What did Windows 3.1 do when you hit Ctrl+Alt+Del?

devblogs.microsoft.com/oldnewthing/20140912-00

September 12, 2014

Raymond Chen

This is the end of `Ctrl`+`Alt`+`Del` week, a week that sort of happened around me and I had to catch up with.

The Windows 3.1 virtual machine manager had a clever solution for avoiding deadlocks: There was only one synchronization object in the entire kernel. It was called "the critical section", with the definite article because there was only one. The nice thing about a system where the only available synchronization object is a single critical section is that deadlocks are impossible: The thread with the critical section will always be able to make progress because the only thing that could cause it to stop would be blocking on a synchronization object. But there is only one synchronization object (the critical section), and it already owns that.

When you hit `Ctrl`+`Alt`+`Del` in Windows 3.1, a bunch of crazy stuff happened. All this work was in a separate driver, known as the *virtual reboot device*. By convention, all drivers in Windows 3.1 were called the *virtual something device* because their main job was to virtualize some hardware or other functionality. That's where the funny name VxD came from. It was short for *virtual x device*.

First, the virtual reboot device driver checked which virtual machine had focus. If you were using an MS-DOS program, then it told all the device drivers to clean up whatever they were doing for that virtual machine, and then it terminated the virtual machine. This was the easy case.

Otherwise, the focus was on a Windows application. Now things got messy.

When the 16-bit Windows kernel started up, it gave the virtual reboot device the addresses of a few magic things. One of those magic things was a special byte that was set to 1 every time the 16-bit Windows scheduler regained control. When you hit `Ctrl`+`Alt`+`Del`, the virtual reboot device set the byte to 0, and it also registered a callback with the virtual machine manager to say "Call me back once the critical section becomes available." The callback didn't do anything aside from remember the fact that it was called at all. And then the code waited for ¾ seconds. (Why ¾ seconds? I have no idea.)

After ¾ seconds, the virtual reboot device looked to see what the state of the machine was.

If the “call me back once the critical section becomes available” callback was never called, then the problem is that a device driver is stuck in the critical section. Maybe the device driver put an *Abort, Retry, Ignore* message on the screen that the user needs to respond to. The user saw this message:

```
 Procomm
This background non-Windows application is not responding.
*  Press any key to activate the non-Windows application.
*  Press CTRL+ALT+DEL again to restart your computer. You will
   lose any unsaved information.

  Press any key to continue _
```

After the user presses a key, focus was placed on the virtual machine that holds the critical section so the user can address the problem. A user who is still stuck can hit `Ctrl`+`Alt`+`Del` again to restart the whole process, and this time, execution will go into the “If you were using an MS-DOS program” paragraph, and the code will shut down the stuck virtual machine.

If the critical section was not the problem, then the virtual reboot device checked if the 16-bit kernel scheduler had set the byte to 1 in the meantime. If so, then it means that no applications were hung, and you got the message

```
 Windows
Although you can use CTRL+ALT+DEL to quit an application that has stopped responding
to the system, there is no application in this state.
To quit an application, use the application’s quit or exit command, or choose the Close
command from the Control menu.

*  Press any key to return to Windows.
*  Press CTRL+ALT+DEL again to restart your computer. You will
   lose any unsaved information in all applications.

  Press any key to continue _
```

(Anachronism alert: The System menu was called the Control menu back then.)

Otherwise, the special byte was still 0, which means that the 16-bit scheduler never got control, which meant that a 16-bit Windows application was not releasing control back to the kernel. The virtual reboot device then waited for the virtual machine to finish processing any pending virtual interrupts. (This allowed any pending MS-DOS emulation or 16-bit MS-DOS device drivers to finish up their work.) If things did not return to this sane state within 3¼ seconds, then you got this screen:

```
 Windows
The system is either busy or has become unstable. You can wait and see if the system
becomes available again and continue working or you can restart your computer.
*  Press any key to return to Windows and wait.
*  Press CTRL+ALT+DEL again to restart your computer. You will
   lose any unsaved information in all applications.

  Press any key to continue _
```

Otherwise, we are in the case where the system returned to a state where there are no active virtual interrupts. The kernel single-stepped the processor if necessary until the instruction pointer was no longer in the kernel, or until it had single-stepped for 5000 instructions and the instruction pointer was not in the heap manager. (The heap manager was allowed to run for more than 5000 instructions.)

At this point, you got the screen that Steve Ballmer wrote.

```
Contoso Deluxe Music Composer
  This Windows application has stopped responding to the system.
  *  Press ESC to cancel and return to Windows.
  *  Press ENTER to close this application that is not responding.
     You will lose any unsaved information in this application.
  *  Press CTRL+ALT+DEL again to restart your computer. You will
     lose any unsaved information in all applications.
```

If you hit `Enter`, then the 16-bit kernel terminated the application by doing `mov ax, 4c00h` followed by `int 21h`, which was the system call that applications used to exit normally. This time, the kernel is making the exit call on behalf of the stuck application. Everything looks like the application simply decided to exit normally.

The stuck application exits, the kernel regains control, and hopefully, things return to normal.

I should point out that I didn’t write any of this code. “It was like that when I got here.”

Bonus chatter: There were various configuration settings to tweak all of the above behavior. For example, you could say that `Ctrl`+`Alt`+`Del` always restarted the computer rather than terminating the current application. Or you could skip the check whether the 16-bit kernel scheduler had set the byte to 1 so that you could use `Ctrl`+`Alt`+`Del` to terminate an application even if it wasn’t hung.[1] There was also a setting to restart the computer upon receipt of an NMI, the intention being that the signal would be triggered either by a dedicated add-on switch or by poking a ball-point pen in just the right spot. (This is safer than just pushing the reset button because the restart would flush disk caches and shut down devices in an orderly manner.)

[1] This setting was intended for developers to assist in debugging their programs because if you went for this option, the program that got terminated is whichever one happened to have control of the CPU at the time you hit `Ctrl`+`Alt`+`Del`. This was, in theory, random, but in practice it often guessed right. That's because the problem was usually that a program got wedged into an infinite message loop, so most of the CPU was being run in the stuck application anyway.

Raymond Chen

Follow