On the perils of holding a lock across a coroutine suspension point, part 3: Solutions

devblogs.microsoft.com/oldnewthing/20210709-00

July 9, 2021

Raymond Chen

So far, we've been looking at the horrible things that can happen if you hold a lock across a coroutine suspension point. What can we do to avoid the problem? For reference, here's the code in question:

```
IAsyncAction MyObject::RunOneAsync()
{
  std::lock_guard guard(m_mutex);

  if (!m_list.empty() {
    auto& item = m_list.front();
    co_await item.RunAsync();
    item.Cleanup();
    m_list.pop_front();
  }
}
```

You need to limit the use of the synchronous lock to synchronous code. In the case of our sample function above, we can extract the item to run while under the lock, but run the item outside the lock.

```
IAsyncAction MyObject::RunOneAsync()
{
  std::optional<Item> item;
  {
    std::lock_guard guard(m_mutex);
    if (!m_list.empty()) {
      item.emplace(std::move(m_list.front()));
      m_list.pop_front();
    }
  }
  if (item) {
    co_await item->RunAsync();
    item->Cleanup();
  }
}
```

This operation is bit easier if you have direct access to the list nodes, since you can just detach the head node from the list, operate on the contents, and then destruct it when done. However, the above mechanism generalizes better to non-node-based containers like `std::vector` or `std::dequeue`.

To avoid the awkwardness of the braces whose sole purpose is to control the scope of the lock guard, you could pull it into lambda:

```
IAsyncAction MyObject::RunOneAsync()
{
  auto item = [&]() -> std::optional<Item> {
    std::lock_guard guard(m_mutex);
    if (m_list.empty()) return {};
    auto item = std::move(m_list.front());
    m_list.pop_front();
    return item;
  }();
  if (item) {
    co_await item->RunAsync();
    item->Cleanup();
  }
}
```

Or pull the lambda into a separate function:

```
std::optional<Item> MyObject::PopFrontItem()
{
  std::lock_guard guard(m_mutex);
  if (m_list.empty()) return {};
  auto item = std::move(m_list.front());
  m_list.pop_front();
  return item;
}

IAsyncAction MyObject::RunOneAsync()
{
  auto item = PopFrontItem();
  if (item) {
    co_await item->RunAsync();
    item->Cleanup();
  }
}
```

Here's another function that uses a lock to make sure nobody sees a partially-initialized widget:

```
IAsyncAction MyObject::ReloadWidgetAsync()
{
  // code in italics is wrong
  std::lock_guard guard(m_mutex);
  m_widget = Widget();
  co_await widget.SetColorAsync(m_color);
}
```

In this case, we could capture the necessary parameters under the lock, do the work outside the lock on the captured parameters, and then update the results inside the lock.

```
IAsyncAction MyObject::ReloadWidgetAsync()
{
  // Capture the color.
  Color color;
  {
    std::lock_guard guard(m_mutex);
    color = m_color;
  }

  // Create a brand new widget
  auto widget = Widget();
  co_await widget.SetColorAsync(color);

  // Save the results
  {
    std::lock_guard guard(m_mutex);
    m_widget = widget;
  }
}
```

Note that the operations that occur under the lock are all synchronous.

There is a bug in the above code: While we are setting up the new widget, somebody might change the color, and we end up reloading the widget with the old color instead of the new one. We'll have to re-check that the color is correct and retry if not.

```
IAsyncAction MyObject::ReloadWidgetAsync()
{
  // Capture the color.
  Color color;
  {
    std::lock_guard guard(m_mutex);
    color = m_color;
  }

  // Create a brand new widget
  auto widget = Widget();

  bool done = false;
  while (!done) {
    co_await widget.SetColorAsync(color);

    // Save the results if valid
    {
      std::lock_guard guard(m_mutex);
      if (color == m_color) {
        m_widget = widget;
        done = true;
      } else {
        // Get the current color and try again
        color = m_color;
      }
    }
  }
}
```

This pattern may look familiar: It's the same pattern we use for lock-free operations. Capture the inputs, perform some work, and then atomically set the result, provided the inputs haven't changed. If the inputs changed, then try again with the revised inputs.

You may not be able to use this pattern, though. Maybe widgets are expensive to construct, and you don't want create extra ones unnecessarily. Or maybe the operation you want to perform doesn't fit the capture / work / update model.

In that case, you could use an awaitable synchronization object, like the one we developed some time ago.

Raymond Chen

Follow