Calling an imported function, the naive way

devblogs.microsoft.com/oldnewthing/20060721-06

July 21, 2006

Raymond Chen

An import library resolves symbols for imported functions, but it isn't consulted until the link phase. Let's consider a naive implementation where the compiler is blissfully unaware of the existence of imported functions. In the 16-bit world, this caused no difficulty at all. The compiler generated a far call instruction and left an external record in the object file indicating that the address of the function should be filled in by the linker. At that time, the linker realizes that the external symbol corresponds to an imported function, so it takes all the call targets, threads them together, and creates an import record in the module's import table. At load time, those call entries are fixed up and everybody is happy. Let's look at how a naive 32-bit compiler would deal with the same situation. The compiler would generate a normal call instruction, leaving the linker to resolve the external. The linker then sees that the external is really an imported function, and, uh-oh, the direct call needs to be converted to an indirect call. But the linker can't rewrite the code generated by the compiler. What's a linker to do? The solution is to insert another level of indirection. (Warning: The information below is not literally true, but it's "true enough". We'll dig into the finer details later in this series.) For each exported function in an import library, two external symbols are generated. The first is for the entry in the imported functions table, which takes the name `__imp__FunctionName`. Of course, the naive compiler doesn't know about this fancy `__imp__` prefix. It merely generates the code for the instruction `call FunctionName` and expects the linker to produce a resolution. That's what the second symbol is for. The second symbol is the longed-for `FunctionName`, a one-line function that consists merely of a `jmp [__imp__FunctionName]` instruction. This tiny stub of a function satisfies the external reference and in turn generates an external reference to `__imp__FunctionName`, which is resolved by the same import library to an entry in the imported function table. When the module is loaded, then, the import is resolved to a function pointer and stored in `__imp__FunctionName`, and when the compiler-generated code calls the `FunctionName` function, it calls the stub which trampolines (via the indirect call) to the real function entry point in the destination DLL. Note that with a naive compiler, if your code tries to take the address of an imported function, it gets the address of the `FunctionName` stub, since a naive compiler simply asks for the address of the `FunctionName` symbol, unaware that it's really coming from an import library.

Next time, we'll look at the `dllexport` declaration specifier and how a less naive compiler generates code for an imported function.

Raymond Chen

Follow