

Virtual Memory: Concepts

Kai Shen

1

A System Using Physical Addressing

2

- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing

3

- Used in all modern servers, desktops, and laptops

Address Spaces

- Each process has a **virtual address space**: Set of $N = 2^n$ virtual addresses $\{0, 1, 2, 3, \dots, N-1\}$
- Each machine has a **physical address space**: Set of M physical addresses $\{0, 1, 2, 3, \dots, M-1\}$
- Each virtual address in a process maps to a physical address on the machine
 - Not always ...

4

Why Virtual Memory (VM)?

- Uses main memory efficiently
 - Transparently use DRAM as a cache for parts of a virtual address space
- Simplifies memory management
 - Each process gets the same uniform linear address space
- Isolates address spaces
 - One process can't interfere with another's memory
 - User program cannot access privileged kernel information

5

VM as a Tool for Caching

- A process's virtual memory space
 - Partly cached in memory or DRAM (efficient utilization of memory space) but always backed by a disk copy.
 - VM allows transparent caching by the operating system
 - Memory blocks are called *pages* (typically 4 kilobytes)

6

DRAM Cache Organization

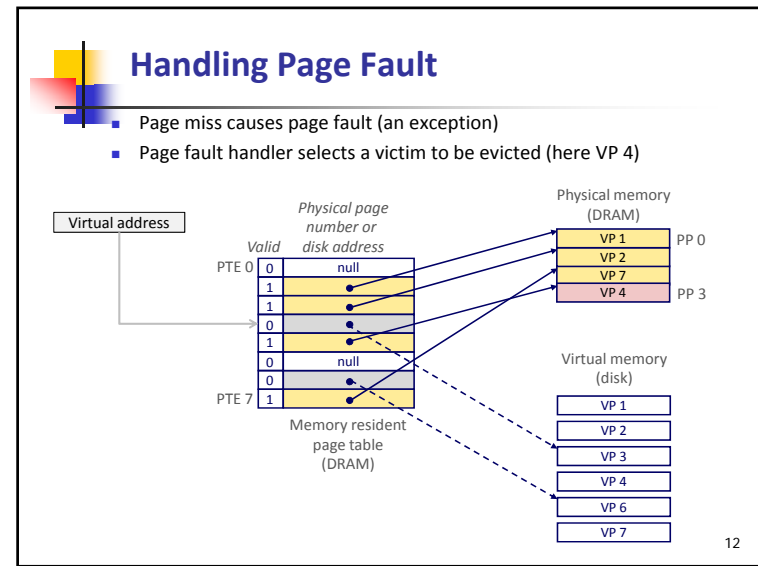
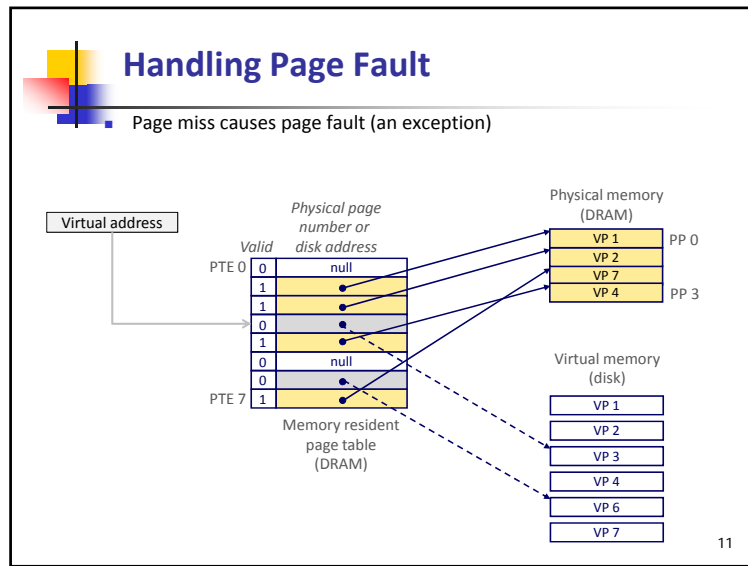
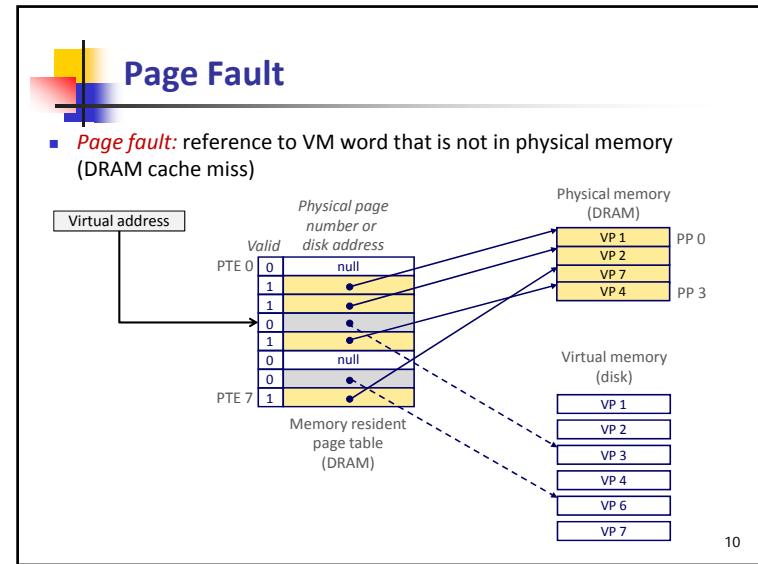
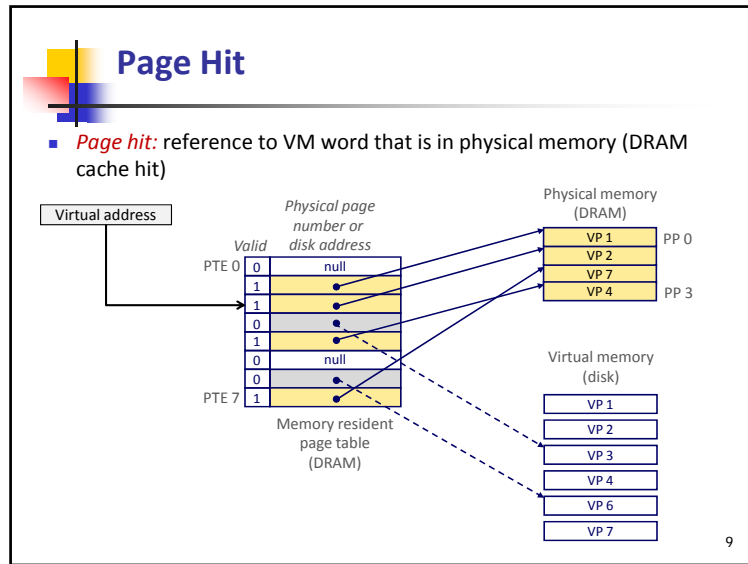
- DRAM cache organization driven by the enormous miss penalty
 - DRAM is about **10x** slower than SRAM
 - Disk is about **10,000x** slower than DRAM
- The speed gap allows complex software management
 - Fully associative
 - Any virtual page can be placed in any physical page
 - Requires a "large" mapping function – different from CPU caches
 - Highly sophisticated, expensive replacement algorithms
 - Write-back rather than write-through

7

Page Tables

- A *page table* is an array of entries (PTEs) that maps virtual pages to physical pages.
 - Per-process kernel data structure in DRAM

8



Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)

The diagram illustrates a memory resident page table (DRAM) with 8 entries (PTE 0 to PTE 7). Each entry contains a 'Valid' bit and a 'Physical page number or disk address'. PTE 0-3 are valid and point to physical pages PP 0-3. PTE 4 is null, causing a page fault. PTE 5-7 are also null. A 'Virtual memory (disk)' contains virtual pages VP 1-7. VP 4 is selected as the victim to be evicted from physical memory.

13

Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!

The diagram shows the same memory resident page table as slide 13. PTE 4 is now valid and points to physical page PP 4. The virtual page VP 4 has been evicted from physical memory but remains in the virtual memory (disk). The offending instruction is restarted, resulting in a page hit.

14

Locality to the Rescue Again!

