Virtual Memory System Examples

CMPU 334 – Operating Systems
Jason Waterman
How to build a complete VM system

• Look at real VM systems
  • VAX/VMS
  • Linux

• What features are needed to realize a complete VM system?
• Look at performance, security, and other improvements
VMS Virtual Memory System

• VAX-11 Minicomputer introduced in the late 70’s

• VMS was designed to run on a broad range of machines
  • Mechanisms and policies had to work well across a range of systems

• 32-bit virtual address, with 512-byte pages
  • How many offset bits?
  • How many VPN bits?

• Upper two bits were used as segment bits
  • Used both segments and page tables
  • Hybrid system
VAX/VMS Address Space

• Very small size for pages (512 bytes)
  • Chosen for historical reasons
  • Excessively large linear page table
    • Use segments to help reduce size of tables

• Four Segments (determined by bits 31 and 30)
  • 00 – User (P0)
  • 01 – User (P1)
  • 10 – System (S)
  • 11 – Unused

• Process space (P0 and P1)
  • Used for each process
  • Base and bounds register for each segment
    • Base holds the address of page table for segment
    • Bounds holds the size of the page table

• System space (S)
  • Protected OS code and data
  • Shared across processes
Notes on the address space

• Page 0 is marked as invalid
  • Used to detect NULL pointer dereferences
• Kernel virtual address space is part of each users address space
  • Makes it easy for kernel to copy data from a process to its own structures
  • Kernel appears as a protected library to applications
• Protection for kernel pages done by adding protection bits in the page table
  • What privilege level the CPU must be at to access a page
VAX Page Table Entries

- Page table entry (PTE) in VAX
  - Valid bit
  - 4 Protection bits
  - Modified (dirty) bit
  - 5-bits reserved for OS
  - Physical frame number (PFN)

- Note: no reference bit
  - Replacement algorithm doesn’t have hardware support for knowing which pages are active
Controlling Memory Hogs

• Developers of VMS were concerned about programs that use a lot of memory, making it hard for other programs to run

• LRU is a global policy
  • Doesn’t share memory fairly among processes

• Solution: segmented FIFO replacement policy
  • Each process has **resident set size** (RSS)
    • The maximum number of pages it can keep in memory
    • Pages are kept on a FIFO list
    • When a process exceeds its RSS, the “first-in” page is evicted

• Problem: FIFO doesn’t perform very well
Second-chance Lists

• To improve FIFO’s performance VMS uses second-chance lists

• A global list where pages are placed before getting evicted from memory
  • Clean-page free list
  • Dirty-page free list

• When a process exceeds its RSS
  • “First-in” page is removed from its per-process FIFO
    • Placed on global clean or dirty second-chance list depending on whether it has been modified

• If another process needs a page, it takes the first free page off the clean list

• If a process faults on one of its pages in the second-chance list it reclaims that page from either the free or dirty list
  • Avoids costly disk access
Page Clustering

• VMS groups large batches of pages together from the global dirty list and writes them out to disk together
  • Optimization to help overcome the small page size in VMS
  • Amortizes the cost of writing one page across the cluster of pages
• After the pages are written out to disk, we can mark them clean
Demand Zeroing of Pages

• For security purposes, the OS should zero out any new pages added to a processes virtual address space
  • Otherwise you would be able to see what was on the page from when some other process used it
    • Could leak encryption keys for example

• Demand zeroing defers the zeroing out of a page until it is referenced

• Example: adding a page of memory to the heap of a process
  • When a page is added to the page table, it is marked as inaccessible (note: this is different than invalid)
    • Physical page is not allocated at this time
  • If the process accesses the page, a trap into the OS takes place
    • OS then finds a physical page, zeros it, and maps it the processes virtual address space (by updating the PTE)
  • If the process never accesses that page, no work is done
Copy-on-write (COW)

• Optimization for copying a page from one address space to another
• Instead of copying the page
  • Mark the physical page as read only (copy-on-write)
  • Have the virtual page in both address spaces map to the (read only) physical page
    • If both pages never write to the page, no further work is done
  • If one of the address spaces does try to write to the page
    • Trap into the OS, which sees the page was marked copy-on-write
    • Then the OS can allocate a new page, fill with data, add to the PTE
• Shared libraries can be marked COW for many processes
• Helpful for \texttt{fork()} and \texttt{exec()}
  • \texttt{fork()} creates an exact copy, but usually is immediately overwritten by \texttt{exec()}
  • Copy-on-write prevents this needless copying
The Linux Virtual Memory System

