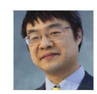
Fixing exception safety in our task_sequencer

devblogs.microsoft.com/oldnewthing/20250328-00

紅樓鍮 March 28, 2025

Raymond Chen



Some time ago, we developed a `task_sequencer` class for running asynchronous operations in sequence. There's a problem with the implementation of `QueueTaskAsync`: What happens if an exception occurs?

Let's look at the various places an exception can occur in `QueueTaskAsync`.

```
template<typename Maker>
auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
{
    auto current = std::make_shared<chained_task>();
    auto previous = [&]
    {
        winrt::slim_lock_guard guard(m_mutex);
        return std::exchange(m_latest, current); ← oops
    }();

    suspender suspend;

    using Async = decltype(maker());
    auto task = [](auto&& current, auto&& makerParam,
                   auto&& contextParam, auto& suspend)
                -> Async
    {
        completer completer{ std::move(current) };
        auto maker = std::move(makerParam);
        auto context = std::move(contextParam);

        co_await suspend;
        co_await context;
        co_return co_await maker();
    }(current, std::forward<Maker>(maker),
      winrt::apartment_context(), suspend);

    previous->continue_with(suspend.handle);

    return task;
}
```

If an exception occurs at `make_shared`, then no harm is done because we haven't done anything yet.

If an exception occurs when starting the lambda task, then we are in trouble. We have already linked the `current` onto `m_latest`, but we will never call `continue_with()`, so the chain of tasks stops making progress.

To fix this, we need to delay hooking up the `current` to the chain of `chained_task`s until we are sure that we have a task to chain. This means that the `std::exchange(m_latest, current);` needs to wait until the task is created. But we can't just swap the order of the two operations because the task does a `std::move(current)`, which empties out the `current`

```
template<typename Maker>
auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
{
    auto current = std::make_shared<chained_task>();

    suspender suspend;

    using Async = decltype(maker());
    auto task = [](auto&& current, auto&& makerParam,
                   auto&& contextParam, auto& suspend)
                -> Async
    {
        completer completer{ std::move(current) };
        auto maker = std::move(makerParam);
        auto context = std::move(contextParam);

        co_await suspend;
        co_await context;
        co_return co_await maker();
    }(current, std::forward<Maker>(maker),
      winrt::apartment_context(), suspend);

    // moved this to after we create the task
    auto previous = [&]
    {
        winrt::slim_lock_guard guard(m_mutex);
        return std::exchange(m_latest, current); ← still oops
    }();

    previous->continue_with(suspend.handle);

    return task;
}
```

The last person to use the `current` is allowed to move from it, so we have to move the move.

```
template<typename Maker>
auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
{
    auto current = std::make_shared<chained_task>();

    suspender suspend;

    using Async = decltype(maker());
    auto task = [](auto&& current, auto&& makerParam,
                   auto&& contextParam, auto& suspend)
                -> Async
    {
        completer completer{ current };
        auto maker = std::move(makerParam);
        auto context = std::move(contextParam);

        co_await suspend;
        co_await context;
        co_return co_await maker();
    }(current, std::forward<Maker>(maker),
      winrt::apartment_context(), suspend);

    auto previous = [&]
    {
        winrt::slim_lock_guard guard(m_mutex);
        return std::exchange(m_latest, std::move(current));
    }();

    previous->continue_with(suspend.handle);

    return task;
}
```

While we're here, we may as well do some cleaning up. For example, there was no need to create the `apartment_context` outside the lambda and move it into the lambda locals. We can just create it as a lambda local.

```
    auto task = [](auto&& current, auto&& makerParam,
                   /* auto&& contextParam, */ auto& suspend)
                -> Async
    {
        completer completer{ current };
        auto maker = std::move(makerParam);
        auto context = winrt::apartment_context();

        co_await suspend;
        co_await context;
        co_return co_await maker();
    }(current, std::forward<Maker>(maker),
      /* winrt::apartment_context(), */ suspend);
```

