How can I expand my thread's stack at runtime?

devblogs.microsoft.com/oldnewthing/20200601-00

June 1, 2020

Raymond Chen

A customer had a library that required a lot of stack space, and they were concerned that the functions in the library may be called on a small stack. Since this is a library, they don't have control over the code that created the stack. Is there a way to expand a thread's stack at runtime?

The size of a thread's stack is fixed when it is created, but you can use fibers to switch to an alternate stack.

The customer's plan was to handle the `DLL_PROCESS_ATTACH` DLL notification by calling `ConvertThreadToFiber` to convert the newly-created thread to a fiber, making a note of whether the conversion failed with `ERROR_ALREADY_FIBER`. It also created a fiber for that thread to use if it called into the library.

When the application calls into the library, the library uses `SwitchToFiber` to switch to the fiber that was preallocated for the thread, does its work, and then switches back to the original calling fiber before returning to the app.

Finally, the DLL responds to the `DLL_PROCESS_DETACH` notification by dstroying the per-thread fiber and calling `ConvertThreadToFiber` if the original call to `ConvertThreadToFiber` succeeded.

The customer had the right idea, but many details of the execution were flawed. One issue is that that library optimistically creates a fiber for every thread, without knowing whether that thread will ever call into the library. In practice, I suspect most threads of the application will not use the library, particularly thread pool threads or other system-managed threads. As a result, this design creates a lot of fibers that never actually get used. Each fiber has a large stack (since that's the entire point of the exercise), so you end up allocating a lot of memory for each stack, memory that in many cases never gets used. This large per-thread memory allocation basically cuts in half the maximum number of threads the process can create.

There's also the converse problem of a DLL being loaded into a process after it has already created a bunch of threads. When that happens, the DLL has missed out on the `DLL_PROCESS_ATTACH` notifications for the threads that already exist. If any of those

threads call into the library, they will find that there is no pre-created fiber for them, and that the thread may not already have been converted to a fiber.

There's also the converse to the converse, which is that when the DLL is unloaded, it has to destroy the per-thread fibers that it had created (can be done, but takes work), and somehow de-fiber-ize all the threads that it has pre-emptively converted to fibers (not really possible).

Which leads to the biggest problem of this design: The code converts the thread to a fiber inside its `DLL_PROCESS_ATTACH` handler. Converting a thread to a fiber has observable effects to the application, and it's not something that libraries should be doing. It's a case of making changes to something that isn't yours.

Converting a thread to a fiber has an observable effect, because the application itself may call `ConvertThreadToFiber` as the very first thing in its thread procedure, and the call will fail with `ERROR_ALREADY_FIBER`. At this point the application is confused. It created a brand new thread, and the thread is already a fiber? Calling `ConvertThreadToFiber` provides a value that can be obtained by calling `GetFiberData`. If the thread is already a fiber, then the application has no way of passing information to the fiber via its fiber data.

It's not out of the question that the application never checks whether `ConvertThreadToFiber` succeeded, because "This is my thread. I just created it. I can surely convert it to a fiber!" In that case, it gets a fiber handle of `nullptr` and will probably crash when it tries to switch to it.

If you're lucky and the application does check the result of `ConvertThreadToFiber`, it will probably start reporting errors to the user of the form "Could not reverse the polarity of the widget. The current thread has already been converted to a fiber." Which is an error message pretty much guaranteed to create confusing and is likely to cost the application's vendor a support call.

You could try to say that applications which use your library must be prepared for the possibility that their threads have already been converted to fibers by the time they run, and while it's possible that applications can enforce that rule within their own code, they probably won't be able to enforce that rule with code that runs in the same process, but which is not directly under their control.

For example, I have a vague recollection (which could very well be a false memory) that the WinINet system DLL used fibers at one point, and if your library had a rule that applications must be ready to find fibers pre-created on freshly-created threads, that application couldn't use both your library and WinINet (say).

Okay, so I laid out the problems with this proposed design. Next time, we'll look at one solution.

Raymond Chen

Follow