A short puzzle about heap expansion

devblogs.microsoft.com/oldnewthing/20100429-00

April 29, 2010

Raymond Chen

At the 2008 PDC, somebody stopped by the *Ask the Experts* table with a question about the heap manager.

> I don't understand why the heap manager is allocating a new segment. I allocated a bunch of small blocks, then freed nearly all of them. And then when my program makes a large allocation, it allocates a new segment instead of reusing the memory I had just freed.

Under the classical model of the heap, the heap manager allocates a large chunk of memory from lower-level operating system services, and then when requests for memory come in from the application, it carves blocks of memory from the big chunk and gives them to the application. (These blocks are called *busy*.) When those blocks of memory are freed, they are returned to the pool of available memory, and if there are two blocks of free memory adjacent to each other, they are combined (*coalesced*) to form a single larger block. That way, the block can be used to satisfy a larger allocation in the future. Under the classical model, allocating memory and then freeing it is a net no-operation. (Nitpicky details notwithstanding.) The allocation carves the memory out of the big slab of memory, and the free returns it to the slab. Therefore, the situation described above is a bit puzzling. After the memory is freed back to the heap, the little blocks should coalesce back into a block big enough to hold a larger allocation. I sat and wondered for a moment, trying to think of cases where coalescing might fail, like if they happened to leave an allocated block right in the middle of the chunk. Or maybe there's some non-classical behavior going on. For example, maybe the look-aside list was keeping those blocks live. As I considered the options, the person expressed disbelief in a different but telling way:

> You'd think the low-fragmentation heap (LFH) would specifically avoid this problem.

Oh wait, you're using the low-fragmentation heap! This is a decidedly non-classical heap implementation: Instead of coalescing free blocks, it keeps the free blocks distinct. The idea of the low-fragmentation heap is to reduce the likelihood of various classes of heap fragmentation problems:

- You want to make a large allocation, and you almost found it, except that there's a small allocation in the middle of your large block that is in your way.
- You have a lot of free memory, but it's all in the form of teeny tiny useless blocks.

That first case is similar to what I had been considering: where you allocated a lot of memory, free most of it, but leave little islands behind. The second case occurs when you have a free block of size *N*, and somebody allocates a block of size *M* < *N*. The heap manager breaks the large block into two smaller blocks: a busy block of size *M* and a free block of size (*N* – *M*). These "leftover" free blocks aren't a problem if your program later requests a block of size *N* – *M*: The leftover block can be used to satisfy the allocation, and no memory goes wasted. But if your program never asks for a block of size *N* – *M*, then the block just hangs around as one of those useless blocks. Imagine, for concreteness, a program that allocates memory in a loop like this:

- `p1 = alloc(128)`
- `p2 = alloc(128)`
- `free(p1)`
- `p3 = alloc(96)`
- (Keep `p2` and `p3` allocated.)
- Repeat

Under the classical model, when the request for 96 bytes comes in, the memory manager sees that 128-byte block (formerly known as `p1`) and splits it into two parts, a 96-byte block and a 32-byte block. The 96-byte block becomes block `p3`, and the 32-byte block sits around waiting for somebody to ask for 32 bytes (which never happens). Each time through this loop, the heap grows by 256 bytes. Of those 256 bytes, 224 are performing useful work in the application, and 32 bytes are sitting around being one of those useless tiny memory allocations which contributes to fragmentation. The low-fragmentation heap tries to avoid this problem by keeping similar-sized allocations together. A heap block only gets re-used for the same size allocation it was originally created for. (This description is simplified for the purpose of the discussion.) (I can't believe I had to write that.) In the above scenario, the low-fragmentation heap would respond to the request to allocate 96 bytes not by taking the recently-freed 128-byte block and splitting it up, but rather by making a brand new 96-byte allocation. This seems wasteful. After all, you now allocated 128 + 128 + 96 = 352 bytes even though the application requested only 128 + 96 = 224 bytes. (The classical heap would have re-used the first 96 bytes of the second 128-byte block, for a total allocation of 128 + 128 = 256 bytes.) This seemingly wasteful use of memory is really an investment in the future. (I need to remember to use that excuse more. "No, I'm not being wasteful. I'm just investing in the future.") The investment pays off at the next loop iteration: When the request for 128 bytes comes in, the heap manager can return the 128-byte block that was freed by the previous iteration. Now there is no waste in the heap at all! Suppose the above loop runs 1000 times. A classical heap would end up with a thousand 128-byte allocations, a thousand 96-byte allocations, and a thousand 32-byte free blocks on the heap. That's 31KB of memory in the heap lost to fragmentation, or about 12%. On the other hand, the low-fragmentation heap would end up with a thousand 128-byte allocations, a thousand 96-byte allocations, and one 128-byte free block. Only 128 bytes has been lost to fragmentation, or just 0.06%. Of course, I exaggerated this scenario in order to make the low-fragmentation heap look particularly good. The low-fragmentation heap operates well when heap allocation sizes tend to repeat, because the repeated-size allocation will re-use a freed allocation of the same size. It operates poorly when you allocate blocks of a certain size, free them, then never ask for blocks of that size again (since those blocks just sit around waiting for their chance to shine, which never comes). Fortunately, most applications don't fall into this latter category: Allocations tend to be for a set of fixed sizes (fixed-size objects), and even allocations for variable-sized objects tend to cluster around a few popular sizes. Generally speaking, the low-fragmentation heap works pretty well for most classes of applications, and you should consider using it. (In fact, I'm told that the C runtime libraries have converted the default C runtime heap to be a low-fragmentation heap starting in Visual Studio 2010.) On the other hand, it's also good to know a little of how the low-fragmentation heap operates, so that you won't be caught out by its non-classical behavior. For example, you should now be able to answer the question which was posed at *Ask the Experts*. As you can see, it often doesn't take much to be an expert. You can do it, too.

Sidebar: Actually, I was able to answer the customer's question even without knowing anything about the low-fragmentation heap prior to the customer mentioning it. (Indeed, I had barely even heard of it until that point.) Just given the name *low-fragmentation heap*, I was able to figure out roughly how such a beast would have operated. I wasn't correct on the details, but the underlying principles were good. So you see, you don't even have to know what you're talking about to be an expert. You just have to be able to take a scenario and think, "How would somebody have designed a system to solve this problem?"

Raymond Chen

Follow