping to think let me shrink it to about half that. And then stepping back and looking at the bigger issue allowed me to fix the problem by making small changes to two lines of code.

[1] This means that we will have to wait before we learn about the mysterious step 1 in the search for an awaiter.

Raymond Chen

Follow