Memory Manager

With the exception of kernel mode processes, which can access memory directly, all other memory addresses are virtual addresses, which means that when a thread accesses memory, the CPU has to determine where the data is actually located. As we know, it can be in one of two places:

- physical memory
- disk (inside a page file).

If the data is already in memory, then the CPU just translates the virtual address to the physical memory address of the data, and access is achieved. On the other hand, if the data is on the disk, then a "page fault" occurs, and the memory manager has to load the data *from* the disk into a physical memory location. Only then can it translate virtual and physical addresses (we'll look at how data is moved back out to pages when we discuss page faults in more detail, later in this chapter).

Now, when a thread needs to allocate a memory page (see the *Pages* section, above), it makes a request to the MM, which allocates virtual pages (using `VirtualAlloc`) and also manages when the physical memory pages are actually created.

To free up memory, the MM just frees the physical and disk-based memory, and marks the virtual address range as free.

Processes and developers are completely unaware of what's going on under the hood. Their memory accesses just work, even though they have been translated, and may first have been loaded from disk.

Using the memory manager

As mentioned earlier, when you request memory using `VirtualAlloc`, the entire process is mediated by the memory manager, and you have three choices available to you. You can:

- **reserve** the virtual memory address range for future use (fast, and ensures the memory will be available, but requires a later commit)
- **claim** it immediately (slower, as physical space has to be found and allocated)
- **commit** previously reserved memory.

Claiming your memory immediately is called "committing," and means that whatever you have committed will be allocated in the page file, but will only make it into physical RAM memory when first used. On the other hand, "reserving" just means that the memory is available for use at some point in the future, but isn't yet associated with any physical storage. Later on, you can commit portions of the reserved virtual memory, but reserving it first is a faster process in the short term, and means that the necessary memory is definitely available to use when you commit it later on (not to mention faster to commit).

Keeping track

To keep track of what has been allocated in a process's VAS, the memory manager maintains a structure called the **Virtual Address Descriptor (VAD)** tree.

Each VAD entry in the tree contains data about the virtual allocation including:

- start address
- end address
- committed size (0 if reserved)
- protection info (Read, Write, etc.), which is actually outside the scope of this book.

If you look at the parameters of `VirtualAlloc` on the Microsoft developer network at HTTP://T1NYURL.COM/VIRTUALALLOC, you can see how some of its parameters are used to build the VAD.

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect);
```

Listing 7.1: `VirtualAlloc` function prototype.

`VirtualAlloc` (Listing 7.1) takes the following parameters:

- `lpAddress` –virtual address
- size of allocation
- `flAllocationType` includes values:
  - `MEM_COMMIT`
  - `MEM_RESERVE`
- `flProtect` includes values:
  - `PAGE_READWRITE`
  - `PAGE_READ.`

So, the VAS state is entirely held within the VAD tree, and this is the starting point for virtual memory management. Any attempt to access virtual memory (read or write) is first checked to ensure access is being made to an existing virtual address, and only then is an attempt made to translate addresses.