C++ coroutines: The problem of the DispatcherQueue task that runs too soon, part 2

devblogs.microsoft.com/oldnewthing/20191224-00

December 24, 2019

Raymond Chen

Last time, we discovered a race condition in C++/WinRT's `resume_foreground(DispatcherQueue)` function when it tries to resume execution on a dispatcher queue. Let's try to fix it.

As a reminder, here's where the problem is:

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

The core of the problem is that the lambda may run to completion (which includes destructing the awaiter) before `await_suspend` gets a chance to save the result of `TryEnqueue()` into `m_queued`, resulting in a store to a freed object.

We need to make sure the lambda waits for `await_suspend` to finish its work before the lambda proceeds with the resumption of the coroutine.

We also need to be mindful that this is a rare race condition, so we want to keep things fast in the common case.

This suggested to me that we should use a lightweight synchronization primitive, rather than a heavy object like a kernel event. If we can solve the entire problem without taking a kernel transition, that would be great.

So let's start with a lightweight synchronization object: The single byte of memory whose address can be waited on. (We've done this before.)

```
struct slim_event
{
  slim_event() = default;

  // Not copyable
  slim_event(slim_event const&) = delete;
  void operator=(slim_event const&) = delete;

  bool signaled = false;

  void signal()
  {
    std::atomic_thread_fence(std::memory_order_release);
    WakeByAddressAll(&signaled);
  }

  void wait()
  {
    // Wait for "signaled" to be "not false" (i.e., true)
    bool False = false;
    while (!signaled) {
      WaitOnAddress(&signaled, &False, sizeof(False), INFINITE);
    }
    std::atomic_thread_fence(std::memory_order_acquire);
  }
};
```

We need to share this synchronization object between `await_suspend` and the lambda. Where can we put it?

- In the lambda.
- In the `await_suspend` function.
- In neither: On the heap as a `shared_ptr`.

I'm not going to make a new heap allocation for it, because that costs us part of the benefit of using a lightweight synchronization primitive in the first place.

What about in the `await_suspend` function?

The `await_suspend` function returns quickly, whereas the lambda may sit in the queue for a long time before finally running. If we put the slim event in the `await_suspend` function, it will have to wait for the lambda to be finished with the slim event before it can safely destruct it and return.

That leaves the lambda. It's okay to make the lambda wait for `await_suspend`, because it won't need to wait long, and most of the time it won't need to wait at all.

What we want to be able to do is get the address of a variable that is sitting inside a lambda.

Let's admit it: There's no easy way to do it. (There are hard ways, though.)

I came up with this idea: As the lambda gets moved from the call site into the function parameter, and then moved from the function parameter into the delegate, we make the lambda keep track of where it has been moved to, and update a variable shared by reference. Note that this requires that the lambda stop moving once it has been placed inside a delegate.

```
struct tracked_slim_event
{
 tracked_slim_event(slim_event*& p)
    : tracker(p) { tracker = &value; }
 tracked_slim_event(tracked_slim_event&& other)
    : tracker(other.tracker) { tracker = &value; }

 slim_event*& tracker;
 slim_event value;
};
```

The `tracked_slim_event` wraps a `slim_event` but also manages a "tracker", which is a pointer that is set to point to the slim event when the object is created, and updated whenever the object is moved. This lets you find the object's final resting place.

Updating a constructor parameter is a sort of inversion of control: Instead of having a method that tells you the answer upon request, you pass in the thing that receives the answer, and the object updates it as the answer changes.

Now we can give the `await_suspend` function access to the `slim_event` hiding inside the lambda.

```
    bool await_suspend(coroutine_handle<> handle)
    {
      slim_event* finder;
      bool result = m_dispatcher.TryEnqueue(
        [handle, tracker = slim_event_tracker(finder)] mutable
        {
          tracker.value.wait();
          handle()
        });

      m_queued = result;

      finder->value.signal();
      return result;
    }
```

We save the result of `TryEnqueue` into `m_queued` while we still can. Only after it has been safely stored do we release the lambda by signaling the slim event.

This is a lot of work to address a race condition. And it turns out I missed a spot when listing the various places we can keep the synchronization object. We'll continue the discussion next time.

Raymond Chen

Follow