# Using fibers to expand a thread's stack at runtime, part 6

devblogs.microsoft.com/oldnewthing/20200612-00

June 12, 2020

Raymond Chen

Last time, we tried to create fibers on demand to expand the stack, effectively creating a segmented stack. We noted that each time you cross a stack expansion boundary, a fiber is created (if calling) or destroyed (if returning). This can result in situations where you end up spending more time creating and destroying fibers than you do performing meaningful work.

We can avoid this problem by keeping a one-level fiber cache.

This does mean that we need to set up our fiber to be suitable for multiple use. Up until now, our fibers did their one thing and then stopped. We need to modify them so they can be asked to do a new thing after finishing their previous thing.

```
struct State
{
  HRESULT (*callback)(void*);
  void* parameter;
  HRESULT result;
  HANDLE originalFiber;
  unique_fiber workFiber;

  static void CALLBACK s_FiberProc(void* parameter)
  {
    reinterpret_cast<State*>(parameter)->FiberProc();
  }

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

The state we maintain for the fiber is basically the same as what we had before. We add the `unique_fiber` to the state so that destructing the state also destroys the fiber. The interesting part is the new `FiberProc` : After switching back to the original fiber, the fiber

procedure loops back and does everything all over again.

This seems weird until you realize that `SwitchToFiber` doesn't return until work fiber is switched to. And before we do that, we give the work fiber a different callback to run. This is what allows the same fiber to be reused: When we want to use the fiber, we set up the callback, set ourselves as the `originalFiber`, and then switch to the fiber and wait for it to switch back.

```
struct Cache
{
  std::unique_ptr<State> state;
};
```

To avoid destroying a fiber, only to create another one immediately, we keep a one-element cache of fiber states.

```
HRESULT RunOnFiberWorker(
    Cache& cache,
    HRESULT (*callback)(void*),
    void* parameter)
{
  // Step 1
  unique_thread_as_fiber threadFiber;
  if (!IsThreadAFiber()) {
    threadFiber.reset(ConvertThreadToFiber(nullptr));
    if (!threadFiber) {
      return HRESULT_FROM_WIN32(GetLastError());
    }
  }

  // Step 2
  auto state = std::move(cache.state);
  if (!state) {
    state = std::unique_ptr<State>(new(std::nothrow) State());
    if (!state) return E_OUTOFMEMORY;
    state->workFiber = unique_fiber{ CreateFiberEx(0,
        EXTRA_STACK_SIZE, 0, State::s_FiberProc, state.get()) };
    if (!state->workFiber) {
      return HRESULT_FROM_WIN32(GetLastError());
    }
  }

  // Step 3
  state->callback = callback;
  state->parameter = parameter;
  state->originalFiber = GetCurrentFiber();

  // Step 4
  SwitchToFiber(state->workFiber.get());

  // Step 5
  cache.state = std::move(state);
  return cache.state->result;
}
```

There are a few things going on here, but they break down nicely into steps.

The first step is to convert the thread to a fiber if it isn't one already. The `unique_thread_as_fiber` will convert the fiber back to a thread if necessary.

The second step is to get a fiber. We first try to steal one from the cache, but if the cache is empty, then we create a new `State` and attach a fiber to it.

The code to create the `State` is a bit annoying because we are being careful to capture the precise failure reason. If we didn't want to be quite so precise, we could have just treated them all as `E_FAIL` or something similarly generic.

Once we have a `State`, we tell the fiber what we want it to do next by giving it a `callback`, `parameter`, and `originalFiber`.

And then we use `SwitchToFiber` to tell the work fiber to run the callback, save the result, and switch back to the `originalFiber`.

When the switch back occurs, we put our state into the cache (possibly pushing out an existing cache entry) and return the result of the callback.

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

template<typename Lambda>
HRESULT RunOnFiberIfNeeded(
    size_t minimumStack,
    Cache& cache,
    Lambda&& lambda)
{
{
  if (is_stack_available(minimumStack)) {
    return lambda();
  } else {
    return RunOnFiber(cache, std::forward<Lambda>(lambda));
  }
}
}
```

The `RunOnFiber` function is basically the same, just with the addition of a `cache` parameter that is forwarded to the worker.

Now we can take it for a spin.

```
HRESULT FrobAllNodes(Tree& tree)
{
  Cache cache;
  return FrobAllNodesWorker(cache, tree);
}

HRESULT FrobAllNodesWorker(
    Cache& cache, Tree&amp;amp, tree)
{
  return RunOnFiberIfNeeded(20480, cache, [&]()
  {
    FrobTheNode(tree.Node());
    for (auto&& child : tree.Children()) {
      FrobAllNodesWorker(cache, child);
    }
    return S_OK;
  });
}
```

We start by creating our cache of a single fiber. The cache starts out empty.

We then enter the recursive function, which forwards the cache into `RunOnFiberIfNeeded`. If no fiber is needed, then the lambda simply runs immediately.

At some point during the recursion, we may end up needing to create a fiber. The first time this happens, we create it from scratch, use it to run the lambda, and then put the fiber into the cache.

The subsequent times we need to create a fiber, we can reuse the one that's already in the cache.

When the recursion completes, we return to `FrobAllNodes`, which destructs the cache and any fiber still lingering in it.

Let's say you are really unlucky and your recursion is so deep that you need *two* fibers. The first fiber goes as expected. The second fiber tries to get a fiber from the cache but cannot, so a second fiber is created. When we are finished with the second fiber, it goes into the cache, so that it can be used further.

Eventually, we unwind all the way out of the first fiber. In this case, putting that first fiber into the cache causes the second fiber (sitting in the cache) to be destroyed. This means that if we then recurse deep enough to require a second fiber, we will have to create it from scratch since we don't have one cached.

My sense is that optimizing the two-fiber scenario is not important because you are unlikely to be bouncing rapidly from zero to two fibers, and then all the way back down to zero again. Even if you did have a scenario that bounced repeatedly from zero to two and back, that

bounce probably won't be rapid because it will likely take a long time to consume the entire stack provided to the first fiber.

That's all for now on the subject of using fibers to expand a thread's stack at runtime. There's more to be said, but I'm going to have to cover other prerequisite topics before returning to this one.

**Bonus chatter**: If you don't want to deal with the hassle of passing the `cache` around, you can put the cache in thread-local storage. Note that you must put it in *thread*-local storage, not *fiber*-local storage: Putting it in fiber-local storage would make it inaccessible each time we change fibers.

**Bonus bonus chatter**: I noted some time ago that fibers aren't useful for much any more. And this series shows that it's still true: We are using fibers not for scheduling but for their side effect of giving allowing us to expand the stack. Pretty much nobody uses fibers for scheduling, which was their original purpose.

Raymond Chen

**Follow**