Creating other types of synchronization objects that can be used with co_await, part 3: Parallel resumption

devblogs.microsoft.com/oldnewthing/20210311-00

March 11, 2021

Raymond Chen

Last time, we developed a library for building awaitable synchronization objects. I noted that when the coroutines are released, they are resumed in sequence, which means that one coroutine can prevent others from progressing. Let's resume them in parallel.

One option is to use `TrySubmitThreadpoolCallback` to put the resumption on the thread pool. In the `awaitable_state` class, replace the `resume_node` method with this version:

```
    static void CALLBACK resume_node_callback(
        PTP_CALLBACK_INSTANCE, void* context) noexcept
    {
        std::experimental::coroutine_handle<>::
            from_address(context).resume();
    }

    void resume_node(impl::node_base* node) noexecpt
    {
        if (!TrySubmitThreadpoolCallback(
            resume_node_callback,
            extra_node(*node).handle.address(),
            nullptr))
        {
            std::terminate(); // fatal
        }
    }
```

Instead of resuming the handle immediately and synchronously, we submit a callback to the thread pool, and have the callback resume the coroutine.

This works, but there is a problem if `TrySubmitThreadpoolCallback` fails, since we have no way to report an error to the caller. All we can do is terminate the process.

An alternative is to use the `CreateThreadpoolWork` / `SubmitThreadpoolWork` pattern which has the advantage of front-loading all of the error conditions. That way, we can throw a low memory exception at the point of the `await` rather than finding ourselves stuck when it comes time to resume.

Our `node_handle` now babysits a threadpool work item:

```
struct node_handle : node_base
{
    PTP_WORK work{};
};
```

This member records the work item that we will use to resume the coroutine. It is non-null if the coroutine is on the synchronization object's wait list. We set this up as part of the suspension:

```
    bool await_suspend(
        std::experimental::coroutine_handle<> handle,
        impl::node<extra_await_data>& node)
    {
        auto guard = std::lock_guard(mutex);
        if (parent().claim(node.extra)) return false;
        node.work = check_pointer(
            CreateThreadpoolWork(resume_node_callback,
                handle.address(), nullptr));
        sentinel.append_node(node);
        return true;
    }
```

When we realize that we need to suspend, we create a work item that will perform the resumption. We can raise a low memory exception at this point, and it will be captured into the caller.

Resuming the coroutine node consists of submitting the work:

```
    void resume_node(impl::node_base* node) noexcept
    {
        SubmitThreadpoolWork(extra_node(*node).work);
    }
```

And we move the work item cleanup into the callback function:

```
static void CALLBACK resume_node_callback(
    PTP_CALLBACK_INSTANCE, void* context, PTP_WORK work)
    noexcept
{
    CloseThreadpoolWork(work);
    std::experimental::coroutine_handle<>::
        from_address(context).resume();
}
```

The work can be closed at any time after it is submitted: Closing a submitted work item does not cancel the outstanding work. We don't want to slow down the `resume_list` method, so we make the work item responsible for its own bookkeeping: That way, the cost is paid by the resuming coroutine rather than the signaling one.

The other bit of bookkeeping is nulling out the `work` now that it's been closed.

```
void await_resume(
    impl::node<extra_await_data>& node) noexcept
{
    node.work = nullptr;
}
```

And finally, we tweak our abandonment detection:

```
void unlink_node(impl::node_base& node) noexcept
{
    auto work = extra_node(*node).work;
    if (node.work) {
        CloseThreadpoolWork(work);
        auto guard = std::lock_guard(mutex);
        node.next = node.prev->next;
        node.prev = node.next->prev;
    }
}
```

There is an additional optimization decision to be made here, which is finding the best place to close the work item. Here's the diagram again:

Awaiter constructed `work = nullptr;`		
↓	↓	

No suspension	Suspended
	`await_suspend`
	`work = work item;`	

	↓	↘

	Resume `resume_node`	Abandoned
	↓	
	Resuming `resume_node_callback` work closed	
	↓	
	Resumed `await_ready` `work = nullptr;`	
↓	↓	↓
Awaiter destructed		

The analyses for the no-suspend path and the abandonment path are the same as last time. The extra decision in the center path is deciding when to close the work item. I decided to do it in `resume_node_callback`: I definitely want the work item to be responsible for closing its own work. That avoids adding extra responsibilities to the signaling coroutine, which is only fair because you don't want to bog down the signaling code with work that wasn't even its idea! And to reduce code size, I want closing the work item to be done in shared code, which in this case is the thread pool work item callback itself. That same callback is going to be used for all resumptions of all nodes used by any client of this library. If closing the work item had been moved to `await_resume`, then that would get inlined into every coroutine's resumption code.

Okay, that was perhaps a deeper dive than you wanted into the subject of creating an awaitable synchronization object. But now that I have this whole thing created, I want to drive it around a bit. We'll start that next time.

Raymond Chen

Follow