The Windows 95 I/O system assumed that if it wrote a byte, then it could read it back

devblogs.microsoft.com/oldnewthing/20150827-00

August 27, 2015

Raymond Chen

In Windows 95, compressed data was read off the disk in three steps.

1. The raw compressed data was read into a temporary buffer.
2. The compressed data was uncompressed into a second temporary buffer.
3. The uncompressed data was copied to the application-provided I/O buffer.

But you could save a step if the I/O buffer was a full cluster:

1. The raw compressed data was read into a temporary buffer.
2. The compressed data was uncompressed directly into the application-provided I/O buffer.

A common characteristic of dictionary-based compression is that a compressed stream can contain a code that says "Generate a copy of bytes X through Y from the existing uncompressed data."

As a simplified example, suppose the cluster consisted of two copies of the same 512-byte block. The compressed data might say "Take these 512 bytes and copy them to the output. Then take bytes 0 through 511 of the uncompressed output and copy them to the output."

So far, so good.

Well, except that if the application wrote to the I/O buffer while the read was in progress, then the read would get corrupted because it would copy the wrong bytes to the second half of the cluster.

Fortunately, writing to the I/O buffer is forbidden during the read, so any application that pulled this sort of trick was breaking the rules, and if it got corrupted data, well, that's its own fault. (You can construct a similar scenario where writing to the buffer during a write can result in corrupted data being written to disk.)

Things got even weirder if you passed a memory-mapped device as your I/O buffer. There was a bug that said, “The splash screen for this MS-DOS game is all corrupted if you run it from a compressed volume.”

The reason was that the game issued an I/O directly into the video frame buffer. The EGA and VGA video frame buffers used planar memory and latching. When you read or write a byte in video memory, the resulting behavior is a complicated combination of the byte you wrote, the values in the latches, other configuration settings, and the values already in memory. The details aren’t important; the important thing is that *video memory does not act like system RAM*. Write a byte to video memory, then read it back, and not only will you not get the same value back, but you probably modified video memory in a strange way.

The game in question loaded its splash screen by issuing I/O directly into video memory, knowing that MS-DOS copies the result into the output buffer byte by byte. It set up the control registers and the latches in such a way that then bytes written into memory go exactly where they should. (It issued four reads into the same buffer, with different control registers each time, so that each read ended up being issued to a different plane.)

This worked great, unless the disk was compressed.

The optimization above relied on the property that writing a byte followed by reading the byte produces the byte originally written. But this doesn’t work for video memory because of the weird way video memory works. The result was that when the decompression engine tried to read what it thought was the uncompressed data, it was actually asking the video controller to do some strange operations. The result was corrupted decompressed data, and corrupted video data.

The fix was to force double-buffering in non-device RAM if the I/O buffer was into device-mapped memory.

Raymond Chen

Follow