Introduction

Let's take a deeper look at how entire virtual memory systems are built together before we wrap up our virtual memory studies. Many page-table designs, interactions with the TLB (sometimes even managed by the OS itself), and tactics for determining which pages to stay in memory and which to kick out have all been seen. A comprehensive virtual memory system, on the other hand, has a variety of performance, functionality, and security characteristics. As a result, here's the deal:

What features are required to create a fully functional virtual memory system? How do they improve the system's performance, security, or other aspects?

We'll do so by looking at two different systems. The first is one of the earliest examples of a "modern" virtual memory manager, which can be found in the VAX/VMS operating system, which was developed in the 1970s and early d; a surprising number of techniques and approaches from this system have survived to this day, making it well worth studying. Some ideas, even if they are 50 years old, are still worth understanding, a concept that is common knowledge in most other professions (for example, physics), but must be articulated in technology-driven sciences (e.g., Computer Science).
The second is that of Linux, which should be self-evident. Linux is a widely used operating system that can run on systems as small and low-powered as phones all the way up to the most scalable multicore processors seen in modern data centers. As a result, its virtualization technology must be adaptable enough to work in all of these scenarios. We'll go through each system in detail to show how the concepts introduced in previous chapters work together to form a full memory manager.

VAX/VMS Virtual Memory

DEC released the VAX-11 minicomputer architecture in the late 1970s (DEC). While DEC was a major participant in the computer business during the mini-computer era, a succession of disastrous moves and the introduction of PC led to its demise. The architecture was implemented in the VAX-11/780 and the VAX-11/750.

The system's operating system, VAX/VMS (or just VMS), was designed by Dave Cutler, who eventually oversaw the effort to develop Microsoft's Windows NT. VMS had the general issue of being run on a wide range of machines, from low-cost VAXen (yes, plural) to high-end and powerful computers in the same architecture family. So the OS needed processes and policies that functioned well across a wide range of systems.

Also, VMS is a great example of software advances exploited to cover architectural problems. However, when designing efficient abstractions and illusions, the hardware designers don't always get it right; we'll examine a few examples of this in the VAX hardware, and what the VMS operating system does to produce an effective, working system despite the hardware defects.

Memory Management Hardware

The VAX-11 provided 512-byte virtual address space per process. So a virtual address was a 23-bit VPN plus a 9-bit offset. The upper two bits of the VPN were also used to differentiate between segments, making the method a combination of paging and segmentation.

Process space is the lowest half of the address space. The user program (P0) is situated in the first half of process space (P0). The stack grows upward in the second half of process space (P1). System space (S) is the upper half of the address space. The OS code and data are protected here, and are shared between processes.

The VAX hardware's limited page size was a major problem for the VMS designers (512 bytes). This size, chosen for historical reasons, causes overly large linear page tables. The VMS designers wanted to avoid overloading memory with page tables.
The system lowered memory stress in two ways. First, by segmenting the user address space into two areas (P0 and P1) per process, the VAX-11 eliminates the requirement for page table space between the stack and the heap. The base and bounds registers hold the address and size of the segment's page table, respectively (i.e., number of page-table entries).

Second, the OS decreases memory load by storing user page tables (for P0 and P1) in kernel virtual memory (KVM). So, while allocating or expanding a page table, the kernel uses virtual memory in segment S. If memory is under extreme stress, the kernel can shift these page tables to disk, freeing up physical memory.

Putting page tables in kernel virtual memory complicates address translation. The hardware must first look for the page-table entry for that page in its own page table (the P0 or P1 page table for that process), which may require consulting the system page table (which resides in physical memory). The VAX's hardware-managed TLBs, which normally (ideally) avoid this arduous lookup, help make this process speedier.

A Real Address Space

VMS allows us to observe how an actual address space is built (Figure 23.1. As shown above, a real address space is much more sophisticated than a simple user code, data, and heap address space.

So the code segment never starts at page 0. Instead, this page is declared inaccessible to help detect null-pointer accesses. So, while creating an address space, consider debugging support, which the inaccessible zero page provides.

