Creating a co_await awaitable signal that can be awaited multiple times, part 2

devblogs.microsoft.com/oldnewthing/20210302-00

March 2, 2021

Raymond Chen

Last time, we created an awaitable signal that can be awaited multiple times. We noted that one problem with the implementation is that the object couldn't be copied: Everybody has to await the same object, which can create lifetime issues.

One way of solving this is to put the `awaitable_event` inside a `shared_ptr` and having everybody share the lifetime. But maybe we want the object itself to be copyable, so people don't have to do all that `shared_ptr` management.

We'll make the object copyable by duplicating the handle into each copy. All of the duplicate handles refer to the same event, so when the event is set, all the clients who are awaiting the handle will wake up.

```
struct awaitable_event
{
  awaitable_event(awaitable_event const& other)
  {
    auto current_process = GetCurrentProcess();
    winrt::check_bool(DuplicateHandle(current_process,
      other.os_handle(), current_process, handle.put(),
      0, false, DUPLICATE_SAME_ACCESS));
  }
  awaitable_event(awaitable_event&&) = default;

  void set() const noexcept
  { SetEvent(os_handle()); }

  auto operator co_await() const noexcept
  { return winrt::resume_on_signal(os_handle()); }

private:
  HANDLE os_handle() const noexcept
  { return handle.get(); }

  winrt::handle handle{
    winrt::check_pointer(CreateEvent(nullptr,
      /* manual reset */ true, /* initial state */ false,
      nullptr)) };
};
```

Now you can pass `awaitable_event` objects around, and people can copy it if they want to await it.

Another option is to make the `awaitable_event` wrap a `shared_ptr`. In that case, copying and destroying the object becomes much cheaper, since no kernel calls are made.

```
struct awaitable_event
{
  void set() const noexcept
  { SetEvent(os_handle()); }

  auto operator co_await() const noexcept
  { return winrt::resume_on_signal(os_handle()); }

private:
  HANDLE os_handle() const noexcept
  { return shared->get(); }

  std::shared_ptr<winrt::handle> shared =
    std::make_shared<winrt::handle>(
      winrt::check_pointer(CreateEvent(nullptr,
        /* manual reset */ true, /* initial state */ false,
        nullptr)));
};
```

One downside of these designs is that the awaiters consume kernel resources since they are awaiting a kernel event, and awaiting an already-signaled event still takes a kernel call in order to see whether the kernel event has been set. One use case for an awaitable signal is having every call to a function first wait for some one-time initialization to complete. This means that you have potentially a lot of awaiters, but once initialization is complete, you also have a lot of calls where the event has already been signaled. Maybe we can do the work entirely in user mode.

We'll do that next time.

Raymond Chen

Follow