
Virtual and Physical Memory

Physical memory is simply the amount of physical RAM installed on the machine. By definition, it has a fixed size that can only be increased by opening the back of the machine and slotting more inside.

Any resource with a fixed limit causes a problem; once that limit is reached – bang – there is nowhere to go. In the absence of any other kind of memory, once the physical memory is all used up, things start to go horribly wrong.

Virtual memory was devised as a way around the problem: using a machine's hard drive as a kind of alternative memory store of last resort. Physical disk capacity is much greater than RAM capacity, and so it makes sense to use part of your disk storage as a way of increasing the amount of available memory. To facilitate this, a large file is created, often called a “swap” or “page” file, which is used to store physical memory sections that haven't been accessed for a while. This allows the memory manager (MM) to reuse physical memory and still have access to “paged” data when requested. I probably don't need to mention the downside of this cunning solution, but I will, anyway: **accessing bits from RAM is pretty fast; getting them off a disk is really slow.**

Virtual memory uses a combination of physical, RAM-based memory and disk-based memory to create a uniform virtual memory address space. Combining the two resources into a single pool gives applications access to a far larger and, as far as processes are concerned, homogenous memory space. In Figure 7.1 you can see a Virtual Address Space (VAS) made up of a sequence of memory blocks, or pages, which point to either physical or disk-based memory.

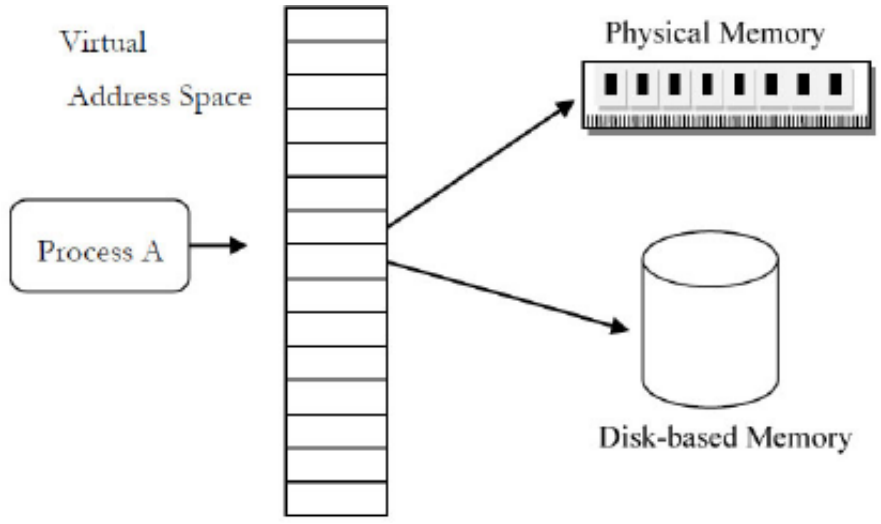

Figure 7.1: Process Virtual Address Space.

Pages

The virtual memory address space is organized into chunks, or pages, and there are two types of pages (you can specify page size in `VirtualAlloc`):

- small (4 KB)
- large (16 MB).

When a chunk of memory is requested by the .NET runtime, a set of virtual pages, sufficiently large to accommodate the requested size, are allocated. If you asked for 9 K, you would get three pages at 4 K each, that is, 12 K in total.

The process address space

Figure 7.1 illustrates how process A has access to its own Virtual Address Space (VAS). When process A reads an address from the VAS, that address's actual location, be it in physical memory or on disk, is determined and then translated to the physical address by the operating system's Virtual Memory Manager (VMM), which takes care of all retrieval and storage of memory, as required. We'll take a closer look at the VMM shortly.

The possible size of the memory space (often called address space) depends on the version of Windows running, and whether it's a 32- or a 64-bit system.

32-bit address space

Each 32-bit Windows OS process has a maximum address space of 4 GB, which is calculated as 2^{32}. This is split between a private space of 2 GB for each process, and a space of 2 GB for the OS's dedicated use, and for any shared use (things like DLLs, which use up address space in the process, but aren't counted in the working set or private bytes end up in here; we'll come to this later).

Note

It is possible for a 32-bit process to access up to 3 GB, if it's compiled as Large Address aware and the OS is booted with the correct option.

64-bit address space

For 64-bit Windows processes, the available address space depends on the processor architecture. It would be *theoretically* possible for a 64-bit system to address up to 18 exabytes (264). However, in reality, current 64-bit processors use 44 bits for addressing virtual memory, allowing for 16 terabytes (TB) of memory, which is equally split between user mode (application processes) and kernel mode (OS processes and drivers). 64-bit Windows applications therefore have a private space of up to 8 TB.

What decides which bits within the VAS are stored on disk or in RAM? The answer is “the memory manager.”