GC Segments and Virtual Memory

In our discussion of the basic GC model and the generational GC model, we have repeatedly assumed that a .NET process usurps the entire memory space available for its hosting process and uses this memory as the backing store for the garbage collected heap. This assumption is blatantly wrong, considering the fact that managed applications can't survive in isolation without using unmanaged code. The CLR itself is implemented in unmanaged code, the .NET base class library (BCL) often wraps Win32 and COM interfaces, and custom components developed in unmanaged code can be loaded into the otherwise "managed" process.

Note Even if managed code could live in total isolation, it does not make sense for the garbage collector to immediately commit the entire available address space. Although committing memory without using it does not imply the backing store (RAM or disk space) for that memory is immediately required, it is not an operation that comes for free. It is generally advisable to allocate only slightly more than the amount of memory that you are likely to need. As we will see later, the CLR reserves significant memory regions in advance, but commits them only when necessary and makes a point of returning unused memory to Windows.

In view of the above, we must refine the garbage collector's interaction with the virtual memory manager. During CLR startup within a process, two blocks of memory called *GC segments* are allocated from virtual memory (more precisely, the CLR host is requested to provide this block of memory). The first segment is used for generation 0, generation 1 and generation 2 (called the *ephemeral segment*). The second segment is used for the large object heap. The size of the segment depends on the GC flavor and on GC startup limits if running under a CLR host. A typical segment size on 32-bit systems with workstation GC is 16MB, and for server GC it is in the 16-64MB range. On 64-bit systems the CLR uses 128MB-2 GB segments with server GC, and 128MB-256MB segments with workstation GC. (The CLR does not commit an entire segment at a time; it only reserves the address range and commits parts of the segment as the need arises.)

When the segment becomes full and more allocation requests arrive, the CLR allocates another segment. Only one segment can contain generation 0 and generation 1 at any given time. However, it does not have to be the same segment! We have previously observed that pinning objects in generation 0 or generation 1 for a long time is especially dangerous due to fragmentation effects in these small memory regions. The CLR handles these issues by declaring another segment as the ephemeral segment, which effectively promotes the objects previously in the younger generations straight into generation 2 (because there can be only one ephemeral segment).

Figure 4-16. *GC segments occupy the virtual address space of the process. The segment containing the young generations is called the ephemeral segment*

When a segment becomes empty as the result of a garbage collection, the CLR usually releases the segment's memory and returns it to the operating system. This is the desired behavior for most applications, especially applications that exhibit infrequent large spikes in memory usage. However, it is possible to instruct the CLR to retain empty segments on a standby list without returning them to the operating system. This behavior is called *segment hoarding* or *VM hoarding*, and can be enabled through CLR hosting with the startup flags of the CorBindToRuntimeEx function. Segment hoarding can improve performance in applications which allocate and release segments very frequently (memory-intensive applications with frequent spikes in memory usage), and reduce out of memory exceptions due to virtual memory fragmentation (to be discussed shortly). It is used by default by ASP.NET applications. A custom CLR host can further customize this behavior by satisfying segment allocation requests from a memory pool or any other source by using the IHostMemoryManager interface.

The segmented model of managed memory space introduces an extremely serious problem which has to with external (virtual memory) fragmentation. Because segments are returned to the operating system when they are empty, an unmanaged allocation can occur in the middle of a memory region that was once a GC segment. This allocation causes fragmentation because segments must be consecutive in memory.

The most common causes of virtual memory fragmentation are late-loaded dynamic assemblies (such as XML serialization assemblies or ASP.NET debug compiled pages), dynamically loaded COM objects and unmanaged code performing scattered memory allocations. Virtual memory fragmentation can cause out of memory conditions even when the perceived memory usage of the process is nowhere near the 2 GB limit. Long-running processes such as web servers that perform memory-intensive work with frequent spikes in memory allocation tend to exhibit this behavior after several hours, days or weeks of execution. In non-critical scenarios where a failover is available (such as a load-balanced server farm), this is normally addressed by process recycling.

Note Theoretically speaking, if the segment size is 64MB, the virtual memory allocation granularity is 4KB and the 32-bit process address space is 2GB (in which there is room for only 32 segments), it is possible to allocate only 4KB × 32 = 128KB of unmanaged memory and still fragment the address space such that not a single consecutive segment can be allocated!

The Sysinternals VMMap utility can diagnose memory fragmentation scenarios quite easily. Not only does it report precisely which region of memory is used for which purposes, it has also a Fragmentation View option that displays a picture of the address space and makes it easy to visualize fragmentation problems. Figure 4-17 shows an example of an address space snapshot in which there are almost 500MB of free space, but no single free fragment large enough for a 16MB GC segment. In the screenshot, white areas are free memory regions.

