Using fibers to expand a thread's stack at runtime, part 2

devblogs.microsoft.com/oldnewthing/20200603-00

June 3, 2020

Raymond Chen

Last time, we wrote a `RunOnFiber` function that accepted a lambda and ran the lambda on a fiber.

Since the `RunOnFiber` function is templated, a new copy of the function is created for each lambda. But we can reduce the code size explosion by factoring out the part that is independent of the lambda.

We use a technique similar to the one we used when we wrote our own simplified version of `std::function`: Convert the lambda to a flat callback and a `void*`.

```
HRESULT RunOnFiberWorker(
    HRESULT (*callback)(void*),
    void* parameter)
{
  struct State
  {
    HRESULT (*callback)(void*);
    void* parameter;
    HANDLE originalFiber;
    HRESULT result = S_OK;

    void FiberProc()
    {
      result = callback(parameter);
      SwitchToFiber(originalFiber);
    }

  } state{ callback, parameter };

  unique_fiber workFiber{ CreateFiberEx(0, EXTRA_STACK_SIZE, 0,
      [](void* parameter)
      {
        reinterpret_cast<State*>(parameter)->FiberProc();
      }, &state) };

  if (!workFiber) return HRESULT_FROM_WIN32(GetLastError());

  unique_thread_as_fiber threadFiber;
  if (!IsThreadAFiber()) {
    threadFiber.reset(ConvertThreadToFiber(nullptr));
    if (!threadFiber) {
      return HRESULT_FROM_WIN32(GetLastError());
    }
  }

  state.originalFiber = GetCurrentFiber();
  SwitchToFiber(workFiber.get());

  return state.result;
}

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

The boilerplate is now in a helper function called `RunOnFiberWorker`, and the template function type-erases the lambda into a `void*` and callback function. The callback function converts the `void*` back into the lambda and invokes it.

The decomposition of the lambda into a callback and `void*` allows the same `RunOnFiber-Worker` to be used for all lambdas. The lambda-specific code is just in the production of the callback function.

This code is not quite finished yet, because there's the case where the lambda is a functor passed by const reference, in which case we need to respect const-ness and invoke the const version of the `operator()` overload.

There also the oddball case where the functor has an overloaded `operator&`. We can avoid that by using `std::addressof`.

A little bit of additional fiddling will take care of that:

```
template<typename Lambda>
HRESULT RunOnFiber(Lambda&& lambda)
{
  using Type = std::remove_reference_t<Lambda>;
  using Decayed = std::remove_cv_t<Type>;
  return RunOnFiberWorker([](void* parameter)
    {
      return (*reinterpret_cast<Type*>(parameter))();
    }, const_cast<Decayed*>(std::addressof(lambda)));
}
```

Next time, we'll look at writing a `RunOnFiber` function that reports errors by means other than just the return value.

Raymond Chen

Follow