Creating other types of synchronization objects that can be used with co_await, part 5: The auto-reset event

devblogs.microsoft.com/oldnewthing/20210315-00

March 15, 2021

Raymond Chen

Our next stop in showing off our library for building awaitable synchronization objects is the auto-reset event.

The auto-reset event is trickier because claiming a token also mutates the object's state. Many of our synchronization objects are of this ilk, so let's add a helper method to our template:

```
template<typename State>
class awaitable_state
{
...
public:
    template<bool fast, typename T,
        typename...Args, typename...Params>
    bool calc_claim(
        std::atomic<T>& value,
        bool (*transition)(T, T&, Params&&...),
        T initial,
        Args&&... args)
    {
        constexpr auto order = fast
            ? std::memory_order_acquire
            : std::memory_order_relaxed;
        for (;;) {
            T future;
            if (!transition(initial, future,
                std::forward<Args>(args)...)) {
                return false;
            }
            auto success = value.compare_exchange_weak(
                initial, future, order,
                std::memory_order_relaxed);
            if constexpr (fast) return success;
            else if (success) return success;
        }
    }
```

This helper function `calc_claim` implements a common state transition pattern: See if the object is in a claimable state, and if so, try to transition it to a claimed state. We do this by starting with an optimistic guess at the value, calculate the new state (by a `transition` method the CRTP derived class is expected to implement), and then try to transition atomatically to that state.

The transition function can return `false` to mean that the transition is not allowed, and we should suspend.

The `fast` version is performed outside the lock: It uses acquire semantics on the compare-exchange to ensure that the data protected by the synchronization object is not access prior to acquisition. The fast version also uses a weak compare-exchange and doesn't bother retrying on compare-exchange failure: In other words, it simply gives up after one try if the answer was "maybe". I figure that if I can't transition immediately, I may as well spend my memory barrier money on entering the lock for real.

The slow version runs inside the lock. Acquiring the lock already set up a memory barrier, so we already established the acquire barrier and don't need to set up another one. The slow version retries until gets a definite yes or no answer.

You might be tempted to use a simple `value.store()` to update the state under the lock, but that would be wrong because it could race against another thread that changed the state using the no-lock fast path.

The odd phrasing of the `return success;` code path is necessary to avoid "conditional expression is constant" warnings when written the somewhat less awkward way:

```
if (fast || success) return success;
```

The parameter after the transition function is an optimistic guess as to the initial value. In the case of an auto-reset event, we know that the only chance of a successful fast claim is if the current state is `true`, so we can just assume that it is and let the compare-exchange tell us if we were wrong.

Any additional parameters after the initial value are forwarded to the transition function as extra parameters.

```
template<bool fast, typename T,
    typename...Args>
bool calc_claim(
    std::atomic<T>& value,
    T initial,
    Args&&... args)
{
    return calc_claim<fast>(value,
        &State::transition,
        initial,
        std::forward<Args>(args)...);
}

template<bool fast, typename T,
    typename...Args, typename...Params>
bool calc_claim(
    std::atomic<T>& value,
    bool (*transition)(T, T&, Params&&...),
    Args&&... args)
{
    return calc_claim<fast>(value,
        transition,
        value.load(std::memory_order_relaxed),
        std::forward<Args>(args)...);
}

template<bool fast, typename T>
bool calc_claim(
    std::atomic<T>& value)
{
    return calc_claim<fast>(value,
        &State::transition,
        value.load(std::memory_order_relaxed));
}
```

Other types of synchronization objects may not have a clear single optimistic guess, so you can omit the initial value and it will just start with the atomic variable's current value. You can also omit the name of the transition function, and we'll look for a static method named `transition`.

Okay, now that we have those helpers, let's look at the auto reset event handle.

```
struct awaitable_auto_reset_event_state :
    async_helpers::awaitable_state<awaitable_auto_reset_event_state>
{
    awaitable_auto_reset_event_state(bool initial)
    : signaled(initial) {}

    std::atomic<bool> signaled;

    static bool transition(bool current, bool& future) noexcept
    {
        if (!current) return false;
        future = false;
        return true;
    }

    bool fast_claim(extra_await_data const&) noexcept
    {
        return calc_claim<true>(signaled, true);
    }

    bool claim(extra_await_data const&) noexcept
    {
        return calc_claim<false>(signaled);
    }

    void set(node_list& list) noexcept
    {
        if (!resume_one(list)) {
            signaled.store(true, std::memory_order_relaxed);
        }
    }
};

struct awaitable_auto_reset_event
    : async_helpers::awaitable_sync_object<awaitable_auto_reset_event_state>
{
    awaitable_auto_reset_event(bool initial = false) :
        awaitable_sync_object(initial) { }

    void set() noexcept
    {
        action_impl(&state::set);
    }

    void reset() noexcept
    {
        get_state().signaled.store(false,
            std::memory_order_release);
    }
};
```

In the case of an auto-reset event, the only case where the claim will succeeed is if the signal state is `true`, so we pass that as the optimistic initial value for the fast-claim case.

The other difference between the manual reset event and the auto-reset event is the behavior when there is a waiter: The auto-reset event releases at most one waiter, and if that happens, then the event remains unset. Only if there is nobody waiting on the event does it transition to the signaled state, ready to be consumed by the next awaiter.

That was trickier that I thought. Next time, we'll look at something that is basically an extended version of the auto-reset event: The semaphore.

Exercise: What is wrong with this version of `set`:

```
void set(node_list& list) noexcept
{
    signaled.store(true, std::memory_order_relaxed);
    if (resume_one(list)) {
        signaled.store(false, std::memory_order_relaxed);
    }
}
```

[1] I could have addressed this by creating a marker type called `start_with_current_value_t` or something like that, but I had no use for it right now, so I skipped it. You can add it yourself if it turns out you need it.

Raymond Chen

Follow