好的，我们来详细深入地探讨一下 Windows 纤程（Fiber）的内部结构。

首先，要理解纤程，必须将其与线程进行对比，因为纤程是构建在线程之上的。

核心概念：线程 vs. 纤程

- **线程 (Thread)**：是操作系统内核调度的基本单位。内核负责管理线程的上下文（寄存器状态、栈等），并决定哪个线程在哪个CPU核心上执行。线程的调度是**抢占式**的，内核根据优先级和时间片强行切换线程。
- **纤程 (Fiber)**：是用户模式下的调度单元。**内核不知道纤程的存在**，它只知道承载纤程的线程。纤程的调度完全由应用程序自己控制（协作式调度）。一个线程可以包含多个纤程，但在任一时刻，只有一个纤程在该线程上运行。

你可以把一个线程想象成一个“**执行车道**”，而纤程则是这个车道上跑的“**车**”。操作系统（交通管理员）只关心车道是否被占用，至于车道上具体跑的是哪辆车，它不关心。换车的工作由司机（应用程序）自己手动完成。

纤程的内部数据结构

当一个线程转换为纤程（通过 `ConvertThreadToFiber`）或创建一个新的纤程（通过 `CreateFiber` 或 `CreateFiberEx`）时，系统会在用户模式的内存中为其分配和初始化一个数据结构。这个结构对内核是不可见的，完全由用户模式的 DLL（主要是 `kernel32.dll` / `kernelbase.dll`）管理。

虽然微软没有完全公开这个结构的详细定义，但通过调试、文档和逆向工程，我们知道它主要包含以下关键信息：

1. FIBER 结构体

这是每个纤程的控制块，类似于线程的 TEB（Thread Environment Block）。它包含了维护纤程执行状态所需的所有信息。

字段类型	名称 (示例)	描述
执行上下文	`FiberContext`	这是一个 `CONTEXT` 结构体，这是最核心的部分。它保存了纤程被挂起时的**CPU寄存器状态**，包括指令指针 (EIP/RIP)、栈指针 (ESP/RSP)、基址指针 (EBP/RBP)、以及通用寄存器 (EAX, EBX等)。当切换回这个纤程时，就是用这个结构体来恢复执行环境。
栈信息	`StackBase`, `StackAllocationBase`, `StackLimit`	指向纤程的**栈内存空间**。纤程可以有自己的栈（默认情况下为1MB），也可以通过 `CreateFiberEx` 指定自定义的栈大小和地址。这些指针定义了栈的顶部、底部和分配边界。
异常处理	`ExceptionList`	指向纤程的**结构化异常处理 (SEH)** 链的头部。这意味着每个纤程有自己独立的异常处理机制。
纤程数据	`FiberData`	一个 `PVOID` （void指针），这是在创建纤程（`CreateFiber`）时传入的自定义数据。应用程序可以用它来存储与纤程相关的状态信息。
版本信息	`FiberVersion`	标识纤程结构的版本。
线程/纤程关联	`ThreadId`	创建此纤程的线程的ID。纤程不能在线程间迁移，它始终绑定于创建它的线程。
TEB指针	`Teb`	指向承载此纤程的线程的**线程环境块 (TEB)**。这建立了纤程与其宿主线程的连接。
调度信息	`Flags`, `UserData`	一些内部标志位，可能用于指示纤程的状态（是否可调度等）。

2. 纤程的栈 (Fiber Stack)

每个纤程通常都有自己的执行栈。这是通过 `VirtualAlloc` 在用户空间分配的内存区域。这确保了纤程之间的栈是独立的，不会相互干扰。当一个纤程运行时，线程的当前栈指针（`ESP/RSP`）会被设置为这个纤程的栈。

关键点：纤程栈的分配是**一次性**的（在创建时分配），并且默认**提交**（commit）物理内存，而不是保留（reserve）。这意味着即使你创建一个拥有1MB栈的纤程但只使用4KB，它也会立即占用1MB的物理内存（或页面文件空间）。这就是为什么需要谨慎使用 `CreateFiberEx` 来管理栈大小的原因。

3. 线程的当前纤程状态

线程本身也需要知道它当前正在执行哪个纤程。

- **TEB 中的指针**：线程的 TEB 中有一个名为 `NtTib.FiberData` 的字段（在 x86 上也可以通过 `FS:[0x10]` 访问），它是一个 `PVOID`。**当一个线程转换为纤程后，这个指针指向当前正在运行的纤程的 FIBER 结构体。**
- **`GetCurrentFiber` 宏**：这个宏的本质就是读取 `TEB->NtTib.FiberData` 的值。它返回的就是当前纤程的 FIBER 结构地址。
- **`GetFiberData` 宏**：这个宏读取 `TEB->NtTib.FiberData` 指向的 FIBER 结构，然后取出其中的 `FiberData` 字段（即创建时传入的用户自定义数据）。

纤程调度的工作原理

纤程调度是**协作式**的，完全在用户模式下进行。核心API是 `SwitchToFiber`。

1. **保存状态**：当调用 `SwitchToFiber(LPVOID lpFiber)` 时：
    - 系统首先获取**当前纤程**的 FIBER 结构（通过 `TEB->NtTib.FiberData`）。
    - 它将当前CPU的所有寄存器状态（除了少数 volatile 寄存器）保存到当前纤程的 FIBER 结构的 `CONTEXT` 成员中。
2. **加载状态**：
    - 系统然后从**目标纤程**（由 `lpFiber` 参数指向的 FIBER 结构）的 `CONTEXT` 成员中加载保存的寄存器状态。
    - 特别重要的是加载栈指针（`ESP/RSP`），这会将线程的执行栈切换到目标纤程的栈。
    - 同时也会加载指令指针（`EIP/RIP`），这决定了代码从哪里开始继续执行。
3. **更新当前指针**：
    - 线程 TEB 中的 `NtTib.FiberData` 被更新为指向目标纤程的 FIBER 结构。
    - `GetCurrentFiber()` 现在会返回新的纤程地址。
4. **执行恢复**：

    - CPU 开始从新的指令指针和栈指针处执行，即目标纤程上次调用 `SwitchToFiber` 的位置之后。

这个过程**不涉及内核模式切换**（除了可能的页面错误等异常），也没有复杂的线程调度算法，因此上下文切换的开销远低于线程切换。

总结与关键点

- **用户模式构造**：纤程是纯粹的用户模式概念，内核不知情。其所有状态（FIBER结构、栈）都存储在用户空间。
- **独立上下文**：每个纤程有自己的 `CONTEXT`（寄存器）、栈和异常处理链。
- **协作式调度**：由应用程序显式调用 `SwitchToFiber` 来切换，调度策略由程序员决定（轮询、优先级等）。
- **绑定线程**：纤程被牢牢绑定在创建它的线程上，绝不能切换到另一个线程上执行。
- **性能与风险**：
    - **优点**：切换速度快，避免了陷入内核的开销。
    - **缺点**：一个纤程如果不主动让出（`SwitchToFiber`），它会独占其宿主线程，导致该线程上的所有其他纤程饿死。此外，纤程栈的默认内存占用可能很高。
- **使用场景**：现代软件开发中纤程已不常用，但在一些特定场景下很有价值，例如：
    - 将遗留的协作式多任务代码（如一些游戏引擎、模拟器）移植到Windows。
    - 实现非常高效的用户级任务调度器，其中应用程序对执行模式有完全的控制权。

希望这个详细的内部结构解释能帮助你透彻地理解 Windows 纤程。