**Chapter 23: Complete Virtual Memory Systems**

Cover two systems: **VAX/VMS** and **Linux**

**23.1 VAX/VMS Virtual Memory**

**Memory Management Hardware**

The VAX-11 provides a 32-bit virtual address space per process, divided into 512-byte pages. The system is a hybrid of paging and segmentation.

The lower-half of the address space was known as “process space” and is unique to each process. In the first half of process space (known as P0), the user program is found, as well as a heap which grows downward. In the second half of process space (P1), we find the stack, which grows upwards. The upper-half of the address space is known as system space (S), although only half of it is used. Protected OS code and data reside here, and the OS is in this way shared across processes.

A concern of the VMS designers was the small size of pages in the VAX hardware. Thus, the first goals of the VMS designers was to ensure that VMS would not overwhelm memory with page tables.

The system reduced the pressure page tables place on memory in two ways.

1. First, by segmenting the user address space into two, the VAX-11 provides a page table for each of these regions (P0 and P1) per process. Thus, no page-table space is needed for the unused portion of the address space between the stack and the heap.
2. The OS reduces memory pressure further by placing user page tables in kernel virtual memory (in segment S), but this also means that address translation is even more complicated.

**A Real Address Space**

Page 0 is never used for code segment, but it is marked inaccessible, in order to provide support for detecting null-pointer accesses. Thus, one concern when designing an address space is support for debugging, which the inaccessible zero page provides here in some form.

More importantly, the kernel virtual address space (i.e., its data structures and code) is a part of each user address space. On context switch, the OS changes the P0 and P1 registers to point to the appropriate page tables of the soon-to-be-run process, but it does not change the S base and bound register. As a result, the same kernel structures are mapped into each user address space.

This construction makes life easier for the kernel; when, for example, the OS is handed a pointer from a user program, it is easy to copy data from that pointer to its own structures.

The last point about this address space relates to protection. The VAX did so by specifying, in protection bits in the page table, what privilege level the CPU must be at in order to access a particular page.

**Page Replacement**

The page table entry (PTE) in VAX contains the following bits: a valid bit, a protection field (4 bits), a modify (or dirty) bit, a field reserved for OS use (5 bits), and finally a physical frame number (PFN) to store the location of the page in physical memory (no **reference bit**).

Chart, diagram, box and whisker chart

Description automatically generated

The developers were also concerned about **memory hogs**, programs that use a lot of memory and make it hard for other programs to run.

To address these two problems, the developers came up with the **segmented FIFO** replacement policy. The idea is that each process has a maximum number of pages it can keep in memory, known as its **resident set size (RSS)**. Each of these pages is kept on a FIFO list; when a process exceeds its RSS, the “first-in” page is evicted. This method is easy to be implemented.

FIFO does not normally perform well. To improve FIFO’s performance, VMS introduced two **second chance lists** where pages are placed before getting evicted from memory, specifically a global **clean-page free list** and **dirty-page list**. When a process P exceeds its RSS, a page is removed from its per-process FIFO; if clean (not modified), it is placed on the end of the clean-page list; if dirty (modified), it is placed on the end of the dirty-page list.

To make swapping I/O more efficient, VMS adds a number of optimizations, but most important is **clustering**. With clustering, VMS groups large batches of pages together from the global dirty list, and writes them to disk in one fell swoop (thus making them clean). Clustering is used in most modern systems, as the freedom to place pages anywhere within swap space lets the OS group pages, perform fewer and bigger writes, and thus improve performance.

**Other Neat Trickes**

VMS had two other now-standard tricks: demand zeroing and copy-on-write. These are lazy optimizations.

With **demand zeroing**, the OS instead does very little work when the page is added to your address space. It puts an entry in the page table that marks the page inaccessible. If the process then reads or writes the page, a trap into the OS takes place. When handling the trap, the OS notices that this is actually a demand-zero page; at this point, the OS does the needed work of finding a physical page, zeroing it, and mapping it into the process’s address space. If the process never accesses the page, all such work is avoided, and thus the virtue of demand zeroing.

The idea of copy-on-write is simple. When the OS needs to copy a page from one address to another instead of copying it, it can map it into the target address space and mark it read only in both address spaces.

**23.2 The Linux Virtual Memory System**

**The Linux Address Space**

Diagram

Description automatically generated with medium confidence

One slightly interesting aspect of Linux is that it contains two types of kernel virtual addresses. The first are known as **kernel logical addresses (kmalloc)**. This is what you would consider the normal virtual address space of the kernel.

The other type of kernel address is a **kernel virtual address (vmalloc)**.

**Page Table Structure**

A picture containing chart

Description automatically generated

The top 16 bits of a virtual address are unused (and thus play no role in translation), the bottom 12 bits (due to the 4-KB page size) are used as the offset (and hence just used directly, and not translated), leaving the middle 36 bits of virtual address to take part in the translation.

**Large Page Support**

recent designs support 2-MB and even 1-GB pages in hardware. Linux has evolved to allow applications to utilize these **huge pages**. Doing so reduces the number of mappings that are needed in the page table. The larger the pages, the fewer the mappings. In addition, it’s better TLB behavior and related performance gains.

Huge pages allow a process to access a large tract of memory without TLB misses, by using fewer slots in the TLB, and thus is the main advantage. There is a shorter TLB-miss path, meaning that when a TLB miss does occur, it is serviced more quickly. In addition, allocation can be quite fast (in certain scenarios), a small but sometimes important benefit.

However, huge pages create internal fragmentation.

**The Page Cache**

This method reduces costs of accessing persistent storage.

The Linux page cache is unified, keeping pages in memory from three primary sources: **memory-mapped files**, file data and metadata from devices, and heap and stack pages that comprise each process. These entities are kept in a page cache hash table, allowing for quick lookup when said data is needed.

The page cache tracks if entries are **clean** (read but not updated) or **dirty** (a.k.a., modified).

Linux uses **2Q replacement** to kick out memory to free up spaces.

**Security And Buffer Overflows**

The first and most simple defense against buffer overflow is to prevent execution of any code found within certain regions of an address space. The NX bit just prevents execution from any page which has this bit set in its corresponding page table entry.

When injected code cannot be added explicitly by the attacker, arbitrary code sequences can be executed by malicious code. This is called **return-oriented programming (ROP)**. An attacker can overwrite the stack such that the return address in the currently executing function points to a desired malicious instruction followed by a return instruction. By stringing together a large number of gadgets (i.e., ensuring each return jumps to the next gadget), the attacker can execute arbitrary code.

To defend against this, LINUX uses **address space layout randomization (ASLR)**. Instead of placing code, stack, and the heap at fixed locations within the virtual address space, the OS randomizes their placement, thus making it quite challenging to craft the intricate code sequence required to implement this class of attacks.

**Other Security Problems: Meltdown And Spectre**

The general weakness exploited in each of these attacks is that the CPUs found in modern systems perform all sorts of crazy behind-thescenes tricks to improve performance.

One avenue to increasing kernel protection was thus to remove as much of the kernel address space from each user process and instead have a separate kernel page table for most kernel datas