How can I write a coroutine that produces a result but keeps on running?

devblogs.microsoft.com/oldnewthing/20220707-00

July 7, 2022

Raymond Chen

A customer wanted to have a coroutine that produced a result but kept on running. Something like this:

```
task<std::shared_ptr<Party>>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    co_await party->invite_guests();
    co_await party->order_food();
    co_return_but_keep_running party; // hypothetical statement
    co_await victim->arrives();
    std::cout << "Surprise!" << std::endl;
}
```

The idea here is that somebody who asks to create a surprise party gets the Party object once the food has been ordered. However, the `CreateSurpriseParty` function isn't finished yet. It keeps on running, and then when the victim arrives, it announces a surprise.

How can you do this in a coroutine?

Well, there's the hard way and the easy way.

The hard way is to add an "early completion" feature to the `task`, so that you can write something like this:

```
task<std::shared_ptr<Party>>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    co_await party->invite_guests();
    co_await party->order_food();

    auto producer = co_await get_result_producer();
    producer.produce(party);

    co_await victim->arrives();
    std::cout << "Surprise!" << std::endl;

    co_return use_already_produced_value();
}
```

You can use the `await_transform` coroutine feature so that the result of the `co_await get_result_producer()` is a special object that communicates with the promise, in this case by telling it to produce a result.

The producer object's `produce` method would do the same thing as the promise's `return_value`: It would take the argument and save it in the promise's result holder, for `await_resume` to return.

The trick is that you don't want to wait until `final_suspend` to wake the awaiting coroutine. You want to do it right away. This means that you need to update the state machine to accommodate the new scenario: The promise has resumed its awaiting coroutine, but the current coroutine is still running. One way to do this is to go back to the old pattern where we used reference counting. The `produce` method would resume the awaiting coroutine, but would *not* decrement the reference count, because we don't do that until we get to `final_suspend`.

Another thing to do is add another overload to `return_value` so that the coroutine can say "Hi, I want to finish now." In my above example, I used a custom object called `use_already_produced_value` as the signal.

An annoying quirk of the coroutine specification is that a promise cannot have both `return_void` and `return_value` methods. If that were legal, then we could say that `co_return` with no arguments is how you declare that your coroutine is truly finished. (This also aligns with the principle that falling off the end of a coroutine is equivalent to `co_return` with no arguments.)

But wait, we're not done yet!

There's a gotcha here, because a `producer.produce()` that occurs while a lock is held will cause the awaiting coroutine to be resumed *under the lock*. That is just asking for a lock inversion deadlock. This is exactly the same problem we ran into when we tried implementing `return_value` in our custom promise type, and which led to us deferring the waking of the waiting coroutine until `final_suspend`.

Maybe you can code your way out of that jam, either by simply requiring that callers not hold any blocking resources when they call `produce()`, or maybe changing `produce()` so it resumes the awaiting coroutine immediately while transfering the current coroutine to a background thread, so the usage pattern would be

```
    co_await producer.produce_and_resume_background(party);
```

But wait, we got so distracted by how we could implement this as a coroutine that we overlooked a much simpler solution. The coroutine is a red herring!

How can I write a normal (non-coroutine) function that returns a value to its caller but keeps on running?

Well, the way you would do that is to queue up the extra work to run asynchronously before returning to the caller.

```
std::shared_ptr<Party>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    party->invite_guests();
    party->order_food();

    // arrange to be called back when the victim arrives
    victim.on_arrival([]() {
        std::cout << "Surprise!" << std::endl;
    });

    return party;
}
```

So just do the same thing in your coroutine.

```
task<std::shared_ptr<Party>>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    co_await party->invite_guests();
    co_await party->order_food();

    // arrange to be called back when the victim arrives
    victim.on_arrival([]() {
        std::cout << "Surprise!" << std::endl;
    });

    co_return party;
}
```

"Okay, Mister Smarty Pants, but what if there is no callback-based mechanism for continuing asynchronously?"

You can use a fire-and-forget coroutine.

```
task<std::shared_ptr<Party>>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    co_await party->invite_guests();
    co_await party->order_food();

    continue_asynchronously_on_background_thread(party, victim);

    co_return party;
}

winrt::fire_and_forget
continue_asynchronously_on_background_thread(
    std::shared_ptr<Party> party,
    std::shared_ptr<Person> victim)
{
    co_await winrt::resume_background();
    co_await victim.arrives();
    std::cout << "Surprise!" << std::endl;
}
```

You may be tempted to move the work inline, but be careful that your lambda doesn't capture any variables.

Bonus reading: CppCoreGuidelines: CP.51: Do not use capturing lambdas that are coroutines.

```
task<std::shared_ptr<Party>>
CreateSurpriseParty(std::shared_ptr<Person> victim)
{
    auto party = std::make_shared<Party>();
    co_await party->invite_guests();
    co_await party->order_food();

    // The rest happens in parallel on a background thread.
    [](auto party, auto victim) -> winrt::fire_and_forget
    {
        co_await winrt::resume_background();
        co_await victim.arrives();
        std::cout << "Surprise!" << std::endl;
    }(party, victim);

    co_return party;
}
```

Raymond Chen

Follow