4.4. Virtual Memory: The Details

access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Introduction_to_System_Administration/s1-memory-virt-details.html

First, we must introduce a new concept: *virtual address space*. Virtual address space is the maximum amount of address space available to an application. The virtual address space varies according to the system's architecture and operating system. Virtual address space depends on the architecture because it is the architecture that defines how many bits are available for addressing purposes. Virtual address space also depends on the operating system because the manner in which the operating system was implemented may introduce additional limits over and above those imposed by the architecture.

The word "virtual" in virtual address space means this is the total number of uniquely-addressable memory locations available to an application, but *not* the amount of physical memory either installed in the system, or dedicated to the application at any given time.

In the case of our example application, its virtual address space is 15000 bytes.

To implement virtual memory, it is necessary for the computer system to have special memory management hardware. This hardware is often known as an *MMU* (Memory Management Unit). Without an MMU, when the CPU accesses RAM, the actual RAM locations never change — memory address 123 is always the same physical location within RAM.

However, with an MMU, memory addresses go through a translation step prior to each memory access. This means that memory address 123 might be directed to physical address 82043 at one time, and physical address 20468 another time. As it turns out, the overhead of individually tracking the virtual to physical translations for billions of bytes of memory would be too great. Instead, the MMU divides RAM into *pages* — contiguous sections of memory of a set size that are handled by the MMU as single entities.

Keeping track of these pages and their address translations might sound like an unnecessary and confusing additional step, but it is, in fact, crucial to implementing virtual memory. For the reason why, consider the following point.

Taking our hypothetical application with the 15000 byte virtual address space, assume that the application's first instruction accesses data stored at address 12374. However, also assume that our computer only has 12288 bytes of physical RAM. What happens when the CPU attempts to access address 12374?

What happens is known as a *page fault*.

4.4.1. Page Faults

A page fault is the sequence of events occurring when a program attempts to access data (or code) that is in its address space, but is not currently located in the system's RAM. The operating system must handle page faults by somehow making the accessed data memory resident, allowing the program to continue operation as if the page fault had never occurred.

In the case of our hypothetical application, the CPU first presents the desired address (12374) to the MMU. However, the MMU has no translation for this address. So, it interrupts the CPU and causes software, known as a page fault handler, to be executed. The page fault handler then determines what must be done to resolve this page fault. It can:

- Find where the desired page resides on disk and read it in (this is normally the case if the page fault is for a page of code)
- Determine that the desired page is already in RAM (but not allocated to the current process) and

reconfigure the MMU to point to it

- Point to a special page containing nothing but zeros and later allocate a page only if the page is ever written to (this is called a *copy on write* page, and is often used for pages containing zero-initialized data)
- Get it from somewhere else (which is discussed in more detail later)

While the first three actions are relatively straightforward, the last one is not. For that, we need to cover some additional topics.

4.4.2. The Working Set

The group of physical memory pages currently dedicated to a specific process is known as the *working set* for that process. The number of pages in the working set can grow and shrink, depending on the overall availability of pages on a system-wide basis.

The working set grows as a process page faults. The working set shrinks as fewer and fewer free pages exist. To keep from running out of memory completely, pages must be removed from process's working sets and turned into free pages, available for later use. The operating system shrinks processes' working sets by:

- Writing modified pages to a dedicated area on a mass storage device (usually known as *swapping* or *paging* space)
- Marking unmodified pages as being free (there is no need to write these pages out to disk as they have not changed)

To determine appropriate working sets for all processes, the operating system must track usage information for all pages. In this way, the operating system determines which pages are actively being used (and must remain memory resident) and which pages are not (and therefore, can be removed from memory.) In most cases, some sort of least-recently used algorithm determines which pages are eligible for removal from process working sets.

4.4.3. Swapping

While swapping (writing modified pages out to the system swap space) is a normal part of a system's operation, it is possible to experience too much swapping. The reason to be wary of excessive swapping is that the following situation can easily occur, over and over again:

- Pages from a process are swapped
- The process becomes runnable and attempts to access a swapped page
- The page is faulted back into memory (most likely forcing some other processes' pages to be swapped out)
- A short time later, the page is swapped out again

If this sequence of events is widespread, it is known as *thrashing* and is indicative of insufficient RAM for the present workload. Thrashing is extremely detrimental to system performance, as the CPU and I/O loads that can be generated in such a situation quickly outweigh the load imposed by a system's real work. In extreme cases, the system may actually do no useful work, spending all its resources moving pages to and from memory.