C++ coroutines: Framework interop

devblogs.microsoft.com/oldnewthing/20191211-00

December 11, 2019

Raymond Chen

So far, we've been looking at the basics of awaitable objects. Even though we barely know anything beyond `await_suspend`, we already know enough to allow us to start diving deeper.

It is frequently the case that you need your awaiter to interact with something outside the C++ standard library. To make it easier to integrate coroutines with existing frameworks, the `coroutine_handle` can be converted to a `void*` by calling its `address()` method, and the resulting `void*` can be converted back to an equivalent `coroutine_handle` by calling `from_address()`.[1]

Most frameworks let you pass a pointer-sized piece of data around to help remember state, and being able to convert a handle into a pointer (and back) lets you pass the coroutine handle through such state parameters. Otherwise, you'd have to copy the `coroutine_handle` to the heap and pass the address of the heap block, and then keep track of when to free the heap block.

Let's demonstrate this by reimplementing `resume_new_thread` in terms of Win32 functions instead of the `std::thread` standard library class.

```
struct resume_new_thread : std::experimental::suspend_always
{
  void await_suspend(
      std::experimental::coroutine_handle<> handle)
  {
    HANDLE thread = CreateThread(nullptr, 0, callback,
                        handle.address(), 0, &threadId);
    if (!thread) throw some_kind_of_error();
    CloseHandle(thread);
  }

  DWORD CALLBACK callback(void* parameter)
  {
    auto handle = std::experimental::coroutine_handle<>::
                    from_address(parameter);
    handle();
    return 0;
  }
};
```

The basic idea is the same as last time: When the coroutine suspends, schedule the continuation on a newly-created thread.

The `CreateThread` function allows you to pass a single pointer-sized piece of data, so we convert our handle to a `void*` by calling the `address` method, and pass that pointer as the reference data to the thread procedure. The thread procedure converts the pointer back into a coroutine handle by calling `from_address`, and then invokes the coroutine to resume execution.

If terseness is your game, you could inline the thread procedure as a stateless lambda, taking advantage of the implicit conversion from a stateless lambda to a function pointer.

```
struct resume_new_thread : std::experimental::suspend_always
{
  void await_suspend(
      std::experimental::coroutine_handle<> handle)
  {
    HANDLE thread = CreateThread(nullptr, 0,
      [](void* parameter) -> DWORD
      {
        std::experimental::coroutine_handle<>::
                    from_address(parameter)();
        return 0;
      }, handle.address(), 0, &threadId);
    if (!thread) throw some_kind_of_error();
    CloseHandle(thread);
  }
};
```

Next time, we'll use what we've learned about awaiters to develop a way to override C++/WinRT coroutine threading defaults.

[1] The method names `address` and `from_ address` give a strong clue as to what the `void*` represents: it's the address of runtime-managed coroutine state, known in the language specification as a *coroutine frame*.

Raymond Chen

Follow