Chapter 23: Complete Virtual Memory Systems

This chapter examines how the mechanisms and policies of virtual memory (VM) are integrated into complete, real-world systems. By studying two influential systems—the classic **VAX/VMS** and the modern **Linux**—we can see how foundational concepts are adapted to solve practical problems related to performance, functionality, and security.

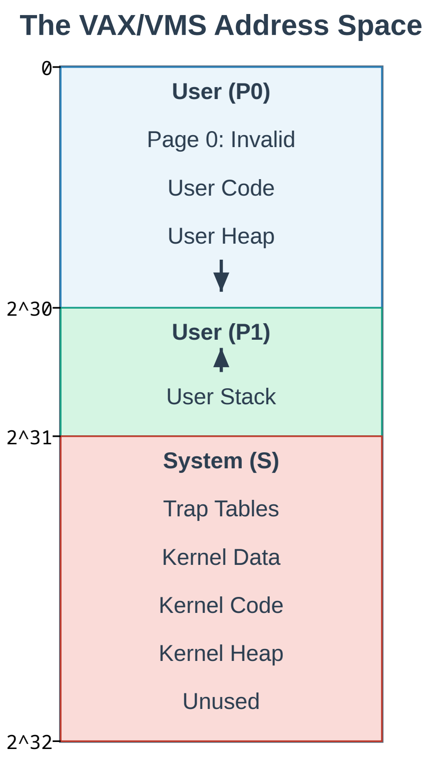
> **The Crux of the Problem: How to Build a Complete VM System?**
>
> What features are needed to realize a complete virtual memory system? How do they improve performance, increase security, or otherwise improve the system?

1. VAX/VMS Virtual Memory: A Classic System

The VAX/VMS system, developed by DEC in the 1970s, is a foundational example of a modern VM system. Many of its innovations are still relevant today.

1.1. Memory Management Hardware

- **Address Space:** VAX/VMS used a 32-bit virtual address space, split into two halves:
  - **Process Space (P0 & P1):** The lower half, unique to each process. P0 contained the user program and a downward-growing heap. P1 contained an upward-growing stack.
  - **System Space (S):** The upper half, shared across all processes, containing the OS kernel's code and data.
- **Page Size:** A very small 512-byte page size, which created a major challenge: potentially huge page tables.
- **Solutions to Large Page Tables:**
  1. **Segmentation:** The address space was segmented (P0, P1, S). This avoided allocating page table entries for the large, unused gap between the heap and stack.
  2. **Paged Page Tables:** User page tables (for P0 and P1) were themselves placed in *kernel virtual memory.* This allowed the OS to swap parts of a process's page table to disk if memory pressure became severe.

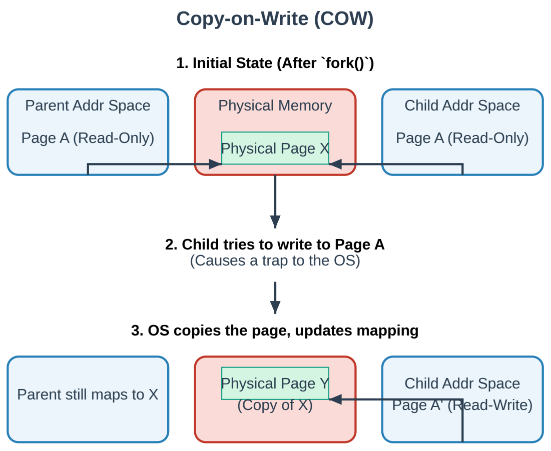


1.2. Page Replacement Policy

- **Hardware Limitation:** The VAX hardware did **not** have a reference/use bit, making a direct LRU approximation like the Clock algorithm impossible.
- **VMS Solution: Segmented FIFO:**
  1. Each process was assigned a **Resident Set Size (RSS)**, the maximum number of pages it could keep in physical memory.
  2. Pages for each process were managed in a per-process **FIFO** list.
  3. When a process exceeded its RSS, the oldest page was evicted.
  4. **Second-Chance Lists:** To improve on pure FIFO, evicted pages were not immediately discarded. Instead, they were placed on one of two global lists: a **clean-page list** or a **dirty-page list**.
  5. If a process faulted on a page that was still on one of these second-chance lists, it could reclaim it quickly without a disk I/O. This mechanism approximated LRU behavior.

1.3. Other VMS Innovations

- **Clustering:** To make swapping more efficient with small pages, VMS would **cluster** (group) many dirty pages together and write them to disk in a single, large, efficient I/O operation.
- **Demand Zeroing:** A lazy optimization. When a new page is added to the heap, the OS doesn't immediately find a physical frame and zero it. Instead, it marks the page as inaccessible. Only when the process first tries to access the page does a trap occur, at which point the OS finds a frame, zeroes it, and maps it in. This avoids work for pages that are allocated but never used.
- **Copy-on-Write (COW):** A lazy optimization for copying pages (e.g., during `fork()`). Instead of physically copying the page, the OS maps the *same* physical page into both address spaces and marks it as **read-only**. If either process later tries to *write* to the page, a trap occurs. Only then does the OS make a true physical copy of the page for the writing process. This makes `fork()` extremely fast.