- Virtual memory-based caching works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
 - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
 - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
 - Thrashing**: Performance meltdown where pages are swapped (copied) in and out continuously

15

VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
 - It can view memory as a simple linear array
 - Mapping function scatters addresses through physical memory as needed
 - Sharing is easy (here: PP 6)

The diagram shows address translation for two processes. Process 1's virtual address space (VP 1, VP 2, ..., N-1) maps to physical address space (PP 2, PP 6, ...). Process 2's virtual address space (VP 1, VP 2, ..., N-1) maps to physical address space (PP 8, PP 6, ..., M-1). Physical page PP 6 is shared by both processes and is noted as 'e.g., read-only library code'.

16

Simplifying Linking and Loading

The diagram shows a vertical stack of memory regions. At the top is 'Kernel virtual memory' (0xc0000000 to 0xc0000000), which is 'Memory invisible to user code'. Below it is the 'User stack (created at runtime)' starting at 0xc0000000 and ending at %esp (stack pointer). Next is the 'Memory-mapped region for shared libraries' starting at 0x40000000. Below that is the 'Run-time heap (created by malloc)' ending at brk. The bottom section, starting at 0x08048000, is 'Loaded from the executable file' and contains a 'Read/write segment (.data, .bss)' and a 'Read-only segment (.init, .text, .rodata)'. The bottom-most region is 'Unused' (0x08048000 to 0).

- Linking**
 - Each program has similar virtual address space
 - Code, stack, and shared libraries always start at the same address
- Loading**
 - execve() allocates virtual pages for .text and .data sections = creates PTEs marked as invalid
 - The .text and .data sections are copied, page by page, on demand by the virtual memory system

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
 - If violated, send process SIGSEGV (segmentation fault)

The diagram shows two processes, Process i and Process j, with their virtual pages mapped to physical pages. Process i has VP 0 (PP 6), VP 1 (PP 4), and VP 2 (PP 2). Process j has VP 0 (PP 9), VP 1 (PP 6), and VP 2 (PP 11). Each virtual page has a table with READ and WRITE permissions. For example, VP 0 of Process i has READ: Yes, WRITE: No. Arrows show the mapping from virtual pages to physical pages in the 'Physical Address Space'.

VM Address Translation

- Virtual address must be translated into physical address for data accesses
- When does it happen?
 - Compile time? Link time? Load time? Runtime?
- It must be fast!
- It must be protected!

Address Translation With a Page Table

The diagram illustrates the translation process. A 'Virtual address' is split into a 'Virtual page number (VPN)' (bits n-1 to p) and a 'Virtual page offset (VPO)' (bits p-1 to 0). The VPN is used to find an entry in the 'Page table'. The entry contains a 'Valid' bit and a 'Physical page number (PPN)'. If the valid bit is 0, it's a page fault. The PPN and VPO are combined to form the 'Physical address', which is split into a 'Physical page number (PPN)' (bits m-1 to p) and a 'Physical page offset (PPO)' (bits p-1 to 0). A box asks 'Where is the page table?'.

Address Translation: Page Hit

- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

21

Address Translation: Page Fault

- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

22

Speeding up Translation with a TLB

- Page table entries (PTEs) are in memory
 - One access to memory becomes two!
- Solution: *Translation Lookaside Buffer* (TLB)
 - Small hardware cache in MMU
 - Maps virtual page numbers to physical page numbers
 - Contains complete page table entries for small number of pages

23

TLB Hit

A TLB hit eliminates a memory access

24

TLB Miss

A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare in practice. Why?

25

Multi-Level Page Tables

- Suppose:
 - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem:
 - Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes
- Common solution:
 - Multi-level page tables
 - Example: 2-level page table
 - Level 1 table: each PTE points to a page table (always memory resident)
 - Level 2 table: each PTE points to a page (paged in and out like any other data)

26

A Two-Level Page Table Hierarchy


32 bit addresses, 4KB pages, 4-byte PTEs

27

Summary

- Programmer's view of virtual memory
 - Each process has its own private linear address space
 - Cannot be corrupted by other processes
- System view of virtual memory
 - Uses memory efficiently by caching virtual memory pages
 - Efficient only because of locality
 - Simplifies memory management and programming
 - Simplifies protection by providing a convenient interpositioning point to check permissions
 - Virtual address translation must be fast and protected

28



Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

29