What happens if you simply return from the thread callback passed to _beginthread and _beginthreadex?

devblogs.microsoft.com/oldnewthing/20171115-00

November 15, 2017

Raymond Chen

Medinoc asks, "What happens when one simply returns from the thread callback? I'd suspect the code gluing between _beginthread() and its callback calls _endthread() upon return, while the code between _beginthreadex() and its callback calls _endthreadex() instead?"

Yup, that's exactly it. If your thread callback function returns, then `_beginthread` calls `_endthread` on your behalf, and then `_beginthreadex` calls `_endthreadex` on your behalf. The value passed to `_endthreadex` is the return value of your thread callback function.

In response to the remark "beginthread() initializes the CRT," Cesar asked, "Which CRT? A process can have more than one CRT, and the thread function can even call functions from several different C runtimes."

The `_beginthread` function initializes the CRT it belongs to. What other choice does it have? It's not like `msvcr80!_beginthread` knows how to initialize the data used by `msvcr90.dll`. If you call `msvcr80!_beginthread`, then the new thread is initialized for the `msvcr80` runtime, since that's the only one it knows about.

If the thread function calls into multiple C runtimes, then that's its decision. If it calls into a C runtime that hasn't been initialized for that thread, then what happens next depends on the behavior of that C runtime. For quite some time now, Microsoft's C runtimes are self-initializing, meaning that the first time you call into them on a thread, they will initialize themselves on the spot. And they will also auto-uninitialize themselves when the thread exits.

Wait, if the C runtime initializes itself on demand and auto-uninitializes, then why bother with `_beginthread` at all?

Well, the functions are still around because they predated the initialize-on-demand and auto-uninitialize behavior. And they do guarantee that the C runtime will be initialized for the new thread. (If not, then the functions return failure.) If you go for the initialize-on-demand case, and the C runtime cannot initialize itself, then something interesting happens.

- Some functions will handle the case where the C runtime failed to initialize in some way. for example, `_tempnam` and `strerror` will return `NULL` to report a failure. (Sometimes this failure mode is documented; sometimes it isn't.) Other functions will fall back to a static buffer instead of a per-thread buffer.
- Other functions will exit the process with the error message "`R6016 - not enough space for thread data`."

But as Harry Johnston noted, "In practice very few applications will survive running out of memory anyway."

Joshua Shaeffer asks, "Instead of automatically closing the handle, how about automatically never opening the handle?"

Not sure what Joshua is trying to say here, because the C runtime didn't open the handle. The handle was created by the operating system and returned by the `CreateThread` function. So the C runtime really doesn't have a choice. The handle gets opened as part of the thread-creation process. All it can do is decide what to do with the handle once it is given one.

Raymond Chen

Follow