The inside story of the outside investigation of SoftRAM 95

devblogs.microsoft.com/oldnewthing/20211111-00

November 11, 2021

Raymond Chen

With the release of Windows 95 came quite a number of software products tailored to run on it. One of them that drew a lot of attention at the time was SoftRAM 95, a product whose box claimed it would “Double Your Memory”.

It turns out that it didn’t.

There are many write-ups of what the software did (and notably didn’t do), but almost none of them dug deep into the code to explain what it was doing.

So I’m writing one.

In late 1995, I was asked to investigate the product because it was causing Windows 95 machines to crash and was generating a lot of support calls, not to mention bad PR. It’s possible that the software was tripping over some bug in Windows 95, and that meant having to put out a patch for it.

I ended up doing a full disassembly of the custom paging driver they installed. The Windows 3.0 and 3.1 device driver kits (DDKs) included the source code for the paging driver, and the software’s custom paging driver was clearly based on the one that shipped in one of the older DDKs.

What they did was allocate a chunk of non-paged memory at system startup and use it as a compression pool. When memory was paged out, the driver compressed the memory and added it to the pool. The format of each block in the compression buffer was a byte specifying the compression algorithm, followed by the compressed data.

When the system asked to page in some memory, the driver checked if there was a compressed copy in the pool. If so, it decompressed the data and returned it. If not, then it read the memory from disk in the usual manner.

To explain what's going on here, let's draw some pictures. We'll rank the pages in the system based on how likely they are to be used, with hot pages (most likely to be used) closer to the left and and cold pages (least likely to be used) closer to the right. An unmodified system looks like this:

…

In memory (fast) On disk (slow)

In the above example of a hypothetical system, we have the 16 hottest pages in memory, and the remaining pages on disk. The size of the "In memory" section is the amount of RAM you have. The size of the "On disk" section can grow up to the maximum size of your page file.

When a new page is allocated, it is inserted on the left as the hottest page. Since the size of the "In memory" section is fixed, that pushes the lowest-ranked in-memory page to disk. In general, nearly all of the pages in the system are cold, so the above diagram is not to scale. The "On disk" section is far bigger than the "In memory" section, but all the excitement is in the "In memory" section, so I've zoomed in on that part.

The idea is to convert some of those RAM pages into compressed RAM pages:

In memory (fast) Compressed (medium) On disk (slow)

I've taken half of the RAM in the system and compressed it. The diagram shows the compressed pages smaller than normal than normal pages due to the compression, to emphasize that the total amount of RAM hasn't changed. What changed is how the memory is being used.

But I can redraw the diagram so that each page is the same size. This is the view of the system from point of view of applications and the memory manager:

In memory (fast) Compressed (medium) On disk (slow)

From the point of view of the rest of the system, it looks like you gained more memory, sort of.

Compressed memory isn't immediately usable like regular memory: When an application requests access to memory that has been compressed, the compressed data is uncompressed into a normal page, and the lowest-ranked normal page becomes a compressed page.[1] This compression and decompression takes time, but since you're competing against a hard drive, it'll still be faster than disk I/O.

The hope is that even though you took away a bunch of fast pages, you replaced them with a larger number of medium-speed pages, thereby lowering the number of accesses to *slow* pages. In the above example, we're assuming a compression ratio of 2x, so we took away 8 fast pages and turned them into 16 medium-speed pages.

	Before	**After**
Normal pages	16	8
Compressed pages		16
Pages on disk	*N*	*N* − 8

Whether this is a net win depends on the memory access patterns of the applications you use. If you had an application that had 6 very hot pages, and 14 additional warm pages, then this could very well be an improvement, since the very hot pages could stay in normal memory, and the 14 warm pages could take turns occupying the two remaining normal memory pages. On the other hand, If you had an application that had 10 very hot pages, and 10 additional warm pages, then this could end up a net loss, because there aren't enough normal pages to hold the application's very hot pages, so you are constantly compressing and decompressing memory, and that extra time spent on compression could exceed the time saved by avoiding the I/O to the four pages that didn't fit in RAM before.

Okay, so we see that the design hinges on the quality of the compression engine. You want to be able to squeeze as much data into those compressed pages so that you can more than make up for the loss of normal pages with a much larger number of compressed pages.

Back to the analysis.

