If I create multiple selectors each of size 4GB, do I get a combined address space larger than 4GB?

devblogs.microsoft.com/oldnewthing/20160606-00

June 6, 2016

Raymond Chen

Every so often, someone comes up with the clever idea of extending the address space of the x86 processor beyond 4GB by creating multiple selectors, each of size 4GB. For example, if you created a 4GB selector for code, another 4GB selector for stack, and another 4GB selector for data, and assigned them distinct memory ranges, then you could load up each selector into the corresponding register (CS, SS, DS) and be able to access 12GB of memory.

Profit!

Well, except that it doesn't actually work.

Segment descriptors on the x86 contain the following pieces of information:

- Various control bits not relevant to this discussion.
- A segment base address (32 bits).
- A segment limit (32 bits, encoded as a 20-bit value and an optional scale; details not important).

In practice, what happens is that the base address is set to zero and the limit is set to `0xFFFFFFFF`, which gives each segment a range of 4GB.

Segments create views into the linear address space. When you access memory by doing, say, `mov al, ds:[ebx]`, what happens is the following:

- The selector in the `ds` register is consulted to obtain its base address and limit. If `ds` references an invalid selector, then a fault occurs.
- The value in `ebx` is checked against the segment limit of the selector held in `ds`. If it is greater than the limit, then a fault occurs.
- The value in `ebx` is added to the selector's base address, producing a linear adddress.
- That linear address is used to access the underlying memory.

The mechanism by which linear addresses map to physical addresses is not relevant to the discussion. (This is where page tables come in.) I'm also ignoring expand-down selectors and other details not related to addressing.

In other words, selectors don't reference memory direcrly. They are merely a window into the linear address space. If you create a selector whose base address is inside the [base address, base address + offset] range of another selector, then both selectors are accessing the same underlying memory.

Linear address space
Selector X
Selector Y
In the above example, we created Selector X with a base address of `0x50000000` and a limit of `0x1FFFFFFF`. This gives selector X a reach of [`0x50000000`, `0x6FFFFFFF`]: An access to `X:0` refers to linear address `0x50000000`, and an access to `X:1FFFFFFF` refers to linear address `0x6FFFFFFF`. Higher offsets from selector X are invalid.

We also created Selector Y with a base address of `0x60000000` and a limit of `0x7FFFFFFF`, giving selector Y a reach of [`0x60000000`, `0xDFFFFFFF`].

Observe that the two selectors overlap. The addresses `X:10000000` and `Y:00000000` refer to the same underlying linear address space. Write a value to to `X:10000000` and you can read it back from `Y:00000000`.

Indeed, this behavior on overlap is relied upon constantly. To use the x86 in flat mode, you create a code selector and a data selector, both of which have a base of `0x00000000`. and a limit of `0xFFFFFFFF`. You put the code selector in the `cs` register and the data selector in the `ss`, `ds`, and `es` registers. The fact that the ranges perfectly overlap means that reading data from a code address reads the same bytes that the CPU would have executed. Conversely, the fact that they overlap means that you can generate code by writing to the data segment.

Okay, you sigh, I can't give each selector its own 4GB of address space. The fact that the base address of the selector is a 32-bit value means that the best I can do is to create a selector whose base is `0xFFFFFFF0` and whose limit is `0xFFFFFFFF`; that at least gives me linear addresses as high as `0xFFFFFFF0 + 0xFFFFFFFF`, or a smidge under 8GB. Still, 8GB is better than 4GB, right?

Well, you don't even get 8GB.

> **3.3.5 32-Bit and 16-Bit Address and Operand Sizes**
>
> With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH ($2^{32} - 1$).

“The maximum linear address is FFFFFFFFH.”

This means that segments whose base + limit is greater than `0xFFFFFFFF` are illegal. All of your selectors have to fit inside [`0x00000000` , `0xFFFFFFFF`].

Now, maybe you could pull some super sneaky tricks like keeping all pages mapped not present, and then when a page fault occurs, determining which selector was the source of the faulting linear address and mapping in the appropriate page at fault time, and then setting the trap flag so that the kernel regains control after the instruction has executed, so that you can unmap the page immediately. But faulting at every instruction is going to make things ridiculously slow, and besides, it won’t help you if somebody performs a block memory copy between two different “pseudo address spaces” that happen to have the same linear address. I guess at that point, you would change the selector base addresses so that the source and destination no longer land on the same page, but at this point you are doing so much work at every instruction that you may as well give up trying to execute code natively and just write a p-code interpreter.

Raymond Chen

Follow