Creating a co_await awaitable signal that can be awaited multiple times, part 4

devblogs.microsoft.com/oldnewthing/20210304-00

March 4, 2021

Raymond Chen

Last time, we created an awaitable signal that can be awaited multiple times, operating entirely in user mode. However, it did perform memory allocations, and that could result in low-memory exceptions. Furthermore, it used a `std::vector`, and pushing a value onto the vector could take a long time if the vector needs to be reallocated.

We can avoid the variable cost of adding a single element to the collection by using a linked list. And we can avoid the memory allocation cost by having the awaiter provide the memory!

```
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
  {
    /* to be filled in */
  };

  struct awaiter
  {
    state& s;
    node n;

    /* to be filled in */
  };

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

An `awaitable_event` is a wrapper around a `shared_ptr` to a `state` structure, which is where all the real work happens.

The idea is that we keep a linked list of the coroutine handles that are waiting for the event to become signaled. The trick is that the nodes for the linked list come from the `awaiter`!

Recall that `co_await` begins by obtaining an awaiter, and then it calls methods on the awaiter to wait for the result, and then when the coroutine resumes, the awaiter is destructed. This means that we can have the awaiter itself provide the memory for the linked list nodes, since it exists for the entire time the coroutine is suspended. The awaiter is a local variable in the coroutine, so the compiler will allocate it in the coroutine frame. The memory for it is therefore effectively preallocated, so we don't have to worry about running out of memory when it comes time to queue the node on to the list. Furthermore, adding an item to a list is an $O(1)$ operation, ensuring that our mutex is never held for very long.

Now that we have a sketch for the design, we dig into the implementation.

```
struct state
{
  std::atomic<bool> signaled = false;
  winrt::slim_mutex mutex;
  node* head = nullptr;
```

As before, the `state` remembers whether the event is signaled (its primary purpose for existence). The mutex ensures that the signaling of the event and capturing the list of waiting coroutine handles is performed atomically.

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

To set the event, we mark the event as signaled and detach the list. To ensure safe behavior if the event is set more than once, we reset the `last` and `head` members. (Alternatively, we could early-out if the event is already signaled, but I feel better removing the dangling pointers.)

Once we've detached the head, we reverse the list so that the nodes are in FIFO order rather than LIFO (to approximate fairness), and then we resume each of the coroutine handles. It's important to capture the contents of the node before invoking the handle because resuming the coroutine will destroy the node.

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

We can short-circuit suspension if the event is already signaled.

```
    bool await_suspend(node& n) noexcept
    {
      auto guard = winrt::slim_lock_guard(mutex);
      if (signaled.load(std::memory_order_relaxed)) return false;
      n.next = head;
      head = &n;
      return true;
    }
```

To wait on the `awaitable_event`, we take the lock and recheck the event state. If it has been signaled while we were waiting for the lock, then we report that there is no need to suspend. Otherwise, we queue the provided node onto the linked list and request to be suspended.

Observe that `await_suspend` is now noexcept since there is no memory allocation. Furthermore, the operation runs in $O(1)$ since it's just manipulating some pointers.

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

There is nothing to report on resume, `await_resume` returns nothing.

The last missing piece is the awaiter. The awaiter's job is to provide the `node` to the `state`.

```
  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(); }
  };
```

The methods just forward to the `state`. The only wrinkle is that `await_suspend` hands over a node preinitialized to hold the coroutine handle, so it is ready to be added to the linked list.

One observation is that we walk the linked list twice at resumption. We walk it one time to reverse the list, and another time to resume the coroutines. We'll try to fix that next time.

Raymond Chen

Follow