Creating a co_await awaitable signal that can be awaited multiple times, part 5

devblogs.microsoft.com/oldnewthing/20210305-00

March 5, 2021

Raymond Chen

So far, we've been creating an awaitable signal that can be awaited multiple times.

In the most recent incarnation, we reversed the list in order to approximate FIFO resume. Doing this at resume time means that the list of nodes gets walked twice, once when reversing, and once when resuming. Let's fix this by keeping a separate pointer to the last pointer, so we can append nodes to the end of the list in $O(1)$. This adds an extra pointer to the size of the `awaitable_event`, but we will earn that memory back by overloading the `last` pointer to indicate whether the event is signaled: Once the event is signaled, we will never append any nodes, so we will set `last` to `nullptr` to indicate that the event is signaled.

The `last` member is a `std::atomic` because we access it outside the lock while another thread is potentially mutating it. The default accessors for `std::atomic` use `std::memory_order_seq_cst`, but we want `std::memory_order_relaxed`, since we don't really mind the data race, as long as the read isn't torn. To avoid having to write out `std::memory_order_relaxed` all the time, I'll introduce a `relaxed_atomic` helper class.

```
// new
template<typename T>
struct relaxed_atomic : std::atomic<T>
{
  using atomic = std::atomic<T>;
  using atomic::atomic;
  using atomic::load;
  using atomic::store;

  T load() const noexcept
  { return atomic::load(std::memory_order_relaxed); }
  void store(T value) noexcept
  { atomic::store(value, std::memory_order_relaxed); }

  operator T() const noexcept { return load(); }
  relaxed_atomic& operator=(T value) noexcept
  { store(value); return *this; }
};

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

  auto operator co_await() const noexcept
  { return awaiter{ *shared }; }

private:
  struct node
  {
    node* next;
    std::experimental::coroutine_handle<> handle;
  };

  struct state
  {
    // std::atomic_bool signaled =false;
    winrt::slim_mutex mutex;
    node* head = nullptr;
    relaxed_atomic<node**> last = &head; // new

    void set()
    {
      node* rest = nullptr;
      {
        auto guard = winrt::slim_lock_guard(mutex);
        last.store(nullptr, std::memory_order_relaxed); // new
        rest = std::exchange(head, nullptr);
      }
      // while (lifo) {
      //     auto n = lifo;
      //     lifo = std::exchange(n->next, fifo);
      //     fifo = n;
      // }
      while (rest) {
        auto handle = rest->handle;
        rest = rest->next;
        handle();
      }
    }

    bool await_ready() const noexcept
    {
      return !last.load(); // new
    }

    bool await_suspend(node& n) noexcept
    {
      auto guard = winrt::slim_lock_guard(mutex);
      auto p = last.load();
      if (!p) return false;
      *p = &n;
      last = &n.next;
      n.next = nullptr;
      return true;
    }

    void await_resume() const noexcept { }
  };

  struct awaiter
  {
    state& s;
    node n;

    bool await_ready() const noexcept { return s.await_ready(); }
    bool await_suspend(
      std::experimental::coroutine_handle<> handle) noexcept
    {
      n.handle = handle;
      return s.await_suspend(n);
    }

    void await_resume() const noexcept { return s.await_resume(); }
  };

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

There’s still a lot to say about this implementation (and the other implementations we’ve been looking at so far). We’ll take up some of the topics next time.

Raymond Chen

Follow