---

2. The Linux Virtual Memory System

The Linux VM is a modern, feature-rich system that runs on a vast range of hardware, from phones to supercomputers. We focus on its implementation for the x86 architecture.

2.1. The Linux Address Space

- Like VMS, the Linux address space is split into a user portion and a kernel portion. On a context switch, the user portion changes, while the kernel mapping remains the same.

- **Kernel Address Types:**
  - **Kernel Logical Addresses:** A region of the kernel's virtual space that is directly mapped to the first portion of physical memory. `kmalloc()` allocates from here. Memory that is contiguous in this space is also contiguous in physical memory, making it suitable for DMA.
  - **Kernel Virtual Addresses:** A separate region allocated by `vmalloc()`. This memory is virtually contiguous but not necessarily physically contiguous. It's used for large buffers where finding a large contiguous physical block would be difficult.

2.2. Page Table Structure

- Linux leverages the hardware-managed, multi-level page table structure of the x86 architecture.
- Modern 64-bit x86 systems use a **four-level page table** to manage the massive 48-bit virtual address space. This deep hierarchy is necessary to keep the page table components page-sized, but it means a TLB miss can require up to four memory accesses to find a translation.

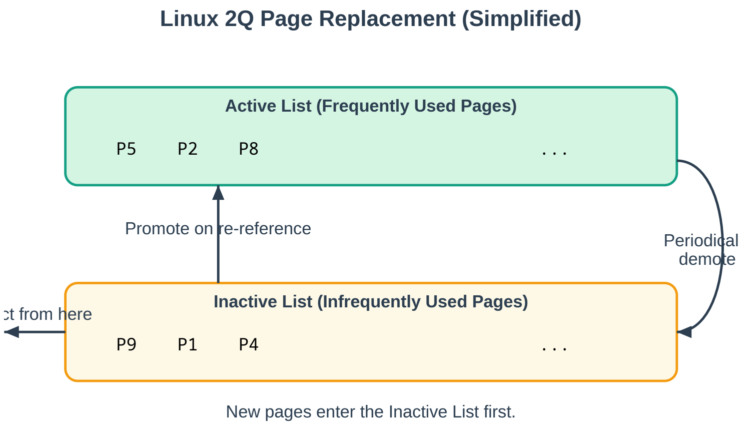
2.3. Large Page Support

- **Motivation:** For applications with very large memory footprints ("big memory" workloads), the TLB can become a bottleneck. Even with a high hit rate, the sheer number of accesses can lead to many TLB misses, degrading performance.
- **Solution:** Modern x86 CPUs support **large pages** (e.g., 2MB or 1GB). Using a single large page allows a single TLB entry to cover a much larger memory region, dramatically reducing TLB misses for certain applications (like databases).
- **Linux Implementation:**
  - **Explicit:** Applications can request large pages via `mmap()`.
  - **Transparent Huge Pages (THP):** The OS automatically tries to use large pages for an application's anonymous memory (heap and stack) without requiring any changes to the application.

2.4. Page Cache and Replacement

- **Unified Page Cache:** Linux uses a single, unified page cache to hold:
  - Memory-mapped files.
  - Regular file data (from `read()`/`write()`).
  - Anonymous memory (heap and stack pages backed by swap space).
- **Page Replacement Policy (A 2Q Variant):**
  - Linux uses a sophisticated LRU approximation similar to the **2Q algorithm** to avoid the worst-case behavior of simple LRU (e.g., with large, sequential file scans).
  - It maintains two lists: an **active list** and an **inactive list**.

- A page is first placed on the inactive list. If it is accessed again while on this list, it is promoted to the active list.
- Replacement victims are chosen from the inactive list. This prevents a large, one-time scan from flushing all the useful, frequently-accessed pages from the active list.


Linux 2Q Page Replacement (Simplified)
Active List (Frequently Used Pages)
P5 P2 P8 ...
Promote on re-reference
Periodical demote
Inactive List (Infrequently Used Pages)
ct from here
P9 P1 P4 ...
New pages enter the Inactive List first.

2.5. Modern Security Features

Modern systems face significant security threats, leading to new VM features.

- **Buffer Overflow Attacks:** A common attack where a malicious user provides an overly long input to a program, overflowing a buffer on the stack and overwriting critical data, such as a function's return address.
- **Defense 1: No-eXecute (NX) Bit:** A hardware feature that allows the OS to mark pages (like the stack and heap) as non-executable. This prevents an attacker from directly running malicious code injected into the stack.
- **Defense 2: Address Space Layout Randomization (ASLR):** The OS randomizes the location of the code, heap, and stack in the virtual address space for each run. This makes it extremely difficult for an attacker to guess the addresses needed to carry out more advanced attacks like **Return-Oriented Programming (ROP)**.
- **Defense 3: Kernel Page-Table Isolation (KPTI):** A defense against hardware vulnerabilities like **Meltdown**. It removes most of the kernel's mappings from the user process's page table, using a separate kernel page table instead. This improves security but adds performance overhead due to the need to switch page tables on every entry to and exit from the kernel.