C++ coroutines: The problem of the DispatcherQueue task that runs too soon, part 1

devblogs.microsoft.com/oldnewthing/20191223-00

December 23, 2019

Raymond Chen

I was experiencing occasional crashes in C++/WinRT's `resume_foreground` function when it tries to resume execution on a dispatcher queue. Here's a simplified version of that function:

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

    bool await_ready()
    {
      return false;
    }

    bool await_suspend(coroutine_handle<> handle)
    {
      m_queued = m_dispatcher.TryEnqueue([handle]
        {
          handle();
        });
      return m_queued;
    }

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

All you need to know about the `DispatcherQueue` object is that the `TryEnqueue` method takes a delegate and schedules it to run on the dispatcher queue's thread. If it is unable to do so (say, because the thread has already exited), then the function returns `false`. The return

value of the `TryEnqueue` method is the result of the `co_await`.

Let's walk through how this function is intended to work.

The `resume_foreground` method returns an object that acts as its own awaiter. When a `co_await` occurs, the coroutine first calls `await_ready`, which returns `false`, meaning "Go ahead and suspend me."

Next, the coroutine calls `await_suspend`. This method tries to queue the resumption of the coroutine onto the dispatcher thread and remembers whether it succeeded in the `m_queued` member variable.

Returning the value of `m_queued` means that if the continuation was successfully scheduled (`true`), the coroutine remains suspended until it is resumed when the handle is invoked. On the other hand, if the continuation was not successfully scheduled (`false`), then the suspension is abandoned, and execution resumes immediately on the same thread.

Either way, when the coroutine resumes, it is told whether the rescheduling onto the dispatcher thread succeeded.

Okay, now that you see how it is intended to work, can you spot the defect?

This code violates one of the rules we gave when we were getting started with awaitable objects: Once you arrange for the `handle` to be called, you cannot access any member variables because the coroutine may have resumed before `async_suspend` finishes.

And that's what's happening here: The dispatcher queue is running the lambda even before the `async_suspend` can save the answer into `m_queued`. As a result, the code crashes (if you're lucky) or corrupts memory (if you're not).

So we need to make sure the lambda doesn't race ahead of `async_suspend`.

Next time, we'll make our first attempt to fix this.

(The fact that I call it our *first* attempt gives you a clue that it may take more than one try.)

Raymond Chen

Follow