What’s up with the mysterious inc bp in function prologues of 16-bit code?

devblogs.microsoft.com/oldnewthing/20110316-00

March 16, 2011

Raymond Chen

A little while ago, we learned about the EBP chain. The EBP chain in 32-bit code is pretty straightforward because there is only one type of function call. But in 16-bit code there are two types of function calls, the *near call* and the *far call*.

A *near call* pushes a 16-bit return address on the stack before branching to the function entry point, which must reside in the same code segment as the caller. The function then uses a `ret` instruction (a *near return*) when it wants to return to the caller, indicating that the CPU should resume execution at the specified address within the same code segment.

By comparison, a *far call* pushes both the segment (or *selector* if in protected mode) and the offset of the return address on the stack (two 16-bit values), and the function being called is expected to use a `retf` instruction (a *far return*) to indicate that the CPU should pop two 16-bit values off the stack to determine where execution should resume.

When Windows was first introduced, it ran on an 8086 with 384KB of RAM. This posed a challenge because the 8086 processor had no memory manager, had no CPU privilege levels, and had no concept of task switching. And in order to squeeze into 384KB of RAM, Windows needed to be able to load code from disk on demand and discard it when memory pressure required it.

One of the really tricky parts of the real-mode memory manager was fixing up all the function pointers when code was loaded and unloaded. When you unloaded a function, you had to make sure that any existing code in memory that called that function didn’t actually call it, because the function wasn’t there. If you had a memory manager, you could mark the segment or page not present, but there is no such luxury on the 8086.

There are multiple parts to the solution, but the part that leads to the answer to the title question is the way the memory manager patched up all the stacks in the system. After all, if you discarded a function, you had to make sure that any reference to that function as a return address on somebody’s stack got fixed up before the code tried to execute that `retf` instruction and found itself returning to a function that didn’t exist.

And that's where the mysterious `inc bp` came from.

The first rule of stack frames in real-mode Windows is that *you must have a bp-based stack frame*. FPO was not permitted. (Fortunately, FPO was also not very tempting because the 16-bit instruction set made it cumbersome to access stack memory by means other than the `bp` register, so the easiest way to do something was also the right way.) In other words, the first rule required that every stack have a valid `bp` chain at all times.

The second rule of stack frames in real-mode Windows is that if you are going to return with a `retf`, then *you must increment the `bp` register before you push it* (and must therefore perform the corresponding decrement after you pop it). This second rule means that code which walks the `bp` chain can find the next function up the stack. If `bp` is even, then the function will use a near return, so it looks at the 16-bit value stored on the stack after the `bp` and doesn't change the `cs` register. On the other hand, if the `bp` is odd, then it knows to look at both the 16-bit offset and the 16-bit segment that were pushed on the stack.

Okay, so let's put it all together: When code got discarded, the kernel walked all the stacks in the system (which it could now do due to these two rules), and if it saw that a return address corresponded to a function that got discarded, it patched the return address to point to a chunk of code which called back into the memory manager to reload the function, re-patch all the return addresses so they now point to the new address where the function got loaded (probably different from where the function was when it was discarded), and then jumped back to the original code as if nothing had happened.

I continue to be amazed at how much Windows 1.0 managed to accomplish given that it had so little to work with. It even used an LRU algorithm to choose which functions to discard by implementing a software version of the "accessed bit", something that modern CPUs manage in hardware.

Raymond Chen

Follow