What was the role of MS-DOS in Windows 95?

devblogs.microsoft.com/oldnewthing/20071224-00

December 24, 2007

Raymond Chen

Welcome, Slashdot readers. Remember, this Web site is for entertainment purposes only.

Sean wants to know what the role of MS-DOS was in Windows 95. I may regret answering this question since it's clear Slashdot bait. (Even if Sean didn't intend it that way, that's what it's going to turn into.)

Here goes. Remember, what I write here may not be 100% true, but it is "true enough." (In other words, it gets the point across without getting bogged down in nitpicky details.)

MS-DOS served two purposes in Windows 95.

- It served as the boot loader.
- It acted as the 16-bit legacy device driver layer.

When Windows 95 started up, a customized version of MS-DOS was loaded, and it's that customized version that processed your `CONFIG.SYS` file, launched `COMMAND.COM`, which ran your `AUTOEXEC.BAT` and which eventually ran `WIN.COM`, which began the process of booting up the VMM, or the 32-bit virtual machine manager.

The customized version of MS-DOS was fully functional as far as the phrase "fully functional" can be applied to MS-DOS in the first place. It had to be, since it was all that was running when you ran Windows 95 in "single MS-DOS application mode."

The `WIN.COM` program started booting what most people think of as "Windows" proper. It used the copy of MS-DOS to load the virtual machine manager, read the `SYSTEM.INI` file, load the virtual device drivers, and then it turned off any running copy of `EMM386` and switched into protected mode. It's protected mode that is what most people think of as "the real Windows."

Once in protected mode, the virtual device drivers did their magic. Among other things those drivers did was "suck the brains out of MS-DOS," transfer all that state to the 32-bit file system manager, and then shut off MS-DOS. All future file system operations would get routed to the 32-bit file system manager. If a program issued an `int 21h`, the 32-bit file system manager would be responsible for handling it.

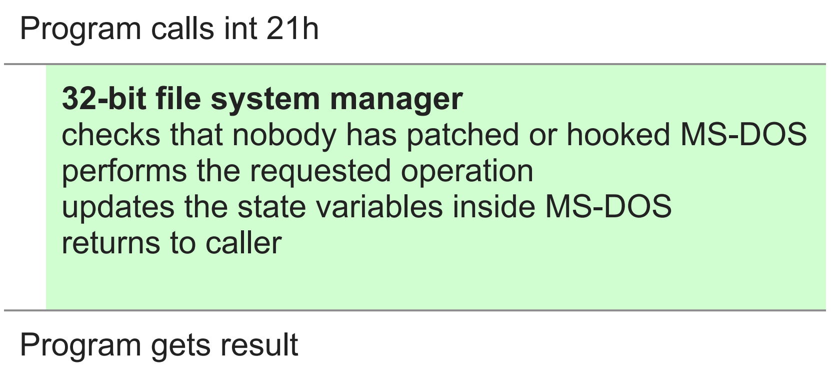
And that's where the second role of MS-DOS comes into play. For you see, MS-DOS programs and device drivers loved to mess with the operating system itself. They would replace the `int 21h` service vector, they would patch the operating system, they would patch the low-level disk I/O services `int 25h` and `int 26h`. They would also do crazy things to the BIOS interrupts such as `int 13h`, the low-level disk I/O interrupt.

When a program issued an `int 21h` call to access MS-DOS, the call would go first to the 32-bit file system manager, who would do some preliminary munging and then, if it detected that somebody had hooked the `int 21h` vector, it would *jump back into the 16-bit code* to let the hook run. Replacing the `int 21h` service vector is logically analogous to subclassing a window. You get the old vector and set your new vector. When your replacement handler is called, you do some stuff, and then call the original vector to do "whatever would normally happen." After the original vector returned, you might do some more work before returning to the original caller.

One of the 16-bit drivers loaded by `CONFIG.SYS` was called `IFSMGR.SYS`. The job of this 16-bit driver was *to hook MS-DOS first* before the other drivers and programs got a chance! This driver was in cahoots with the 32-bit file system manager, for its job was to *jump from 16-bit code back into 32-bit code* to let the 32-bit file system manager continue its work.

In other words, MS-DOS was just an extremely elaborate decoy. Any 16-bit drivers and programs would patch or hook what they thought was the real MS-DOS, but which was in reality just a decoy. If the 32-bit file system manager detected that somebody bought the decoy, it told the decoy to quack.

Let's start with a system that didn't contain any "evil" drivers or programs that patched or hooked MS-DOS.


