Consequences of using variables declared __declspec(thread)

devblogs.microsoft.com/oldnewthing/20101122-00

November 22, 2010

Raymond Chen

As a prerequisite, I am going to assume that you understand how TLS works, and in particular how `__declspec(thread)` variables work. There's a quite thorough treatise on the subject by Ken Johnson (better known as Skywing), who comments quite frequently on this site. The series starts here and continues for a total of 8 installments, ending here. That last page also has a table of contents so you can skip over the parts you already know to get to the parts you don't know. Now that you've read Ken's articles... No, wait I know you didn't read them and you're just skimming past it in the hopes that you will be able to fake your way through the rest of this article without having read the prerequisites. Well, okay, but don't be surprised when I get frustrated if you ask a question that is answered in the prerequisites. Anyway, as you learned from Part 5 of Ken's series, the `__declspec(thread)` model, as originally envisioned, assumed that all DLLs which use the feature would be present at process startup, so that all the `_tls_index` values can be computed and the total sizes of each module's TLS data can be calculated before any threads get created. (Well, okay, the initial thread already got created, but that's okay; we'll set up that thread's TLS before we execute any application code.) If you loaded a `__declspec(thread)`-dependent module dynamically, bad things happened. For one, TLS data was not set up for any pre-existing threads, since those threads were initialized before your module got loaded. Windows doesn't have a time machine where it can go back in time to when those threads were initialized and pre-reserve space for the TLS variables your new module needed. Nope, your module is just out of luck with respect to those pre-existing threads, and if it tries to use `__declspec(thread)` variables, it'll find that its TLS slot never got initialized, and there's no data there to access. Unfortunately, there's an even worse problem, which Ken quite ably elaborates on in Part 6: The `_tls_index` variable inside the module arrived after the train left the station. All those TLS indices were assigned at process initialization. When it loads dynamically, the `_tls_index` variable just sits there, and nobody bothers to initialize it, leaving it at its default value of zero. (Too bad the compiler didn't initialize it to `TLS_OUT_OF_INDEXES`.) As a result, the module thinks that its TLS variables are at slot zero in the TLS array, leading to what Ken characterizes as "one of the absolute worst possible kinds of problems to debug": Two modules both think they are the rightful owners of the same data, each with a different concept of what that data is supposed to be. It'd be like if

there was a bug in `HeapAllocate` where it returned the same pointer to two separate callers. Each caller would use the memory, cheerfully believing that the values the code writes to the memory will be there when it comes back. What truly frightens me is that there's at least one person who considers this horrific data corruption bug a *feature*. webcyote calls this bug "sharing all variables between the EXE and the DLL" and complained that fixing the bug breaks programs that "depend on the old behavior". That's like saying "We found that if we use this exact pattern of memory allocations, we can trick `HeapAllocate` into allocating the same memory twice, so we will have our EXE allocate some memory, then perform the magic sequence of allocations, and then load the DLL, and then the DLL will call `HeapAllocate` to allocate some memory, and it will get the same pointer back, and now the EXE and DLL can share memory." Whoa. Mind you, this crazy "EXE and DLL sharing thread variables" trick is extremely fragile. You have to intentionally delay loading the DLL until after process startup. (If you load it as part of an explicit dependency, then you don't trigger the bug and the DLL gets its own set of variables as intended.) And then you have to make sure that the EXE and DLL declare exactly the same variables in exactly the same order and link the OBJ files in exactly the right sequence, so that all the offsets match. Oh, and you have to make sure your DLL is loaded only into the EXE with which it is in cahoots. If you load it into any other EXE, it will start corrupting that EXE's thread variables. (Or, if the EXE doesn't use thread variables, it'll corrupt some other random DLL's thread variables.) If the feature had been intended to be used in this insane way, they would have been called "shared variables" instead of "thread variables". No wait, they would have been called "thread variables that sometimes end up shared under conditions outside your DLL's control." I wonder if Webcyote also drives a manual transmission and just slams the gear stick into position without using the clutch. Yes, you can do it if you are really careful and get everything to align just right, but if you mess up, your transmission explodes and spews parts all over the road. Don't abuse a bug in the loader. If you want shared variables, then create shared variables. Don't create per-thread variables and then intentionally trigger a bug that causes them to overlay each other by mistake. That's such a crazy idea that it probably never occurred to anyone that somebody would actually build a system that relies on it!

Exercise: A customer ran into a problem with the "inadvertently sharing variables between the EXE and the DLL" bug. Here is the message from the customer liaison:

> My customer has a DLL that uses static thread local storage (`__declspec(thread)`), and he wants to use this DLL from his C# program. Unfortunately, he is running into the limitation when running on Windows XP that DLLs which use static thread local storage crash when they try to access their thread variables. The customer cannot modify the DLL. What do you recommend?

Update: Commenter shf gives the most complete answer.

Raymond Chen

Follow