# Using fibers to expand a thread’s stack at runtime, part 4

devblogs.microsoft.com/oldnewthing/20200605-00

June 5, 2020

Raymond Chen

Last time, we transported some ephemeral error state from the temporary fiber to the originating thread. A common way of reporting an error in C++ is to use an exception, so let’s use that.

```
[[noreturn]] void ThrowWin32Error(DWORD error);

template<typename RetType>
RetType RunOnFiberWorker(
    RetType (*callback)(void*),
    void* parameter)
    // RetType failureValue
{
  // static_assert(std::is_trivially_copyable_v<RetType>);
  // static_assert(std::is_trivially_destructible_v<RetType>);

  struct State
  {
    RetType (*callback)(void*);
    void* parameter;
    std::variant<std::exception_ptr, RetType>
      capturedValue;
    HANDLE originalFiber;

    void FiberProc()
    {
      try {
        capturedValue.template emplace<1>
          (callback(parameter));
      } catch (...) {
        capturedValue.template emplace<0>
          (std::current_exception());
      }
      SwitchToFiber(originalFiber);
    }

  } state{ callback, parameter };
  // std::move(failureValue), errno

  DWORD error = RunOnFiberTypeNeutral(
    [](void* parameter)
    {
      reinterpret_cast<State*>(parameter)->FiberProc();
    }, &state, &state.originalFiber);
  if (error != ERROR_SUCCESS) {
    ThrowWin32Error(error);
  }

  auto e = std::get_if<0>(&state.capturedValue);
  if (e) std::rethrow_exception(*e);
  return std::get<1>(std::move(state.capturedValue));
}

template<typename Lambda>
auto RunOnFiber(Lambda&& lambda)
{
  using Type = std::remove_reference_t<Lambda>;
  using RetType = decltype(lambda());
  return RunOnFiberWorker<RetType>(
    [](void* parameter)
    {
      return (*reinterpret_cast<Type*>(parameter))();
    }, &lambda);
}
```

In this case, we capture the result of the callback with a `std::variant` of an `exception_ptr` or the formal return value. The `exception_ptr` is used if the callback threw an exception.

The `exception_ptr` is the first type in the variant because the `RetType` may not be default-constructible, and even if it has a default constructor, that default constructor may be heavy with unwanted side-effects. Putting the `exception_ptr` as the first type in the variant means that the default constructor for the variant creates an `exception_ptr`, which is default-constructible.

We then ask `RunOnFiberTypeNeutral` to do the fiber magic. If it failed, then we use some program-specific `ThrowWin32Error()` function to transform the Win32 error into some kind of an exception.

Otherwise, we know that the fiber ran to completion. In the fiber procedure, we call the callback and we save the result in the `capturedValue` variable as the `RetType`. If the callback threw an exception, we catch the exception and stow it in the `capturedValue` as an `exception_ptr`.

After returning to the original thread, we keep inside the variant to see whether it holds an exception or a value. If it holds an exception, we rethrow it. Otherwise, we return the value.

We must use the explicit index versions of `variant::` `emplace`, `get`, and `get_if` rather than the more readable type-based versions because the type-based version won't work if the `RetType` is `exception_ptr`!

Observe that the `RetType` is always moved. There is no requirement that it be default-constructible or copyable.

Note that the above code does not work if the lambda returns a reference. I'll leave that as an exercise.

Even if you don't plan on working with fibers, this series showed how to transport state between threads, which is still useful.

But wait, our discussion of using fibers to expand a stack dynamically isn't over. We'll pick up additional topics next time.

Raymond Chen

**Follow**