And instead of passing the other lambda parameters by reference, we can use a reference capture. We just have to be careful to copy them to locals before our first `co_await`. (And there was another bug in the original version: It moved the `maker` into the local rather than forwarding it.)

```
auto task = [&]() -> Async
{
    completer completer{ current };
    auto local_maker = std::forward<Maker>(maker);
    auto context = winrt::apartment_context();

    co_await suspend;
    co_await context;
    co_return co_await local_maker();
}(/* ⟦ ... ⟧ */);
```

And now that the last consumer of `current` is the `QueueTaskAsync` function itself, we can swap it instead of moving it out of one variable and into another.

```
/* auto previous = [&] */
{
    winrt::slim_lock_guard guard(m_mutex);
    m_latest.swap(current);
}/* () */

current->continue_with(suspend.handle);
```

The name `current` is confusing for a variable that actually holds the `previous` entry, so let's give it a generic name `node`.

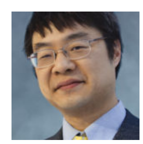
The result of all this cleanup is the following class. Everything outside `QueueTaskAsync` is unchanged, but I include it here for copy-paste-friendliness.

```
struct task_sequencer
{
    task_sequencer() = default;
    task_sequencer(const task_sequencer&) = delete;
    void operator=(const task_sequencer&) = delete;

private:
    using coro_handle = std::experimental::coroutine_handle<>;

    struct suspender
    {
        bool await_ready() const noexcept { return false; }
        void await_suspend(coro_handle h)
            noexcept { handle = h; }
        void await_resume() const noexcept { }

        coro_handle handle;
    };

    static void* completed()
    { return reinterpret_cast<void*>(1); }

    struct chained_task
    {
        chained_task(void* state = nullptr) : next(state) {}

        void continue_with(coro_handle h) {
            if (next.exchange(h.address(),
                        std::memory_order_acquire) != nullptr) {
                h();
            }
        }

        void complete() {
            auto resume = next.exchange(completed());
            if (resume) {
                coro_handle::from_address(resume).resume();
            }
        }

        std::atomic<void*> next;
    };

    struct completer
    {
        ~completer()
        {
            chain->complete();
        }
        std::shared_ptr<chained_task> chain;
    };
```

```
    winrt::slim_mutex m_mutex;
    std::shared_ptr<chained_task> m_latest =
        std::make_shared<chained_task>(completed());

public:
    template<typename Maker>
    auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
    {
        auto node = std::make_shared<chained_task>();

        suspender suspend;

        using Async = decltype(maker());
        auto task = [&]() -> Async
        {
            completer completer{ current };
            auto local_maker = std::forward<Maker>(maker);
            auto context = winrt::apartment_context();

            co_await suspend;
            co_await context;
            co_return co_await local_maker();
        }();

        {
            winrt::slim_lock_guard guard(m_mutex);
            m_latest.swap(node);
        }

        node->continue_with(suspend.handle);

        return task;
    }
};
```

Author

Raymond Chen



Raymond has been involved in the evolution of Windows for more than 30 years. In 2003, he began a Web site known as The Old New Thing which has grown in popularity far beyond his wildest imagination, a development which still gives him the heebie-jeebies. The Web site spawned a book, coincidentally also titled The Old New Thing (Addison Wesley 2007). He occasionally appears on the Windows Dev Docs Twitter account to tell stories which convey no useful information.

3 comments

Join the discussion.

- FM

  Since we have “deducing this” now, can we rewrite the async lambda to capture by value and also take `this` by value?

  NB

  LB 1 week ago

  Yes, but so far I don’t think any compilers support that yet. It is a good way to allow lambdas to be coroutines though.

- JM

  Jacob Manaker

  “And instead of passing the other lambda parameters by reference, we can use a reference capture. We just have to be careful to copy them to locals before our first co_await.”

  That strikes me as an argument in favor of explicit captures:

  ```
  auto task = [&maker, &current, &suspend]() -> Async { /* same */ }();
  ```

Stay informed

Get notified when new posts are published.