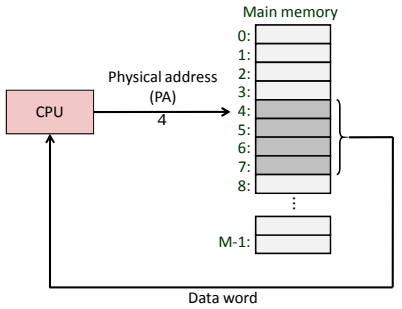
Virtual Memory: Concepts

Kai Shen

1

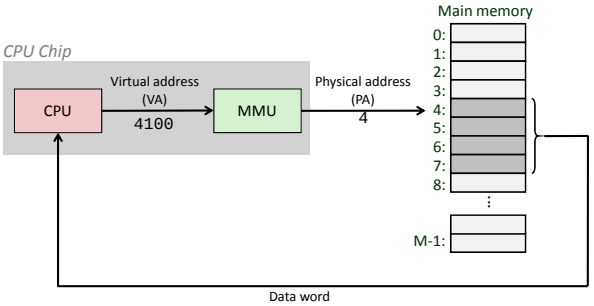
A System Using Physical Addressing



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

2

A System Using Virtual Addressing



- Used in all modern servers, desktops, and laptops

3

Address Spaces

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

4

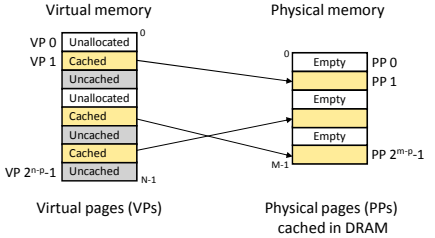
Why Virtual Memory (VM)?

- 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

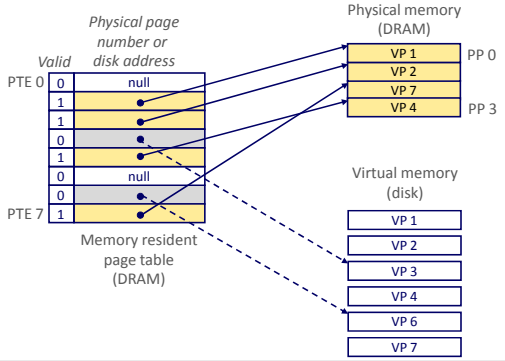
DRAM Cache Organization

- 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

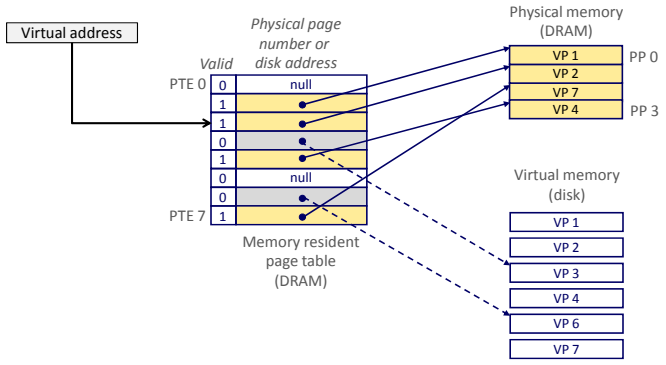
- A *page table* is an array of entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM



8

Page Hit

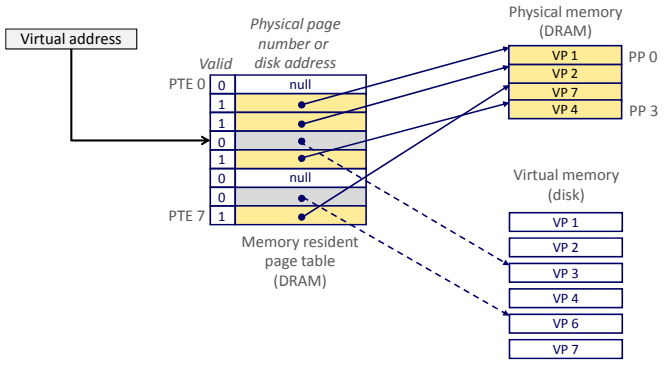
- *Page hit:* reference to VM word that is in physical memory (DRAM cache hit)



9

Page Fault

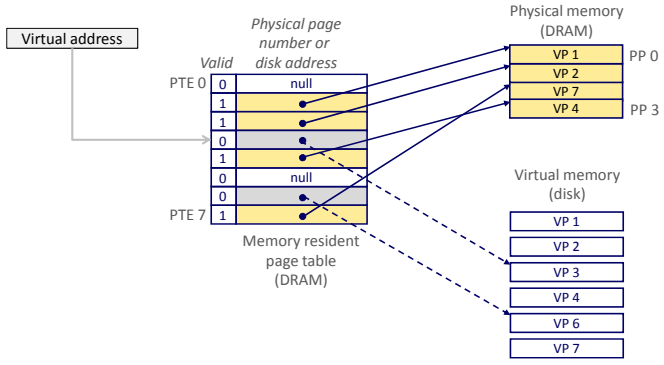
- *Page fault:* reference to VM word that is not in physical memory (DRAM cache miss)



