# Using fibers to expand a thread's stack at runtime, part 3

devblogs.microsoft.com/oldnewthing/20200604-00

June 4, 2020

Raymond Chen

We've been working on a `RunOnFiber` function that creates a fiber with a large stack and runs the lambda on it. This is handy if you have a function that requires a lot of stack, and you're not sure whether your caller provided enough.

The version we developed assumed that the only thing of interest that is returned from the lambda is its return value. But you might have additional context that needs to be returned, such as the Win32 last-error code or the C runtime `errno`.

We can adapt our existing version to capture additional state about the fiber so it can be transferred to the original thread.

```
DWORD RunOnFiberTypeNeutral(
    LPFIBER_START_ROUTINE fiberProc,
    void* parameter,
    HANDLE* originalFiber)
{
  unique_fiber workFiber{ CreateFiberEx(0, EXTRA_STACK_SIZE, 0,
      fiberProc, parameter) };

  if (!workFiber) return GetLastError();

  unique_thread_as_fiber threadFiber;
  if (!IsThreadAFiber()) {
    threadFiber.reset(ConvertThreadToFiber(nullptr));
    if (!threadFiber) {
      return GetLastError();
    }
  }

  *originalFiber = GetCurrentFiber();

  SwitchToFiber(workFiber.get());
  return ERROR_SUCCESS;
}

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;
    RetType capturedValue;
    int capturedErrno{};
    DWORD capturedError{};
    HANDLE originalFiber;

    void FiberProc()
    {
      capturedValue = callback(parameter);
      capturedErrno = errno;
      capturedError = GetLastError();
      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) {
    state.capturedError = error;
  }

  SetLastError(state.capturedError);
  errno = state.capturedErrno;
  return std::move(state.capturedValue);
}
```

Since this template may be used with different `RetType`s, we factor out the part that is type-independent into a `RunOnFiberTypeNeutral` helper. There is a bit of trickiness in the helper function: We make sure to capture the error code immediately and return it explicitly. This is important because the destructors for the `unique_fiber` and `unique_thread_as_fiber` may change the value of `GetLastError()`, so we need to grab it while we still can.

We expand the state shared with the fiber to include not only the return value of the callback, but also the `GetLastError()` and `errno` values as they were at the completion of the callback.

When all is said and done, we restore the error code and `errno` to the current thread and then return the value that was captured from the execution of the callback.

If `RunOnFiberTypeNeutral` fails, then we take the error code and put it into the `capturedError` so that the normal cleanup code will apply it to the thread before returning.

I'm assuming that if anything goes wrong with `CreateFiber` or `ConvertThreadToFiber`, then the `failureValue` and existing Win32 error code are sufficient to explain what went wrong. I leave setting the `errno` as an exercise.

Note that we require that the return value type be trivially copyable and trivially destructible. Trivial copyability is required so that we can return it without disturbing the Win32 last-error code or `errno`. Trivial destructibility is required so that we can destruct the `failureValue` parameter and the `capturedValue` without disturbing the Win32 last-error code or `errno`.

This requirement is not a problem in practice, because a return type that requires nontrivial copy or destructor operations is going to trigger invisible code execution for temporary objects, which is likely to affect ephemeral thread-local state. In that case, you cannot rely on

the Win32 last-error code or the `errno` value anyway, so there's no reason to try to preserve that value from the fiber back to the original thread.

We do not, however, require that the return value type be trivially constructible. Instead, we accept a `failureValue` and use that if something goes wrong with setting up the fiber. We are careful to `std::move` the value around, even though that doesn't really mean anything for trivially-copyable types. But it'll come in handy later. Typically, the return type in these cases is a simple scalar, like an integer.

Here's the new `RunOnFiber` function that accepts a `failureValue`:

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

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

Next time, we'll look at another error-reporting mechanism: C++ exceptions.

Raymond Chen

**Follow**