The story of the mysterious WINA20.386 file

devblogs.microsoft.com/oldnewthing/20120206-00

February 6, 2012

Raymond Chen

matushorvath was curious about the WINA20.386 file that came with some versions of MS-DOS.

The WINA20.386 file predates my involvement, but I was able to find some information on the Internet that explained what it was for. And it's right there in KB article Q68655: Windows 3.0 Enhanced Mode Requires WINA20.386:

> **Windows 3.0 Enhanced Mode Requires WINA20.386**
>
> Windows 3.0 enhanced mode uses a modular architecture based on what are called virtual device drivers, or VxDs. VxDs allow pieces of Windows to be replaced to add additional functionality. WINA20.386 is such a VxD. (VxDs could be called "structured" patches for Windows.)
>
> Windows 3.0 enhanced mode considers the state of the A20 line to be the same in all MS-DOS virtual machines (VMs). When MS-DOS is loaded in the high memory area (HMA), this can cause the machine to stop responding (hang) because of MS-DOS controlling the A20 line. If one VM is running inside the MS-DOS kernel (in the HMA) and Windows task switches to another VM in which MS-DOS turns off A20, the machine hangs when switching back to the VM that is currently attempting to execute code in the HMA.
>
> WINA20.386 changes the way Windows 3.0 enhanced mode handles the A20 line so that Windows treats the A20 status as local to each VM, instead of global to all VMs. This corrects the problem.

(At the time I wrote this, a certain popular Web search engine kicks up as the top hit for the exact phrase "Windows 3.0 Enhanced Mode Requires WINA20.386" a spam site that copies KB articles in order to drive traffic. Meanwhile, the actual KB article doesn't show up in the search results. Fortunately, Bing got it right.)

That explanation is clearly written for a technical audience with deep knowledge of MS-DOS, Windows, and the High Memory Area. matushorvath suggested that "a more detailed explanation could be interesting." I don't know if it's interesting; to me, it's actually quite

boring. But here goes.

The A20 line is a signal on the address bus that specifies the contents of bit 20 of the linear address of memory being accessed. If you aren't familiar with the significance of the A20 line, this Wikipedia article provides the necessary background.

The High Memory Area is a 64KB-sized block of memory (really, 64KB minus 16 bytes) that becomes accessible when the CPU is in 8086 mode but the A20 line is enabled. To free up conventional memory, large portions of MS-DOS relocate themselves into the HMA. When a program calls into MS-DOS, it really calls into a stub which enables the A20 line, calls the real function in the HMA, and then disables the A20 line before returning to the program. (The value of the HMA was discovered by my colleague who also discovered the fastest way to get out of virtual-8086 mode.)

The issue is that by default, Windows treats all MS-DOS device drivers and MS-DOS itself as global. A change in one virtual machine affects all virtual machines. This is done for compatibility reasons; after all, those old 16-bit device drivers assume that they are running on a single-tasking operating system. If you were to run a separate copy of each driver in each virtual machine, each copy would try to talk to the same physical device, and bad things would happen because each copy assumed it was the only code that communicated with that device.

Suppose MS-DOS device drivers were treated as local to each virtual machine. Suppose you had a device driver that controlled a traffic signal, and as we all know, one of the cardinal rules of traffic signals is that you never show green in both directions. The device driver has two variables: NorthSouthColor and EastWestColor, and initially both are set to Red. The copy of the device driver running in the first virtual machine decides to let traffic flow in the north/south direction, and it executes code like this:

```
if (EastWestColor != Red) {
 SetEastWestColor(Red);
}
SetNorthSouthColor(Green);
```

Since both variables are initially set to Red, this code sets the north/south lights to green.

Meanwhile, the copy of the device driver in the second virtual machine wants to let traffic flow in the east/west direction:

```
if (NorthSouthColor != Red) {
 SetNorthSouthColor(Red);
}
SetEastWestColor(Green);
```

Since we have a separate copy of the device driver in each virtual machine, the changes made in the first virtual machine do not affect the values in the second virtual machine. The second virtual machine sees that both variables are set to Red, so it merely sets the east/west color to green.

On the other hand, both of these device drivers are unwittingly controlling the same physical traffic light, and it just got told to set the lights in both directions to Green.

Oops.

Okay, so Windows defaults drivers to global. That way, you don't run into the double-bookkeeping problem. But this causes problems for the code which manages the A20 line:

Consider a system with two virtual machines. The first one calls into MS-DOS. The MS-DOS dispatcher enables the A20 line and calls the real function, but before the function returns, the virtual machine gets pre-empted. The second virtual machine now runs, and it too calls into MS-DOS. The MS-DOS dispatcher in the second virtual machine enables the A20 line and calls into the real function, and after the function returns, the second virtual machine disables the A20 line and returns to its caller. The second virtual machine now gets pre-empted, and the first virtual machine resumes execution. Oops: It tries to resume execution in the HMA, but the HMA is no longer there because the second virtual machine disabled the A20 line!

The WINA20.386 driver teaches Windows that the state of the A20 should be treated as a per-virtual-machine state rather than a global state. With this new information, the above scenario does not run into a problem because the changes to the A20 line made by one virtual machine have no effect on the A20 line in another virtual machine.

matushorvath goes on to add, "I would be very interested in how Windows 3.0 found and loaded this file. It seems to me there must have been some magic happening, e.g. DOS somehow forcing the driver to be loaded by Windows."

Yup, that's what happened, and there's nothing secret about it. When Windows starts up, it broadcasts an interrupt. TSRs and device drivers can listen for this interrupt and respond by specifying that Windows should load a custom driver or request that certain ranges of data should be treated as per-virtual-machine state rather than global state (known to the Windows virtual machine manager as instance data). MS-DOS itself listens for this interrupt, and when Windows sends out the "Does anybody have any special requests?" broadcast, MS-DOS responds, "Yeah, please load this WINA20.386 driver."

So there you have it, the story of WINA20.386. Interesting or boring?

Raymond Chen

Follow