Creating other types of synchronization objects that can be used with co_await, part 9: The shared mutex (continued)

devblogs.microsoft.com/oldnewthing/20210319-00

March 19, 2021

Raymond Chen

Last time, we tried to make an awaitable shared mutex, but ran into some problems.

The first problem was that shared locks can starve out exclusive locks. To fix that, we say that you cannot steal the lock if there is already anybody waiting. You have to go to the end of the queue. We'll have to expose this information from the `awaitable_state` template type.

```
template<typename State>
class awaitable_state
{
...
public:
    bool any_waiters() const noexcept
    {
        return !sentinel.empty();
    }
...
};
```

You can check for waiters outside the lock, but the result may be stale by the time you look at it.

```
bool fast_claim(extra_await_data const& e) noexcept
{
    if (any_waiters()) return false;
    if (e.exclusive) {
        return calc_claim<true>(owners, exclusive_transition, 0);
    } else {
        return calc_claim<true>(owners, shared_transition);
    }
}

bool claim(extra_await_data const& e) noexcept
{
    if (any_waiters()) return false;
    if (e.exclusive) {
        return calc_claim<false>(owners, exclusive_transition);
    } else {
        return calc_claim<false>(owners, shared_transition);
    }
}
```

We add a check to say that claims always fail if there is somebody ahead of you in the wait queue.

Another problem we had was deal with races when the last shared lock is released. We fix that by temporarily marking the shared mutex as if it were exclusively owned, so that nobody can try to claim it. Since claims can come in without the lock, we need to go into a compare-exchange loop to go into this state only if we are releasing the last shared lock.

```
void unlock_shared(node_list& list)
{
    auto current = owners.load(std::memory_order_relaxed);
    int future;
    do {
        if (current == 1) future = -1;
        else future = current - 1;
    } while (owners.compare_exchange_weak(current, future,
        std::memory_order_relaxed);
    if (future == -1) {
        resume_a_bunch(list);
    }
}
```

When releasing a shared lock, we see if we are the last one. If so, then we temporarily switch to an exclusive lock (value −-1) to prevent anybody else from claiming the lock. Otherwise, we just decrement the shared lock count. Eventually, the compare-exchange will succeed, and if we ended up releasing the last shared lock, we proceed to resume a bunch of waiters.

There are no changes needed to `resume_a_bunch` because we were careful not to update the `owners` until all the work is done.

Well, that was annoying. But I'll make up for it by creating a new type of synchronization object that doesn't exist in Win32: The event where you can wait for it to become set *or* clear.

Bonus chatter: Technically, there is a data race on `sentinel` when we check whether it is empty outside the lock. One way to fix this is to make the sentinel links `std::atomic` with relaxed access. Another way is to encode additional information in the `owners`. For example, we might have

- −1: Owned exclusively.
- `INT_MIN` + *n*: Owned shared *n* times, with an exclusive waiter.
- 0: Not owned.
- *n*: Owned shared *n* times with no exclusive waiter.

The acquisition transitions are the same, with the extra wrinkle that when we add an element to the queue, we also set the sign bit in `owners`. And when releasing a shared lock, we need to mask off the sign bit before checking whether the lock has been released to zero.

Raymond Chen

Follow