Using fibers to expand a thread’s stack at runtime, part 5

devblogs.microsoft.com/oldnewthing/20200611-00

June 11, 2020

Raymond Chen

Now that we figured out how to probe whether there is sufficient remaining stack, we can use that to avoid all the work of creating and managing a fiber if we don’t need it.

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

If there is sufficient stack available, then we just run the lambda immediately. Otherwise, we ask `RunOnFiber` to create a fiber and run the lambda on that temporary fiber. When the lambda returns, we destroy the temporary fiber.

We basically reinvented the segmented stack.

One place you can use this technique is if your function is highly recursive. In that case, you may not know how much stack is needed for the entire operation, but you do know how much stack you need for each step of the recursion. You can kick off the next level of the recursion with `RunOnFiberIfNeeded` , and the function will consume the calling thread’s stack until it runs low, and then overflow onto a fiber.

Here’s the original function:

```
void FrobAllNodes(Tree& tree)
{
  FrobTheNode(tree.Node());
  for (auto&& child : tree.Children()) {
    FrobAllNodes(child);
  }
}
```

And here’s a revised version:

```
void FrobAllNodes(Tree& tree)
{
  RunOnFiberIfNeeded(20480, [&]()
  {
    FrobTheNode(tree.Node());
    for (auto&& child : tree.Children()) {
      FrobAllNodes(child);
    }
  });
}
```

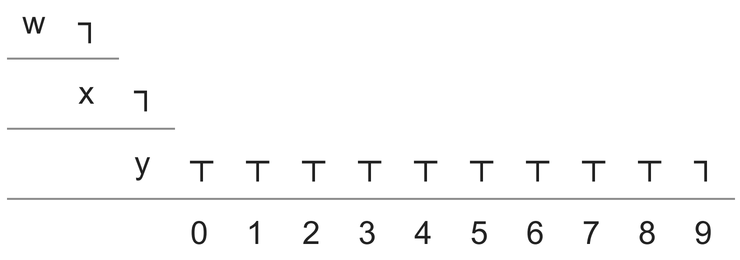
This code checks whether the stack is running low, and if so switches to a fiber.

Problem solved?

Not quite.

What can happen is that when a thread’s stack gets too low, the code creates a fiber for the next level of the recursion. That level returns, and the temporary fiber is destroyed. And then the code moves to the next child and starts a new level of recursion. That new level also creates a fiber, and the process repeats. Each time we cross the boundary, we either create a fiber (when recursing) or destroy one (when returning).

If the recursion crosses the boundary a lot, then you end up spending a lot of time managing fibers and not a lot of time doing work. For example, consider a tree that is four levels deep, with a single child at the first three levels, and ten children at level four.



If the original stack had enough space to accommodate three levels of the recursion, then what’ll happen is that we’ll quickly get through the first three levels, and then each of the ten leaf nodes will create and destroy its own fiber:

- Process w.
- Recurse for x.
- Process x.
- Recurse for y.
- Process y.

- Recurse for 0 — create a fiber.
- Process 0.
- Return from 0 — destroy a fiber.
- Recurse for 1 — create a fiber.
- Process 1.
- Return from 1 — destroy a fiber.
- Recurse for 2 — create a fiber.
- Process 2.
- Return from 2 — destroy a fiber.
- ⋮
- Recurse for 9 — create a fiber.
- Process 9.
- Return from 9 — destroy a fiber.
- Return from z.
- Return from y.
- Return from x.

This is known as the "hot-split" problem: If the stack is almost depleted, then the next call will trigger the creation of a new stack segment (which for us is a fiber). When that call returns, the segment is freed. If the same call is repeated in a loop, you end up incurring significant overhead creating and destroying stack segments.

We'll look at addressing this problem next time.

Raymond Chen

Follow