There's also a large object heap for unmanaged code, but it's inside the regular heap

devblogs.microsoft.com/oldnewthing/20110930-00

September 30, 2011

Raymond Chen

Occasionally, a customer will ask for assistance explaining some strange heap behavior, or at least heap behavior that appears to be strange if you assume that the heap behaves purely classically.

> I need to understand the heap behavior we're seeing. I have a sample program which allocates five blocks of memory from the process heap, each of size 100 bytes. When we dump the heap blocks with the `!heap -x` command, we find that all of them belong to the same heap segment, and when we do a `!vadump -v`, we find that they all live on the same page. On the other hand, if we allocate five blocks of size 512KB, then we find that each one gets placed in its own virtually-allocated range, and the range is not size 512KB but rather 516KB.
>
> Questions:
>
> 1. Why does the system combine the 100-byte allocations? Why is there no extra overhead for the small allocations, but the large allocations have a 4KB overhead?
> 2. Why does the system allocate a separate virtual address range for the 512KB block? Which component makes this decision and what are the factors involved?
> 3. Why does allocating a 512KB block require 4KB of overhead?
> 4. Is there any documentation for this behavior?

Let's look at the last question first. There is no documentation for this behavior because it is all implementation detail. All the heap is required to do is return you a block of memory of the size you requested. How it goes about getting this memory block is at the discretion of the heap manager, and as we've seen in the past, the precise algorithm has varied over time. The claim that there is no extra overhead for the small allocations is incorrect. There is still a small amount of overhead, but you can't see it because the `!heap -x` command doesn't report it. There is still overhead in the heap to keep track of the memory block's actual size, the fact that the memory block is allocated (and not free), and other overhead-type things. Indeed, even for large blocks, you didn't see the overhead reported by the `!heap -x` command; you had to drop below the heap and use the `!vadump` command to see it. (And how do you know that the other 3.9KB are being lost to overhead? Maybe they are being held

for a future 3.9KB allocation?) Okay, so fine, small blocks have overhead, but why do larger blocks have significantly higher overhead? The answer to this is in the second question: The system allocates large blocks in a separate virtual address range because the default process heap treats large memory blocks differently from small memory blocks, in the same way that the CLR treats large and small objects differently. You might say that the unmanaged heap also has a large-object sub-heap. When objects get large, the heap switches to `VirtualAlloc`. You can think of it as creating a custom segment just for that object. And since `VirtualAlloc` allocates memory in integer multiples of the page size, any nonzero overhead will result in a full extra page being allocated since the requested allocation size was itself already an integral multiple of the page size. Mathematically:

> $roundup(n \times$ `PageSize` $+ v,$ `PageSize` $) = n \times$ `PageSize` $+ roundup(v,$ `PageSize` $).$

Therefore, even if a heap allocation has only one byte of overhead, you will have to pay a full page for it due to rounding. The precise point at which the heap will switch to `VirtualAlloc` has changed over time, so don't rely on it. In Windows 95, the switchover point was around 4MB, but Windows NT set the cutover to something close to 512KB. If you're interested in details of the internal heap bookkeeping, you can check out *Advanced Windows Debugging* or *Windows Internals*. Note that information about the heap implementation is to be used for diagnostic and educational purposes only. Do not write code that takes advantage of the internals of the heap architecture, because those details change all the time. The purpose of a heap is to reduce memory and address space overhead compared to direct allocation via methods like `VirtualAlloc` by combining multiple small allocations into a single 64KB block, at a cost of some overhead per item. Although there is a small amount of overhead per item, the overall savings makes it a win. If you need eight hundred 100-byte chunks of memory, and you didn't have a heap manager, you would allocate eight hundred 64KB blocks (since `VirtualAlloc` allocates address space in 64KB chunks) and commit the first page in each, for a total memory usage of around 3MB and address space usage of around 50MB. Even if the heap overhead were a ridiculous 100%, the one thousand allocations would fit into forty 4KB pages, reducing the memory usage to 160KB and the address space usage to 192KB, a massive savings. (Looking at things in terms of relative overhead, 4KB out of a 512KB allocation is still less than one percent. You wish your index fund had such a low overhead!) The advantage of chunking allocations together diminishes as the size of the allocations increase. By the time you reach 512KB, the heap is not really buying you much savings, since you're already buying in bulk. In fact, the switchover to direct `VirtualAlloc` is an admission by the heap manager that the allocation size has become so large that it is starting to make the overall heap degrade. If the 512KB allocation were made in the classic heap, it would probably not start on an exact page boundary; when you went to free the memory, there would be little fragments left over at the start and end which could not be decommitted and which would end up as little committed pages scattered about fragmenting your address space. (Besides, I don't think the heap decommits partial segments anyway.) If you know ahead of time that your allocation is large,

and you control both the code which allocates the memory and the code which frees the memory, you can switch to `VirtualAlloc` directly and avoid burdening the heap with very large allocations. (On the other hand, the heap does this cutover automatically, so perhaps you're better off just letting the heap designers decide how big is "too big.")

Bonus chatter: And no, the heap manager will never ask `VirtualAlloc` for large pages.

Raymond Chen

Follow