Rethinking the way DLL exports are resolved for 32-bit Windows

devblogs.microsoft.com/oldnewthing/20060720-20

July 20, 2006

Raymond Chen

Over the past few days we've learned how 16-bit Windows exported and imported functions from DLLs and that the way functions are exported from 32-bit DLLs matches the 16-bit method reasonably well. But the 16-bit way functions are imported simply doesn't work in the 32-bit world. Recall that in 16-bit Windows, the fixups for an imported function are threaded through the code segment. This works great in 16-bit Windows since there was a single address space: Code segments were shared globally, and once a segment was loaded, each process could use it. But 32-bit Windows uses separate address spaces. If the fixups were threaded through the code segment, then loading a code page from disk would necessarily entail modifying it to apply the fixups, which prevents the pages from being shared by multiple processes. Even if the fixup table were kept external to the code segment, you would still have to fix up the code pages to establish the jump targets. (With sufficient cleverness, you could manage to share the pages if all the fixups on a page happened to agree exactly with those of another process, but the bookkeeping for this would get rather messy.) But beyond just being inefficient, the idea of applying import fixups directly to the code segment is downright impossible. The Alpha AXP has a "call direct" instruction, but it is limited to functions that are at most 128KB away. If you want to call a function that is further away, you have to load the destination address into a temporary register and call through that register. And as we saw earlier, loading a 32-bit value into a register on the Alpha AXP is a two-step operation which depends on whether bit 15 of the value you want to load is set or clear. Since this is an imported function, we have no idea at compile or link time whether the target function's address will have bit 15 set or clear. (And the Alpha AXP was hardly the only architecture that restricted the distance to direct calls. The Intel ia64 can make direct calls to functions up to 4MB away, and the AMD x86-64 and Intel EM64T architectures can reach up to 2GB away. This sounds like a lot until you realize that they are 64-bit processors with 16 exabytes of address space. Once again, we see that the x86 architecture is the weirdo.) Both of the above concerns made it undesirable (or impossible) for import fixups to modify code. Instead, import fixups have to apply to data. Rather than applying a fixup for each location an imported function was used, a single fixup is applied to a table of function pointers. This means that calls to imported functions are really indirect calls through the function pointer. On an x86, this means that instead of `call ImportedFunction` the generated code says `call [__imp__ImportedFunction]`, where `__imp__ImportedFunction` is the name of the variable that holds the function pointer for that imported function. This means that resolving imported functions is a simple matter of looking up the target addresses and writing the results into the table of imported function addresses. The code itself doesn't change; it just reads the function address at runtime and calls through it.

With that simple backgrounder, we are equipped to look at some of the deeper consequences of this design, which we will do next time.

Raymond Chen

Follow