32-Bit vs. 64-Bit

I remember being asked in philosophy class, "If the world and everything in it suddenly doubled in size, would you notice?" This was a thought experiment with no real answer, and no real implication; it was intended to help you think critically.

With software development, this takes on new significance. This essentially does happen and, in this particular case, we *do* notice. The first, most obvious difference is the effect on memory resource limits. A 32-bit application is capped (without going into deeper details) at 2 GB of Virtual Address Space, whereas a 64-bit application is capped at 8 TB. This is a substantial difference, and sounds like a limit that we are not likely to reach any time soon, but bear in mind that it wasn't too long ago that 2 GB sounded like a ridiculous amount of memory.

Of course, not everything has doubled in size, which is the source of most interoperability issues, and one of the reasons why the change up to 64-bit systems is so noticeable. Pointers and references have doubled in size; other data types have not.

Let's start with the basics and then work our way in. Any given program compiled as a 64-bit application will use more memory than the 32-bit version of the same program. In fact, you will use more than twice the memory, and there are a couple of reasons for this. As we have just seen, the size of a reference has doubled, and the size of a pointer has doubled, which means that, among other things, the metadata for any given object has doubled. On a 32-bit system, each object includes an 8-byte header; on a 64-bit system, these objects will include a 16-byte header.

The size of the header is important, because it is this that led to the recommendations for the smallest possible size of an object. It also means that, if the actual data in a class takes up less space than this header, then that class should be a struct, instead. Otherwise, it will feel like your program is thrashing when your objects use more memory for the *header* than for the actual data. So, because not everything doubles in size when switching from 32-bit architecture to 64-bit architecture, you may want to switch some objects from classes to structs to make the most effective use of memory.

However, bear in mind that this tactic may backfire if you are forced to box and unbox the new structs. Generics will help avoid boxing (as mentioned in Chapter 4, *Common Memory Problems*), but you must be careful. Also, remember the guideline that structs must be immutable; if you cannot follow the guidelines for structs, or if these changes obscure your underlying business logic, then you should carefully weigh the pros and cons for your application.

There are naturally other factors that will lead to your application using more memory; for starters, the actual size of your modules will be increased. For example, `mscorwks.dll` is about 5 MB on x86, 10 MB on x64, and 20 MB on ia64. Also, the CLR itself becomes less frugal with memory than it was with 32 bits (because it can afford to be). Instead, more attention is paid to other optimizations that will have more of an impact. You can clearly see this by investigating the segment size. If you run this `windb` command on a memory dump: `! SOS.eeheap –gc`, you will notice dramatically larger segments (if you're not familiar with it, this command summarizes heap sizes). Don't worry about how much bigger the segments are - this is just an exercise to demonstrate my point. Microsoft is rather tight-lipped on the actual size, and you should not make any design decisions based on what you see, since they reserve the right to change it in the future.

Another point to bear in mind - the Large Object Heap does not change. The size threshold is the same for 64 bits as it was for 32 bits and, since your objects will be bigger in a 64-bit environment, this means that you may have objects winding up in the LOH that were normally placed in the SOH when everything was 32 bit.

The CLR takes advantage of the larger memory limits by raising the thresholds that will trigger garbage collections which, in turn, means that your application will go through fewer garbage collection cycles, although each cycle may take longer. As a result, more memory may be tied up in collectable objects before the system gets around to collecting them.

All of this means that your memory footprint will go up much more than you might expect and, instead of getting an `OutOfMemoryException`, your application may run along happily until all of your system resources are used.

There are a few actions that will be faster in a 64-bit application but, in general, most tasks will actually be *slower.* This is because, while manipulating memory in QWORDS (64-bit blocks, or quad-words) instead of DWORDS (32-bit blocks, or double-words) is faster, this difference is generally absorbed by new costs. The real advantage you get is in having more memory to work with. As mentioned a moment ago, this will naturally lead to fewer garbage collections. Fewer garbage collections will then mean that your code will not be interrupted and brought to safe points as often, which is an immense performance benefit.