Creating other types of synchronization objects that can be used with co_await, part 1: The one-shot event

devblogs.microsoft.com/oldnewthing/20210309-00

March 9, 2021

Raymond Chen

So far, we've been looking at how we could build a one-shot event that can be used with `co_await`. There are other types of synchronizations objects you may want to use with coroutines, so let's write a library for creating all sorts of awaitable synchronization objects.

The intended usage is that you define your awaitable synchronization object in terms of state and actions. (The actions are things that alter the state.) Here's an example for the one-shot event we've been using so far:

```
struct awaitable_oneshot_event_state :
    async_helpers::awaitable_state<awaitable_oneshot_event_state>
{
    // Private state: Has the event been signaled?
    std::atomic<bool> signaled{ false };

    // optional fast path: Return true if wait is satisfied.
    bool fast_claim(extra_await_data const&) const noexcept
    {
        return signaled.load(std::memory_order_acquire);
    }

    // required slow path: Return true if wait is satisfied.
    // Called with the lock held.
    bool claim(extra_await_data const&) const noexcept
    {
        return signaled.load(std::memory_order_relaxed);
    }

    // action: Alter state and resume coroutines.
    void set(node_list& list)
    {
        signaled.store(true, std::memory_order_relaxed);
        resume_all(list);
    }
};
```

In our case, we have a single boolean of state which records whether the event has been signaled. The optional `fast_claim` method provides a fast path that doesn't take the lock: It peeks at the signaled state and returns `true` if the event is already signaled, indicating that no waiting needs to be done at all. We need to use acquire semantics on the atomic boolean to ensure that no memory operations on the protected region are advanced ahead of the check of the signal. (Otherwise, we would be reading variables before they were ready.)

The regular `claim` method does the same thing, but it is done under the protection of a lock, so it can use relaxed ordering, since the lock will provide the necessary memory barriers. Loads and stores with relaxed ordering typically turn into simple loads and stores with no memory barriers, so this gives you the best performance on most systems.[1]

Our only action is signaling the event. The action is given a list of nodes to which it can add the coroutines it wishes to resume. In our case, we signal the event by setting the `signaled` state to `true` (using the efficient relaxed ordering because the ambient lock will protect us), and then asking to resume all the waiters.

Once you've defined your state, you can define your synchronization object which wraps the state. The wrapper makes the synchronization object copyable.

```
struct awaitable_oneshot_event
    : async_helpers::awaitable_sync_object<awaitable_oneshot_event_state>
{
    void set()
    {
        action_impl(&state::set);
    }
};
```

Our only action is setting the event, which we do by asking the `action_impl` helper method to forward the request into the state. When the action implementation in the state returns, any coroutines that were added to the list are resumed.

And that's it, an awaitable synchronization object. Now the hard part is writing the library that makes all of this possible. We'll do that next time.

[1] Even when relaxed loads and stores map to plain loads and stores, they are not entirely the same as plain loads and stores, because they cannot be coalesced by the optimizer.

```
std::atomic<int> value;

if (value.load(std::memory_order_relaxed) == 0 ||
    value.load(std::memory_order_relaxed) == 1) ...
```

The above sequence will issue two memory reads. If you want the reads to be coalesced, you need to coalesce them yourself.

```
auto capture = value.load(std::memory_order_relaxed);
if (capture == 0 || capture == 1) ...
```

Raymond Chen

Follow