Program calls int 21h
32-bit file system manager
checks that nobody has patched or hooked MS-DOS
performs the requested operation
updates the state variables inside MS-DOS
returns to caller
Program gets result

This was paradise. The 32-bit file system manager was able to do all the work without having to deal with pesky drivers that did bizarro things. Note the extra step of updating the state variables inside MS-DOS. Even though we extracted the state variables from MS-DOS during the boot process, we keep those state variables in sync because drivers and programs frequently “knew” how those state variables worked and bypassed the operating system and accessed them directly. Therefore, the file system manager had to maintain the charade that MS-DOS was running the show (even though it wasn’t) so that those drivers and programs saw what they wanted.

Note also that those state variables were per-VM. (I.e., each MS-DOS “box” you opened got its own copy of those state variables.) After all, each MS-DOS box had its idea of what the current directory was, what was in the file tables, that sort of thing. This was all an act, however, because the real list of open files was kept in by the 32-bit file system manager. It had to be, because disk caches had to be kept coherent, and file sharing need to be enforced globally. If one MS-DOS box opened a file for exclusive access, then an attempt by a program running in another MS-DOS box to open the file should fail with a sharing violation.

Okay, that was the easy case. The hard case is if you had a driver that hooked `int 21h`. I don’t know what the driver does, let’s say that it’s a network driver that intercepts I/O to network drives and handles them in some special way. Let’s suppose also that there’s some TSR running in the MS-DOS box which has hooked `int 21h` so it can print a 1 to the screen when the `int 21h` is active and a 2 when the `int 21h` completes. Let’s follow a call to a local device (not a network device, so the network driver doesn’t do anything):

Program calls int 21h

32-bit file system manager notices that somebody has patched or hooked MS-DOS jumps to the hook (which is the 16-bit TSR)
16-bit TSR (front end) prints a 1 to the screen calls previous handler (which is the 16-bit network driver)
16-bit network driver (front end) decides that this isn’t a network I/O request calls previous handler (which is the 16-bit IFSMGR hook)
16-bit IFSMGR hook tells 32-bit file system manager that it’s time to make the donuts

32-bit file system manager
regains control
performs the requested operation
updates the state variables inside MS-DOS
returns to caller

16-bit network driver (back end)
returns to caller

16-bit TSR (back end)
prints a 2 to the screen
returns to caller

Program gets result

Notice that all the work is still being done by the 32-bit file system manager. It's just that the call gets routed through all the 16-bit stuff to maintain the charade that 16-bit MS-DOS is still running the show. The only 16-bit code that actually ran (in red) is the stuff that the TSR and network driver installed, plus a tiny bit of glue in the 16-bit IFSMGR hook. Notice that no 16-bit MS-DOS code ran. The 32-bit file manager took over for MS-DOS.

A similar sort of "take over but let the crazy stuff happen if somebody is doing crazy stuff" dance took place when the I/O subsystem took over control of your hard drive from 16-bit device drivers. If it recognized the drivers, it would "suck their brains out" and take over all the operations, in the same way that the 32-bit file system manager took over operations from 16-bit MS-DOS. On the other hand, if the driver wasn't one that the I/O subsystem recognized, it let the driver be the one in charge of the drive. If this happened, it was said that you were going through the "real-mode mapper" since "real mode" was name for the CPU mode when protected mode was not running; in other words, the mapper was letting the 16-bit drivers do the work.

Now, if you were unlucky enough to be using the real-mode mapper, you probably noticed that system performance to that drive was pretty awful. That's because you were using the old clunky single-threaded 16-bit drivers instead of the faster, multithread-enabled 32-bit drivers. (When a 16-bit driver was running, no other I/O could happen because 16-bit drivers were not designed for multi-threading.)

This awfulness of the real-mode mapper actually came in handy in a backwards way, because it was an early indication that your computer got infected with an MS-DOS virus. After all, MS-DOS viruses did what TSRs and drivers did: They hooked interrupt vectors and took over control of your hard drive. From the I/O subsystem's point of view, they looked just like a 16-bit hard disk device driver! When people complained, "Windows suddenly started running really slow," we asked them to look at the system performance page in the control panel and see if it says that “Some drives are using MS-DOS compatibility.” If so, then it meant that the real-mode mapper was in charge, and if you didn’t change hardware, it probably means a virus.

Now, there are parts of MS-DOS that are unrelated to file I/O. For example, there are functions for allocating memory, parsing a string containing potential wildcards into FCB format, that sort of thing. Those functions were still handled by MS-DOS since they were just “helper library” type functions and there was no benefit to reimplementing them in 32-bit code aside from just being able to say that you did it. The old 16-bit code worked just fine, and if you let it do the work, you preserved compatibility with programs that patched MS-DOS in order to alter the behavior of those functions.

Raymond Chen

Follow