If 16-bit Windows had a single input queue, how did you debug applications on it?

devblogs.microsoft.com/oldnewthing/20141126-00

November 26, 2014

Raymond Chen

After learning about the bad things that happened if you synchronized your application's input queue with its debugger, commenter kme wonders how debugging worked in 16-bit Windows, since 16-bit Windows didn't have asynchronous input? In 16-bit Windows, all applications shared the same input queue, which means you were permanently in the situation described in the original article, where the application and its debugger (and everything else) shared an input queue and therefore would constantly deadlock. The solution to UI deadlocks is to make sure the debugger doesn't have any UI. At the most basic level, the debugger communicated with the developer through the serial port. You connected a dumb terminal to the other end of the serial port. Mine was a Wyse 50 serial console terminal. All your debugging happened on the terminal. You could disassemble code, inspect and modify registers and memory, and even patch new code on the fly. If you wanted to consult source code, you needed to have a copy of it available somewhere else (like on your other computer). It was similar to using the `cdb` debugger, where the only commands available were `r`, `db`, `eb`, `u`, and `a`. Oh, and `bp` to set breakpoints. Now, if you were clever, you could use a terminal emulator program so you didn't need a dedicated physical terminal to do your debugging. You could connect the target computer to your development machine and view the disassembly and the source code *on the same screen*. But you weren't completely out of the woods, because what did you use to debug your development machine if it crashed? The dumb terminal, of course.[1]

Target machine ← Development machine ← Wyse 50
Debugger Debugger dumb terminal

I did pretty much all my Windows 95 debugging this way. If you didn't have two computers, another solution was to use a debugger like CodeView. CodeView avoided the UI deadlock problem by not using the GUI to present its UI. When you hit a breakpoint or otherwise halted execution of your application, CodeView talked directly to the video driver to save the first 4KB of video memory, then switched into text mode to tell you what happened. When you resumed execution, it restored the video memory, then switched the video card back into

graphics mode, restored all the pixels it captured, then resumed execution as if nothing had happened. (If you were debugging a graphics problem, you could hit F3 to switch temporarily to graphics mode, so you could see what was on the screen.) If you were *really fancy*, you could spring for a monochrome adapter, either the original IBM one or the Hercules version, and tell CodeView to use that adapter for its debugging UI. That way, when you broke into the debugger, you could still see what was on the screen! *We had multiple monitors before it was cool.*

[1] Some people were crazy and cross-connected their target and development machines.

Target machine	Development machine
Debugger	Debugger

This allowed them to use their target machine to debug their development machine and vice versa. But if your development machine crashed *while it was debugging the target machine*, then you were screwed.

Raymond Chen

Follow