Using fibers to expand a thread's stack at runtime, part 1

devblogs.microsoft.com/oldnewthing/20200602-00

June 2, 2020

Raymond Chen

Last time, we considered a library that required a lot of stack space, and critiqued one proposed solution that involved pre-emptively converting every thread to a fiber.

The solution is simply not doing anything at all in the `DLL_PROCESS_ATTACH` notification. Instead, do the work when the application calls into the library.

For concreteness, let's say that the library reports errors with `HRESULT` values.

```
HRESULT TransformWidget(Widget& widget, Options options)
{
 return RunOnFiber([&]() -> HRESULT {
  return TransformWidgetWorker(widget, options);
 });
}
```

The `RunOnFiber` function's job is to create a temporary fiber and run the lambda on it. The lambda's job is to do whatever it is the library needs to do. In this example, you would fill in the lambda with whatever you need to do in order to transform the widget. I just used a placeholder function.

Of course, the interesting part for the purpose of today's discussion is the `RunOnFiber` function.

We start with two custom deleters.

```
struct fiber_deleter
{
 using pointer = HANDLE;
 void operator()(HANDLE h) { DeleteFiber(h); }
};
using unique_fiber =
    std::unique_ptr<HANDLE, fiber_deleter>;

struct thread_as_fiber_deleter
{
 using pointer = HANDLE;
 void operator()(HANDLE) { ConvertFiberToThread(); }
};
using unique_thread_as_fiber =
    std::unique_ptr<HANDLE, thread_as_fiber_deleter>;
```

The `fiber_deleter` let us create a `unique_fiber` RAII type which destroys the fiber.

The `thread_as_fiber_deleter` lets us create a `unique_thread_as_fiber` RAII type which undoes the `ConvertThreadToFiber` if it succeeded. (As a bonus check, we could assert that the provided handle is equal to `GetCurrentFiber()`.)

The idea is to convert the thread to a fiber, do our work, and then convert the fiber back to a thread before returning to the application. We leave the thread in the same state we found it. That way, the application never observes a thread that has been converted to a fiber behind its back.

```
template<typename Lambda>
HRESULT RunOnFiber(Lambda&& lambda)
{
  struct State
  {
    Lambda& lambda;
    HANDLE originalFiber;
    HRESULT result = S_OK;

    void FiberProc()
    {
      result = lambda();
      SwitchToFiber(originalFiber);
    }

  } state{ lambda };

  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;
}
```

Okay, let's walk through this.

We start by capturing all of the information we need to share with the fiber into a `State` object. We start by giving the fiber access to the lambda that it needs to run.

Next, we create a fiber to use for our large stack-consuming operation. We assign a specific stack size for this fiber because the entire reason for the fiber is that we need to be running a a stack of known minimum size.

The fiber's procedure extracts the lambda from the state and executes it. It saves the result of the lambda into the `result` member, and then switches back to the original fiber.

What's the original fiber? We'll get there.

Next, we convert the thread to a fiber if it isn't one already. We use a `unique_thread_as_fiber` to remember whether this succeeded, so we know whether to convert the fiber back to a thread before we return.

After we are sure the thread is a fiber (either because it already was a fiber, or because we converted it to one), we get the fiber and save into the `originalFiber` so that our worker fiber can switch back.

And then the fun begins: We switch to our worker fiber. That fiber then runs the lambda, and then switches back to the original fiber. Switching back to the original fiber returns control back to the point immediately after the `SwitchToFiber` call in the `RunOnFiber` function.

Now that we have run the lambda on the fiber, we can return the result that we captured in the fiber.

Thanks to RAII, the destructors for the `threadFiber` and `workFiber` do the work of restoring the thread to its original state and destroying our temporary fiber.

That's the basic idea. Creating a fiber only as needed means that we don't waste memory by filling it with fibers that never end up being used, or fibers which are sitting around idle waiting for something to do. Converting the thread to a fiber only for the lifetime of the library function, and then converting it back, means that we do not interfere with any other code that wanted to control the fiber state of the thread.

Now that we have the basic idea, we can start refining and extending it. We'll take a simple first step next time.

Raymond Chen

Follow