It's fine to use fibers, but everybody has to be on board with the plan

devblogs.microsoft.com/oldnewthing/20100226-00

February 26, 2010

Raymond Chen

We saw fibers a long time ago when I looked at how you can use fibers as a form of coroutines to simplify the writing of enumerators. A fiber is a handy tool, but it's a tool with very sharp edges. Since fibers are promiscuous with threads, you have to be careful when running code that cares about what thread it is running on, because that code may discover that its thread changed out from under it. For example, critical sections and mutexes remember which thread owns them. If you enter a critical section on a fiber, and then you unschedule the fiber, then reschedule it onto a different thread, and then you leave the critical section, your critical section will end up corrupted because you broke the rule that says that a critical section must be exited on the same thread that entered it. Actually, you were already in bad shape once you unscheduled the fiber while it owned a resource: An unscheduled fiber cannot release the resource. Unscheduling a fiber is like suspending a thread: Anybody who later waits for that fiber to do anything will be waiting for an awful long time, because the fiber isn't running at all. The difference, though, is that the fiber is unscheduled at controlled points in its execution, so you at least have a chance at suspending it at a safe time if you understand what the fiber is doing. For example, suppose you enter a critical section on a fiber, and then unschedule the fiber. Some time later, a thread (either running as a plain thread or a thread which is hosting a fiber) tries to enter the critical section. One of two things can happen:

1. The thread happens to be the same one that was hosting the fiber that entered the critical section. Since a thread is permitted to re-enter a critical section it had previously acquired, the attempt to enter the critical section succeeds. You now have two chunks of code both running inside the critical section, which is exactly what your critical section was supposed to prevent. Havoc ensues.
2. The thread happens to be different from the one that was hosting the fiber that entered the critical section. That thread therefore blocks waiting for the critical section to be released. But in order for that to happen, you have to reschedule the owning fiber on its original thread so it can exit its protected region of code and release the critical section.

More generally, if you use an object which has thread affinity on a fiber, you are pretty much committed to keeping that fiber on that thread until the affinity is broken. This affinity can be subtle, because most code was not written with fibers in mind. Any code which calls `TlsGetValue` has thread affinity, because thread local storage is a per-thread value, not a per-fiber value. (This also applies to moral equivalents to `TlsGetValue`, like code which calls `GetCurrentThreadId` and uses it as a lookup key in a table.) You need to use `FlsGetValue` to get values which follow fibers around. But on the other hand, if the code is not running on a fiber, then you can't call `FlsGetValue` since there is no fiber to retrieve the value from. This dichotomy means that it's very hard if not impossible to write code that is both thread-safe and fiber-aware if it needs to store data externally on a per-thread/fiber basis. Even if you manage to detect whether you are running on a thread or a fiber and call the appropriate function, if somebody calls `ConvertThreadToFiber` or `ConvertFiberToThread`, then the correct location for storing your data changed behind your back. If you are calling into code that you do not yourself control, then in the absence of documentation to the contrary, you don't really have enough information to know whether the function is safe to call on a fiber. For example, C runtime functions like `strcmp` have thread affinity (even though there's nothing obviously threadlike about comparing strings) because they rely on the current thread's locale. Bottom line: (similar to the bottom line from last time): You have to understand the code that runs on your fiber, or you may end up accidentally stabbing yourself in the eyeball.

Bonus chatter: Structured exception handling is fiber-safe since it is stack-based rather than thread-based. Note, however, that when you call `ConvertThreadToFiber`, any active structured exception handling frames on the thread become part of the fiber.

Raymond Chen

Follow