4.3. Basic Virtual Memory Concepts

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

While the technology behind the construction of the various modern-day storage technologies is truly impressive, the average system administrator does not need to be aware of the details. In fact, there is really only one fact that system administrators should always keep in mind:

There is never enough RAM.

While this truism might at first seem humorous, many operating system designers have spent a great deal of time trying to reduce the impact of this very real shortage. They have done so by implementing *virtual memory* — a way of combining RAM with slower storage to give a system the appearance of having more RAM than is actually installed.

4.3.1. Virtual Memory in Simple Terms

Let us start with a hypothetical application. The machine code making up this application is 10000 bytes in size. It also requires another 5000 bytes for data storage and I/O buffers. This means that, to run this application, there must be 15000 bytes of RAM available; even one byte less, and the application would not be able to run.

This 15000 byte requirement is known as the application's *address space*. It is the number of unique addresses needed to hold both the application and its data. In the first computers, the amount of available RAM had to be greater than the address space of the largest application to be run; otherwise, the application would fail with an "out of memory" error.

A later approach known as *overlaying* attempted to alleviate the problem by allowing programmers to dictate which parts of their application needed to be memory-resident at any given time. In this way, code only required once for initialization purposes could be written over (overlayed) with code that would be used later. While overlays did ease memory shortages, it was a very complex and error-prone process. Overlays also failed to address the issue of system-wide memory shortages at runtime. In other words, an overlayed program may require less memory to run than a program that is not overlayed, but if the system still does not have sufficient memory for the overlayed program, the end result is the same — an out of memory error.

Virtual memory turns the concept of an application's address space on its head. Rather than concentrating on how *much* memory an application needs to run, a virtual memory operating system continually attempts to find the answer to the question, "how *little* memory does an application need to run?"

While it at first appears that our hypothetical application requires the full 15000 bytes to run, think back to our discussion in Section 4.1 *Storage Access Patterns* — memory access tends to be sequential and localized. Because of this, the amount of memory required to execute the application at any given time is less than 15000 bytes — usually a lot less. Consider the types of memory accesses required to execute a single machine instruction:

- The instruction is read from memory.
- The data required by the instruction is read from memory.
- After the instruction completes, the results of the instruction are written back to memory.

The actual number of bytes necessary for each memory access varies according to the CPU's architecture, the actual instruction, and the data type. However, even if one instruction required 100 bytes of memory for each type of memory access, the 300 bytes required is still much less than the application's entire 15000-byte address space. If a way could be found to keep track of an application's memory requirements as the application runs, it would be possible to keep the application running while using less memory than its address space would otherwise dictate.

But that leaves one question:

If only part of the application is in memory at any given time, where is the rest of it?

4.3.2. Backing Store — the Central Tenet of Virtual Memory

The short answer to this question is that the rest of the application remains on disk. In other words, disk acts as the *backing store* for RAM; a slower, larger storage medium acting as a "backup" for a much faster, smaller storage medium. This might at first seem to be a very large performance problem in the making — after all, disk drives are so much slower than RAM.

While this is true, it is possible to take advantage of the sequential and localized access behavior of applications and eliminate most of the performance implications of using disk drives as backing store for RAM. This is done by structuring the virtual memory subsystem so that it attempts to ensure that those parts of the application currently needed — or likely to be needed in the near future — are kept in RAM only for as long as they are actually needed.

In many respects this is similar to the relationship between cache and RAM: making a little fast storage and a lot of slow storage act just like a lot of fast storage.

With this in mind, let us explore the process in more detail.

4.4. Virtual Memory: The Details

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.

4.5. Virtual Memory Performance Implications

access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Introduction_to_System_Administration/s1-memory-concepts-perf.html

While virtual memory makes it possible for computers to more easily handle larger and more complex applications, as with any powerful tool, it comes at a price. The price in this case is one of performance — a virtual memory operating system has a lot more to do than an operating system incapable of supporting virtual memory. This means that performance is never as good with virtual memory as it is when the same application is 100% memory-resident.

However, this is no reason to throw up one's hands and give up. The benefits of virtual memory are too great to do that. And, with a bit of effort, good performance is possible. The thing that must be done is to examine those system resources impacted by heavy use of the virtual memory subsystem.

4.5.1. Worst Case Performance Scenario

For a moment, take what you have read in this chapter and consider what system resources are used by extremely heavy page fault and swapping activity:

- RAM — It stands to reason that available RAM is low (otherwise there would be no need to page fault or swap).
- Disk — While disk space might not be impacted, I/O bandwidth (due to heavy paging and swapping) would be.
- CPU — The CPU is expending cycles doing the processing required to support memory management and setting up the necessary I/O operations for paging and swapping.

The interrelated nature of these loads makes it easy to understand how resource shortages can lead to severe performance problems.

All it takes is a system with too little RAM, heavy page fault activity, and a system running near its limit in terms of CPU or disk I/O. At this point, the system is thrashing, with poor performance the inevitable result.

4.5.2. Best Case Performance Scenario

At best, the overhead from virtual memory support presents a minimal additional load to a well-configured system:

- RAM — Sufficient RAM for all working sets with enough left over to handle any page faults[1]
- Disk — Because of the limited page fault activity, disk I/O bandwidth would be minimally impacted
- CPU — The majority of CPU cycles are dedicated to actually running applications, instead of running the operating system's memory management code

From this, the overall point to keep in mind is that the performance impact of virtual memory is minimal when it is used as little as possible. This means the primary determinant of good virtual memory subsystem performance is having enough RAM.

Next in line (but much lower in relative importance) are sufficient disk I/O and CPU capacity. However, keep in mind that these resources only help the system performance degrade more gracefully from heavy faulting and swapping; they do little to help the virtual memory subsystem performance (although they obviously can play a major role in overall system performance).