Creating a co_await awaitable signal that can be awaited multiple times, part 3

devblogs.microsoft.com/oldnewthing/20210303-00

March 3, 2021

Raymond Chen

Last time, we created an awaitable signal that can be awaited multiple times, but noted that it took kernel transitions a lot. Let's implement the entire thing in user mode.

```
struct awaitable_event
{
  void set() const { shared->set(); }

  auto await_ready() const noexcept
  {
    return shared->await_ready();
  }

  auto await_suspend(
    std::experimental::coroutine_handle<> handle) const
  {
    return shared->await_suspend(handle);
  }

  auto await_resume() const noexcept
  {
    return shared->await_resume();
  }

private:
  struct state
  {
    std::atomic<bool> signaled = false;
    winrt::slim_mutex mutex;
    std::vector<std::experimental::coroutine_handle<>> waiting;

    void set()
    {
      std::vector<std::experimental::coroutine_handle<>> ready;
      {
        auto guard = winrt::slim_lock_guard(mutex);
        signaled.store(true, std::memory_order_relaxed);
        std::swap(waiting, ready);
      }
      for (auto&& handle : ready) handle();
    }

    bool await_ready() const noexcept
    { return signaled.load(std::memory_order_relaxed); }

    bool await_suspend(
      std::experimental::coroutine_handle<> handle)
    {
      auto guard = winrt::slim_lock_guard(mutex);
      if (signaled.load(std::memory_order_relaxed)) return false;
      waiting.push_back(handle);
      return true;
    }

    void await_resume() const noexcept { }
  };
```

```
  std::shared_ptr<state> shared = std::make_shared<state>();
};
```

The `awaitable_event` contains a `shared_ptr` to an internal `state` object, which is where all the work really happens. Operations on the `awaitable_event` are all forwarded to the `state` object, so all of the public methods are relatively uninteresting. The excitement happens in the `state` object, so let's focus on that.

To wait for the `awaitable_event`, we begin with `await_ready`, which returns whether the event is already signaled. If it is already signaled, then `await_ready` returns `true`, which bypasses the suspension entirely. An event that represents "initialization complete" will spend nearly all of its time in the signaled state, and this short-circuit gives an optimized path for the compiler so it doesn't have to spill register variables in the case that the event is already signaled.

If the event is not signaled, then we get to `await_suspend`. We take the lock and check a second time whether the event has been signaled. If so, then we return `false` meaning "I reject the suspension. Keep running."[1]

On the other hand, if the event is truly not signaled, then we push the coroutine handle onto our list of waiting coroutine handles, and we're done.

To signal the event, we take the lock, mark the event as signaled, and swap out the vector of waiting coroutine handles for an empty list. These coroutine handles are now ready: We iterate over the vector and resume each one.

This works relatively well, except that once you have a large number of waiting coroutines (say, because initialization is taking a really long time), the `push_back` on the vector might take a long time if the vector needs to be reallocated. The operation is still amortized $O(1)$, but the per-instance cost can be as high as $O(n)$.

Furthermore, the `push_back` can throw an exception due to low memory (note that `await_suspend` is not marked `noexcept`).

We'll address both of these issues next time.

[1] I always have to pause to think whenever I get to the `return` statements in the `await_ready` and `await_suspend` methods, because the return values have opposite sense. I have to remember that you want to "suspend if not ready".

Raymond Chen

Follow