- Focus on Linux Intel x86
- Address Space divided into kernel and user space
  - Each user process has own userspace
  - Kernel region same across processes
- Kernel region is protected, divided into logical and virtual regions
  - Logical – mapped directly to physical memory
    - $0xC0000000 -> 0x00000000$
    - Translation is trivial
    - Contiguous regions in logical memory are contiguous in physical memory
  - Virtual – map to any physical page
    - Easy to allocate
    - Used for large buffers where finding large chunk of physical memory is challenging
Page Table Structure

- Hardware managed, multi-level page table
- One page table per process
  - OS sets up mappings in its kernel memory
  - Points a privileged register at the start of the page directory
  - Hardware handles the rest
- OS gets involved at process creation, deletion, and context switches
- X86-64 (64-bit addresses)
  - Four-level page table, 4k page size
  - Only the bottom 48 bits are currently used
Large Page Support

• X86 allows for multiple page sizes in hardware
  • 2-MB and even 1-GB pages

• Reduces number of mappings in the page table

• More Importantly, it reduces the pressure on the TLB
  • Some applications spend 10% of their system servicing TLB misses

• Implemented in Linux incrementally
  • Applications explicitly request memory with large pages via `mmap()`
  • In recent versions of Linux have added transparent large page support
    • Automatically looks for opportunities to allocate 2 MB pages without application modification

• Issues
  • Larger pages can lead to internal fragmentation
Page Cache

• Since hard drives are so slow, Linux uses aggressive caching to keep popular data items in memory

• Page cache keeps pages in memory from three sources
  • Memory-mapped files
  • File data
  • Heap and stack pages

• Page cache keeps track if entries are clean or dirty
  • Dirty pages are periodically written back to disk by background threads
  • After a certain period of time or if too many pages are dirty

• How does Linux decide which pages to evict?
  • 2Q replacement algorithm
2Q Replacement Algorithm

- Design goal: LRU-like performance without the penalty for some common access patterns
  - Reading through a large file can kick every other file out of memory
  - If you just read through the file once, those pages are never referenced again

- Keep two lists and divide memory between them
  - When a page is accessed for the first time it is put on the inactive list
  - When it is re-referenced, the page is promoted to the active list
  - When needed, pages are evicted from the inactive list
  - Pages are periodically moved from the bottom of the active list to the inactive list
    - Active list is about 2/3 the total page cache size
    - LRU is a approximated by an algorithm similar to the clock replacement algorithm

- Handles the case of cyclic large-file access
  - Pages that are never re-referenced are confined to the inactive list and won’t evict pages from the active list
Security and Buffer Overflows

• How to protect from buffer overflow attacks?
  • Prevent execution of any code found within the stack
  • NX bit (AMD) and XD bit (Intel) prevents execution from any page that has this bit set in the PTE

• Return-oriented programming (ROP) strings together pieces of existing code (called gadgets)
  • Mitigate by performing Address space layout randomization (ASLR)
    • OS randomizes the placement of code, heap, and stack each time a program is run
    • Makes it almost impossible to predict the addresses of the gadgets in memory

• Kernel address space layout randomization (KASLR)
  • Kernel is placed at a random location
Meltdown and Spectre

• These attacks reveal memory by taking advantage of **speculative execution** done by the processor
  • The CPU guess which instructions will soon be executed in the future
  • Starts executing them ahead of time
  • Correct guesses mean the program runs faster
  • Wrong guesses are undone by the CPU by restoring state (registers)

• However, state is not fully restored
  • E.g., caches, branch predictors

• These subtle changes in state can make vulnerable the contents of memory
  • Even protected memory

• Can’t disable speculative execution, programs will run much much slower
Kernel Page Table Isolation (KPTI)

• Remove as much of the kernel address space from each user process
• Have a separate kernel page table for most kernel data
• Kernel code and data structures are no longer mapped into each process
• When switching into the kernel, a switch to the kernel page table is needed
  • The cost is performance
  • Switching page tables is costly
• Solves some, but not all of the problems of speculative execution