# Why is address space allocation granularity 64KB?

devblogs.microsoft.com/oldnewthing/20031008-00

October 8, 2003

Raymond Chen

You may have wondered why VirtualAlloc allocates memory at 64KB boundaries even though page granularity is 4KB.

You have RISC processors like the Alpha AXP to thank for that.

RISC processors typically lack a "load 32-bit integer immediate" instruction. To load a 32-bit integer, you actually load two 16-bit integers and combine them.

So if allocation granularity were finer than 64KB, a DLL that got relocated in memory would require two fixups per relocatable address: one to the upper 16 bits and one to the lower 16 bits. And things get worse if this changes a carry or borrow between the two halves. (For example, moving an address 4KB from 0x1234F000 to 0x12350000, this forces both the low and high parts of the address to change. Even though the amount of motion was far less than 64KB, it still had an impact on the high part due to the carry.)

But wait, there's more.

The Alpha AXP actually combines two *signed* 16-bit integers to form a 32-bit integer. For example, to load the value 0x1234ABCD, you would first use the LDAH instruction to load the value 0x1235 into the high word of the destination register. Then you would use the LDA instruction to add the signed value -0x5433. (Since 0x5433 = 0x10000 – 0xABCD.) The result is then the desired value of 0x1234ABCD.

```
LDAH t1, 0x1235(zero) // t1 = 0x12350000
LDA  t1, -0x5433(t1)  // t1 = t1 - 0x5433 = 0x1234ABCD
```

So if a relocation caused an address to move between the "lower half" of a 64KB block and the "upper half", additional fixing-up would have to be done to ensure that the arithmetic for the top half of the address was adjusted properly. Since compilers like to reorder instructions, that LDAH instruction could be far, far away, so the relocation record for the bottom half would have to have some way of finding the matching top half.

What's more, the compiler is clever and if it needs to compute addresses for two variables that are in the same 64KB region, it shares the LDAH instruction between them. If it were possible to relocate by a value that wasn't a multiple of 64KB, then the compiler would no longer be able to do this optimization since it's possible that after the relocation, the two variables no longer belonged to the same 64KB block.

Forcing memory allocations at 64KB granularity solves all these problems.

If you have been paying really close attention, you'd have seen that this also explains why there is a 64KB "no man's land" near the 2GB boundary. Consider the method for computing the value 0x7FFFABCD: Since the lower 16 bits are in the upper half of the 64KB range, the value needs to be computed by subtraction rather than addition. The naïve solution would be to use

```
LDAH t1, 0x8000(zero) // t1 = 0x80000000, right?
LDA  t1, -0x5433(t1)  // t1 = t1 - 0x5433 = 0x7FFFABCD, right?
```

Except that this doesn't work. The Alpha AXP is a 64-bit processor, and 0x8000 does not fit in a 16-bit signed integer, so you have to use -0x8000, a negative number. What actually happens is

```
LDAH t1, -0x8000(zero) // t1 = 0xFFFFFFFF`80000000
LDA  t1, -0x5433(t1)   // t1 = t1 - 0x5433 = 0xFFFFFFFF`7FFFABCD
```

You need to add a third instruction to clear the high 32 bits. The clever trick for this is to add zero and tell the processor to treat the result as a 32-bit integer and sign-extend it to 64 bits.

```
ADDL t1, zero, t1    // t1 = t1 + 0, with L suffix
// L suffix means sign extend result from 32 bits to 64
                     // t1 = 0x00000000`7FFFABCD
```

If addresses within 64KB of the 2GB boundary were permitted, then every memory address computation would have to insert that third ADDL instruction just in case the address got relocated to the "danger zone" near the 2GB boundary.

This was an awfully high price to pay to get access to that last 64KB of address space (a 50% performance penalty for all address computations to protect against a case that in practice would never happen), so roping off that area as permanently invalid was a more prudent choice.

Raymond Chen

**Follow**