The kernel virtual address space (data structures and code) are part of each user address space. The OS alters the P0 and P1 registers to point to the process's page tables, but not the S base and bound registers, so the "same" kernel structures are mapped into each user address space.

Reasons why the kernel is mapped to each address space The OS can easily copy data from a user program's pointer (e.g., on a write() system call) to its own structures using this design. The OS is written and constructed naturally, without regard for the data it accesses. Moving data between user apps and the kernel would be difficult and painful if the kernel were fully in physical memory. With this design, the kernel appears to programs as a protected library.

Last but not least, this address area is protected. This prevents apps from reading or writing OS data. To enable this, the hardware must support multiple page protection levels. The VAX accomplished it by stating in the page table protection bits what privilege level the CPU must have to access a page. Thus, attempts to access system data and code from user code will result in an OS trap and (you guessed it) the termination of the offending process.

Page Replacement

A valid bit, a protection field (4 bits), a modify (or dirty) bit, a field reserved for the OS (5 bits), and lastly a physical frame number (PFN) to store the page's actual memory address. No reference bit, astute reader! It can't use hardware to determine which pages are active.

Memory hogs, programs that utilize a lot of memory and slow down other processes, were also a worry. This is true of most of the policies we've looked at so far; for example, LRU is a global policy that doesn't share memory fairly.

The developers devised the segmented FIFO replacement policy to overcome these issues. The concept is simple: each process has a memory limit (known as the resident set size) (RSS). The "first-in" page is evicted when a process exceeds its RSS. FIFO is simple to create and does not require hardware support.

Of fact, as we saw earlier, pure FIFO is inefficient. To increase FIFO performance, VMS included two second-chance lists, a global clean-page free list and a dirty-page list. An out-of-order page is removed from the per-process FIFO and placed at the end of the clean-page list, or the dirty-page list, if it has been updated.

So, if another process Q wants a page, it removes the first one. The free (or filthy) list is reclaimed from the original process P if the original process P fails on that page before it is reclaimed. The segmented FIFO technique approaches LRU performance as global second-chance lists grow.

Another VMS improvement helps overcome the tiny page size. Disk I/O during swapping could be wasteful due to the short page size. VMS optimizes swapping I/O in numerous ways, but the most essential is clustering. Clustering gathers huge batches of pages from the global dirty list and writes them all at once (thus making them clean). The ability to store pages wherever within swap space allows the OS to aggregate pages, execute fewer and larger writes, and therefore increase performance.

Other Neat Tricks

VMS also used demand zeroing and copy-on-write. These are lazy optimizations. VMS (and most current systems) demand page zeroing. Consider adding a page to your address space, say in your heap. In a simple implementation, the OS finds a page in physical memory, zeros it (for security; otherwise, you'd be able to see what was on the page when another process used it!), and maps it into your address space (i.e., setting up the page table to refer to that physical page as desired). But naive implementation might be costly, especially if the page is never utilized.

The OS does very little effort when adding a page to your address space; it just adds an entry to the page table marking it unavailable. When the process reads or writes the page, it traps the OS. A demand-zero page is identified by the OS (typically by certain bits designated in the "reserved for OS" portion of the page table entry). The OS then finds a physical page, zeroes it, and maps it into the process's address space. This task is avoided if the process never accesses the page.

Copy-on-write is another great VMS (and current OS) improvement (COW for short). The principle is simple: instead of copying a page from one address space to another, the OS can map it into the target address space and declare it read-only in both address spaces. If both address spaces just read the page, no further action is taken by the OS, and hence no data is moved.

A page write attempt from one of the address spaces will be logged into the OS. The OS will then (lazily) allocate a new page, fill it with data, and map it into the address space of the faulting process. The procedure repeats, creating a private duplicate of the page.

COW has various uses. Any shared library can be mapped copy-on-write into numerous processes' address spaces, saving memory. COW is crucial in UNIX systems due to fork() and exec semantics (). Fork(), as you may recall, generates an exact replica of the caller's address space; this is slow and data-intensive. Worse, a later exec() call overwrites most of the address space, combining the calling process's and the program's address spaces. By doing a copy-on-write fork() instead, the OS saves much of the unnecessary copying and improves efficiency.