Creating other types of synchronization objects that can be used with co_await, part 6: The semaphore

devblogs.microsoft.com/oldnewthing/20210316-00

March 16, 2021

Raymond Chen

Our next stop in showing off our library for building awaitable synchronization objects is the semaphore. This will look very familiar because a semaphore with a maximum token count of 1 is the same thing as an auto-reset event, so we can just extend our auto-reset event implementation to support multiple tokens.

```
struct awaitable_semaphore_state :
    async_helpers::awaitable_state<awaitable_semaphore_state>
{
    awaitable_semaphore_state(unsigned int initial)
    : tokens(initial) {}

    std::atomic<unsigned int> tokens;

    static bool transition(
        unsigned int current, unsigned int& future) noexcept
    {
        if (!current) return false;
        future = current - 1;
        return true;
    }

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

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

    void set(node_list& list) noexcept
    {
        if (!resume_one(list)) {
            signaled.fetch_add(1, std::memory_order_relaxed);
        }
    }

    void set_many(node_list& list, unsigned int count) noexcept
    {
        for (; count && resume_one(list); --count) { }

        if (count) {
            tokens.fetch_add(count, std::memory_order_relaxed);
        }
    }
};

struct awaitable_semaphore
    : async_helpers::awaitable_sync_object<awaitable_semaphore_state>
{
    awaitable_semaphore(unsigned int initial = 0) :
        awaitable_sync_object(initial) { }

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

```
    }

    void set_many() noexcept
    {
        action_impl(&state::set_many);
    }
};
```

This is basically the same as an auto-reset event, except that the object state is a count of tokens instead of a single boolean value. Claiming a token involves decrementing the token count by one, rather than by "decrementing" a `true` to a `false`. Similarly, setting the semaphore increments the token count instead of "incrementing" the `false` to `true`.

Since semaphores can hold more than one token, we don't have a good "optimistic" value for the success case, so we just use the one-parameter version of `calc_claim` that uses the atomic variable's current value as its starting point.

Semaphores have the extra operation of "set many" which lets you set multiple tokens at once. We implement that by resuming as many waiters as we have tokens to resume, and then adding any leftover tokens to the token count for future consumption.

Netx time, we'll look at mutexes and recursive mutexes, which are quirky in the world of coroutines.

Raymond Chen

Follow