I found the compression algorithm. It was called to take the memory from the I/O buffer and compress it into the compression buffer. Its counterpart decompression algorithm was used on page-in requests to decompress the data in the compression buffer and put it in the I/O buffer.

They implemented only one compression algorithm.

It was `memcpy`.

In other words, their vaunted patent-pending compression algorithm was “copy the data uncompressed.”

The whole compression architecture was implemented, with a stub compression function that did no compression, presumably with the idea that “Okay, and then we’ll put an awesome compression function here, but for now, we’ll just use this stub function so we can validate our design.” But they ran out of time and shipped the stub.

It’s like creating a RAM drive for your swap file. All you did was take some pages away from the “fast” category and create the same number of pages in the “medium” category.

So far, what we have is a placebo with some performance degradation. Why was it crashing? Did their design uncover a bug in the Windows 95 memory manager?

When they added code to implement the compression buffer, they didn’t use critical sections or any other synchronization primitives to protect the data structures. If two threads started paging at the same time, the driver corrupted its data structures due to concurrency. The next time the driver went to uncompress the data for a page, it got confused and produced the wrong memory, and that’s why Windows 95 was crashing.

They were inadvertently simulating a broken hard drive.

This also explained why the crashes usually happened when the system was under heavy paging load: Under those conditions, there is more likelihood that two paging requests will overlap, thereby triggering the corruption.

Oh, remember that I said that their driver was based on the one included in the DDK? They didn’t bother to change the name in the device driver description block, so when the driver crashed, the crash was blamed on `PAGEFILE`. Since this is the name of the default paging driver, the crash message made it look like a default Windows driver had crashed, when in fact it was their substitute with the same name.

Bonus reading: I was not the only one to do an analysis of the internals of SoftRAM 95. Some guy named Mark Russinovich also took the product apart and came to the same conclusions. I wonder what happened to that guy. He seems kind of sharp.

Bonus chatter: A colleague of mine noted that SoftRAM 95 was a follow-up to a similar product designed for Windows 3.1. The Windows 3.1 version tuned your system in two ways to increase the number of programs you could run at once: First, it increased the size of your page file, which is something you could have done manually anyway. Second, it used some tricks to keep certain components out of conventional memory. These tricks were well-known to system optimization tools, and you could even get a utility from Microsoft for free that did the same thing, written by that very colleague. Windows 95 used a dynamic page file, and it addressed the conventional memory problem as well, leaving the Windows 3.1 of their product with nothing to do when ported to Windows 95.

Bonus bonus chatter: I heard through the grapevine (by way of marketing) that the company had two developers for the product, one for the device driver and the other for the user interface. So I guess what they were missing was a third developer to write the compression engine.

Bonus bonus bonus chatter: My boss at the time was sent to staff a booth at a trade show. One of the other companies at the show was the manufacturer of SoftRAM 95. When the show concluded, he went over and traded a Microsoft staff shirt for one of their shirts, and then gave it to me.

Bonus bonus bonus bonus chatter: How did this program manage to earn a "Designed for Windows 95" sticker? Easy: Because it was designed for Windows 95. The logo program rules say that you have to follow certain guidelines, like using the standard File Open dialog to select files and installing into the Program Files directory. The program followed those guidelines, so they got the sticker. Some people point out that it should have failed the "functions substantially as described" rule, but that rule wasn't saying that every claim must be independently verified by an in-depth reverse-engineering. If that were the standard, then it could take years to pass that test. Validating a "Learn German in 30 Days" program would have to verify that all the vocabulary and grammar are correct, and then have somebody use the program for 30 days and see if they learned German. What the rule is trying to do is catch a program that fails to run on the minimum system configuration, or which purports to be a word processor, but in fact plays tic-tac-toe.

[1] From what I recall (but my recollection could be faulty), the compressed data was managed in a simple way: In a circular buffer. Whenever a new page was compressed, it was appended to the buffer, and whenever a page needed to be evicted, the page at the head of the buffer was chosen. This makes bookkeeping very simple, but it does have its own inefficiencies: When a page got paged in from the compression buffer, and then was paged back out, the memory for the old compressed data still hung around in the compression buffer wasting space. In theory, they could have reused that memory right away, but doing so would have complicated the algorithms since you now have fragmentation to deal with. The circular buffer design has the advantage of having extremely simple memory management. This was back in the days when device drivers were all written in assembly language. Simplicity of design is very important because you're implementing that design instruction by instruction.

Raymond Chen

Follow