Getting MS-DOS games to run on Windows 95: Virtual memory

devblogs.microsoft.com/oldnewthing/20160328-00

March 28, 2016

Raymond Chen

A lot of games of the pre-Windows 95 era use so-called MS-DOS Extenders, which are libraries that provide a protected-mode environment to MS-DOS applications. The application is technically two programs glued together: The first program is the MS-DOS Extender itself, also known as the server, and the second program is the game that is a client of the MS-DOS Extender.

The interface that the client uses to talk to the extender takes a variety of forms. There were a handful of extenders that made up their own interface from scratch, but most extenders implement either the VCPI or DPMI interface. The main difference that mattered to me is that VCPI gives the application full control of the CPU at ring zero, which means that it cannot coexist with any other operating system. DPMI is a much friendlier interface to a host operating system, since it allows the host operating system to remain in control while still granting the client access to protected mode.

One nice feature of the DPMI extenders is that when they start up, they look to see if they are already running inside a DPMI extender. If so, then the extender shuts itself off and allows the client to talk directly to the existing DPMI extender. (In the case of a program running inside an MS-DOS virtual machine created by Windows, the existing DPMI extender is the Windows virtual machine manager.)

Now things get interesting: The client application was written with the assumption that it is using the MS-DOS extender that is included with the application, but in reality it is talking to the DPMI host that comes with Windows. The fact that programs seem to run mostly okay in spite of running under a foreign extender is either completely astonishing or totally obvious, depending on your point of view. It’s completely astonishing because, well, you’re taking a program written to be run in one environment, and running it in a different environment. Or it’s totally obvious because they are using the same DPMI interface, and as long as the interface has the same behavior, then naturally the program will continue to work, because that’s why we have interfaces!

In practice, the issues that arose with running under Windows DPMI instead of the built-in extender's DPMI fell into a few categories, all due to differences between the implementations, despite both adhering to the documented interface.[1]

One is virtual memory.

The built-in extender didn't implement virtual memory, and client applications often use programming techniques that don't work well in a virtual memory environment.

Virtual memory introduced latency that applications hadn't been designed for. They would allocate memory and put an interrupt handler in it. This works fine if there is no virtual memory, but once you enable virtual memory, the memory for the interrupt handler might get paged out, and then bad things would happen, usually race conditions.

A more common programming pattern that falls down in the face of virtual memory is an application that simply allocates all the memory in the system and adds it to a memory pool. (As a bonus, the applications often initialize the memory as it was allocated.) If you have virtual memory, the client application can go a very long time before it runs out of memory because each "memory" allocation is really a disk allocation. This typically manifested itself by the application taking a long time to start up, accompanied by heavy disk activity as the operating system swapped out tons of pages until you ran out of disk space.

As I recall, we fixed this by setting a default policy for MS-DOS applications to limit EMS and XMS memory usage to the actual amount of physical RAM installed in the computer. (The DPMI memory quota defaulted to 8MB or a little bit less than physical memory, whichever is lower.) That way, these programs that try to allocate all the memory in the system would give up before the swap file spiraled out of control. This was the setting known as *Auto* in the memory properties page.

Bonus chatter: There was one program that not only allocated all the memory in the system and added it to a memory pool. Later during the execution of the program, it would ask for still more memory, and if the call succeeded, the program crashed!

[1]These issues were called out in the DPMI documentation, but since the applications assumed that they were running under their built-in extender, they figured that the warnings in the DPMI documentation didn't apply to them. It never occurred to them that their preferred DPMI extender would not actually be the one in charge.

Raymond Chen

Follow