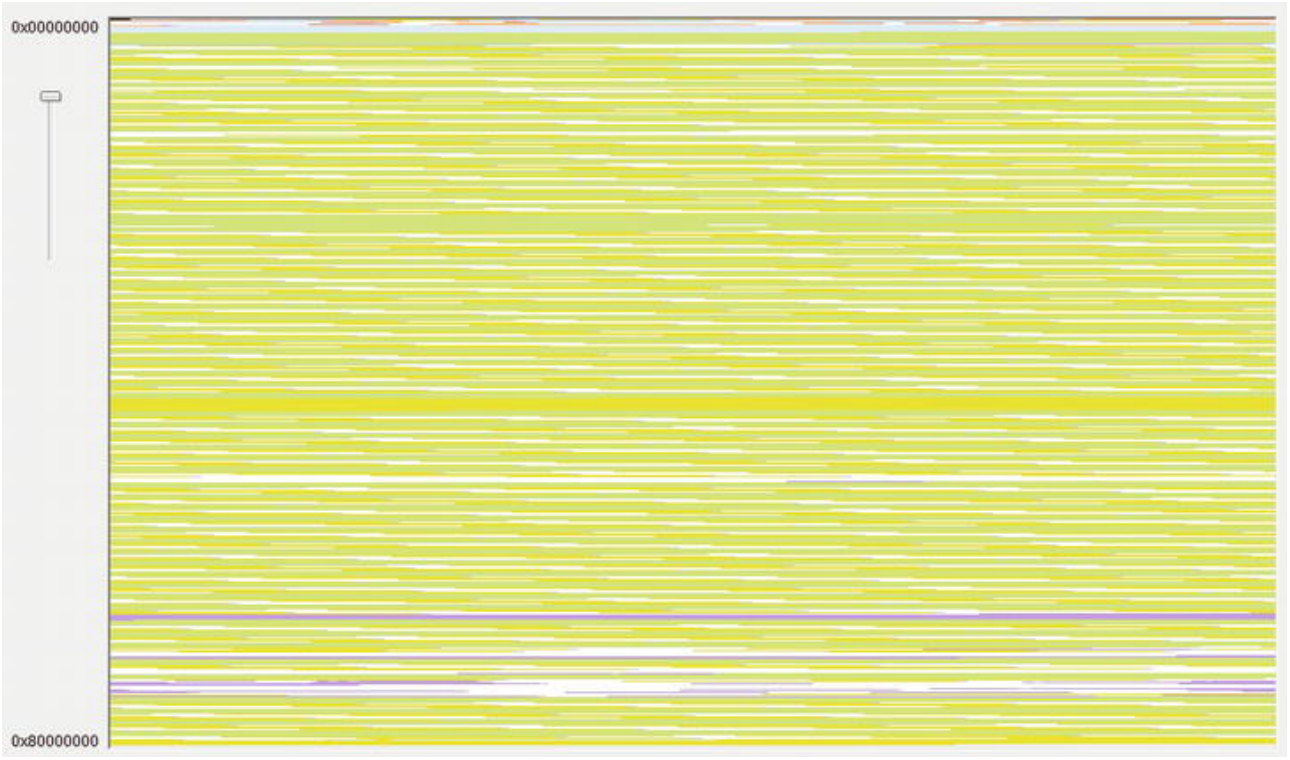

Figure 4-17. A badly fragmented address space snapshot. There are almost 500MB of available memory, but no chunk large enough to fit a GC segment

EXPERIMENTING WITH VMMAP

You can try using VMMap with a sample application and see how quickly it points you in the right direction. Specifically, VMMap makes it very easy to determine whether the memory problems you're experiencing originate within the managed heap or elsewhere, and facilitates the diagnostics of fragmentation problems.

1. Download VMMap from Microsoft TechNet (http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx) and store it on your machine.
2. Run the OOM2.exe application from this chapter's sample code folder. The application quickly exhausts all available memory and crashes with an exception. When the Windows error reporting dialog appears, do not dismiss it—keep the application running.
3. Run VMMap and choose the OOM2.exe process to monitor. Note the amount of available memory (the "Free" row in the summary table). Open the Fragmentation View (from the View menu) to examine the address space visually. Inspect the detailed list of memory blocks for the "Free" type to see the largest free block in the application's address space.
4. As you can see, the application does not exhaust all the virtual address space, but there is not enough room to allocate a new GC segment—the largest available free block is smaller than 16 MB.
5. Repeat steps 2 and 3 with the OOM3.exe application from this chapter's sample code folder. The memory exhaustion is somewhat slower, and occurs for a different reason.

Whenever you stumble upon a memory-related problem in a Windows application, VMMap should always be ready on your toolbelt. It can point you to managed memory leaks, heap fragmentation, excessive assembly loading, and many other problems. It can even profile unmanaged allocations, helping to detect memory leaks: see Sasha Goldshtein's blog post on VMMap allocation profiling (http://blog.sashag.net/archive/2011/08/27/vmmap-allocation-profiling-and-leak-detection.aspx).

There are two approaches for addressing the virtual memory fragmentation problem:

- Reducing the number of dynamic assemblies, reducing or pooling unmanaged memory allocations, pooling managed memory allocations or hoarding GC segments. This category of approaches typically extends the amount of time until the problem resurfaces, but does not eliminate it completely.
- Switching to a 64-bit operating system and running the code in a 64-bit process. A 64-bit process has 8TB of address space, which for all practical purposes eliminates the problem completely. Because the problem is not related to the amount of physical memory available, but rather strictly to the amount of virtual address space, switching to 64-bit is sufficient regardless of the amount of physical memory.

This concludes the examination of the segmented GC model, which defines the interaction between managed memory and its underlying virtual memory store. Most applications will never require customizing this interaction; should you be forced to do so in the specific scenarios outlined above, CLR hosting provides the most complete and customizable solution.