The case of the application that used thread local storage it never allocated

devblogs.microsoft.com/oldnewthing/20221128-00

November 28, 2022

Raymond Chen

An application compatibility issue was found with a program that crashed because its Thread Local Storage (TLS) slots were being corrupted.

Upon closer inspection, the real problem was not that the application's TLS was being corrupted. The problem was that the application was using TLS slots it never allocated, so it was inadvertently using somebody else's TLS slots as its own. And of course, when the true owner updated the TLS value, the application interpreted that as corruption.

It's like just parking your car in a reserved space that belongs to someone else, and then when that other person parks in their own space, you complain, "Hey, what are you doing in my parking space?"

The program wants to allocate 38 TLS slots, and some reverse-compiling reveals that it does so like this:

```
DWORD g_firstTls;

bool AllocateContiguousTlsSlots()
{
    g_firstTls = TlsAlloc();

    for (int i = 1; i < 38; i++) {
        DWORD tls = TlsAlloc();
        if (tls != g_firstTls + i) return false;
    }
    return true;
}
```

The program calls `TlsAlloc()` 38 times, and requires that the values it receives are all consecutive. If any non-consecutive value is received, then it declares that TLS slot allocation has failed.

The problem is that the code that calls `AllocateContiguousTlsSlots()` doesn't check the return value. It assumes that the allocations all succeeded!

What started happening is that the program allocated its first TLS slot and got 9. Then 10, then 11, and so on up to 15. But when it called `TlsAlloc()` again, the slot index it got back was 17, not the expected value of 16. This caused `AllocateContiguousTlsSlots()` to fail, but since the program never checked whether `AllocateContiguousTlsSlots()` succeeded, it just assumed that it had ownership of slots 9 through 46, even though it had only allocated 9 through 15, and then 17. Slots 16 and 18 through 46 were never allocated, but the program used them anyway.

Why did this start happening all of a sudden?

Because I made a performance optimization that reduced the memory usage of the system by 8KB per thread.

Over the decades of its existence, the main DLL used by the shell, `shell32.dll`, accumulated quite a few features, and some of those features require TLS slots, so they allocate the TLS slots at initialization. It got to the point where a program that used `shell32.dll` and other frequently-used DLLs would end up allocating a total of over 64 TLS slots during initialization.

The value of 64 is important, because once the 65th TLS slot is allocated, the kernel goes into "overflow" mode and allocates an extra 1024 TLS slots for each thread, bringing the total to the magic number of 1088. On a 32-bit system, 1024 TLS slots occupy 4KB of memory. On a 64-bit system, it takes 8KB of memory. The result was that nearly every application that used `shell32.dll` (which in practice is nearly every application) was paying an extra 8KB of memory per thread.

But it's rare that a single program exercises every single feature of `shell32.dll`.

I changed the code so that instead of pre-allocating the TLS slots, the various components allocated their TLS slots on demand when the component was used for the first time. Depending on the application, this resulted in a typical savings of 13 to 19 TLS slots, bringing the total number of allocated TLS slots below the magic number of 64.

Hooray, I saved 8KB of memory per thread in nearly every process in the system. It also meant that certain buggy programs would not crash when they allocated a TLS slot and got an index larger than 64.

In the case of the application that was having compatibility problems, the reason they couldn't get their 38 contiguous TLS slots was that I was *too good* at reducing the number of TLS slots used during initialization. With the specific mix of DLLs this particular application used, the number of allocated TLS slots was only 10. The in-use slots were 0 through 8, and 16.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…
•	•	•	•	•	•	•	•	•								•				…

What's up with that weirdo TLS slot out at index 16?

It turns out that one particular TLS slot has to be fixed at index 16 for a different compatibility reason. I don't know the details, but accepted that it was locked to index 16.

The compatibility team wrote a shim that kicks in when you run the program that was making untoward assumptions about TLS slot numbers. If the program calls `TlsAlloc()` and gets an index less than 16, then the compatibility shim loops back and calls `TlsAlloc()` again until it gets a value larger than 16. That way, when this program allocates TLS slots, it starts at slot 17. Early in the initialization of the program, all of the slots from 17 and beyond are all available, so its 38 consecutive calls get consecutive TLS indices 17 through 54.

The `AllocateContiguousTlsSlots()` function succeeds, and the program doesn't crash.

Another day in the life of application compatibility.

Raymond Chen

Follow