10

Handling Page Fault

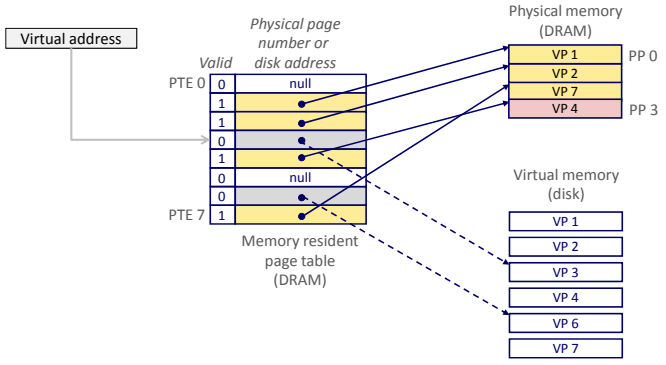
- Page miss causes page fault (an exception)



11

Handling Page Fault

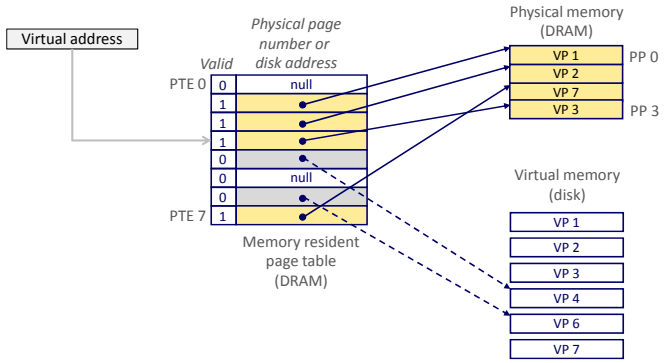
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



12

Handling Page Fault

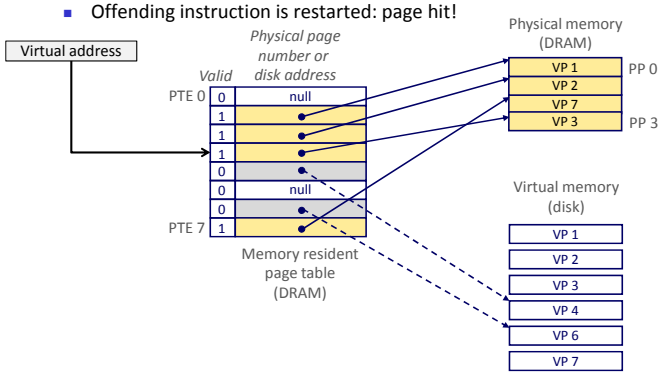
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



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!



14

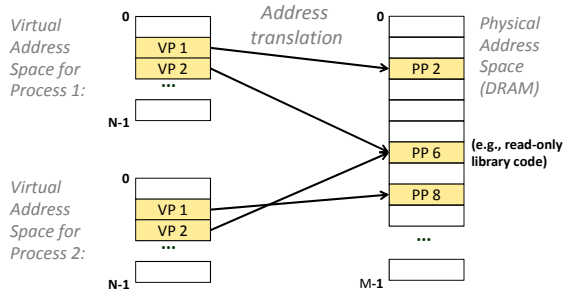
Locality to the Rescue Again!

- 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

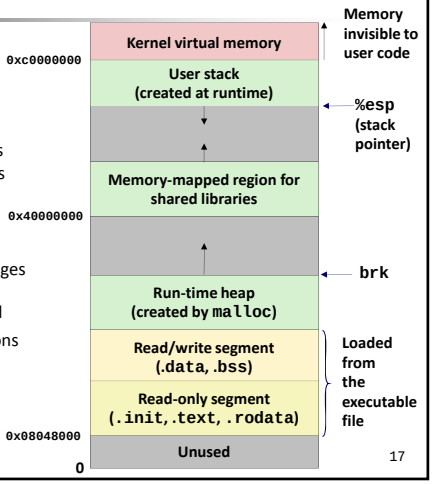
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory as needed
  - Sharing is easy (here: PP 6)



16

Simplifying Linking and Loading

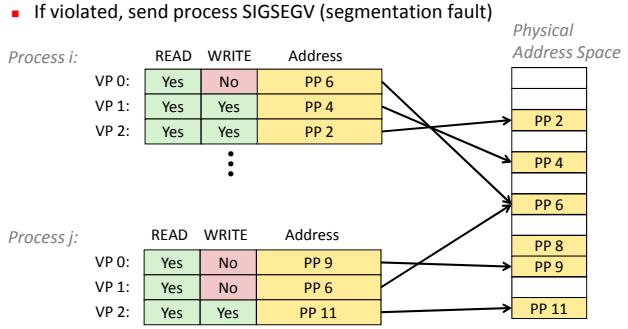
- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address
- Loading
  - `execve()` allocates virtual pages for .text and .data sections = creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page,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



17

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)



18

VM Address Translation

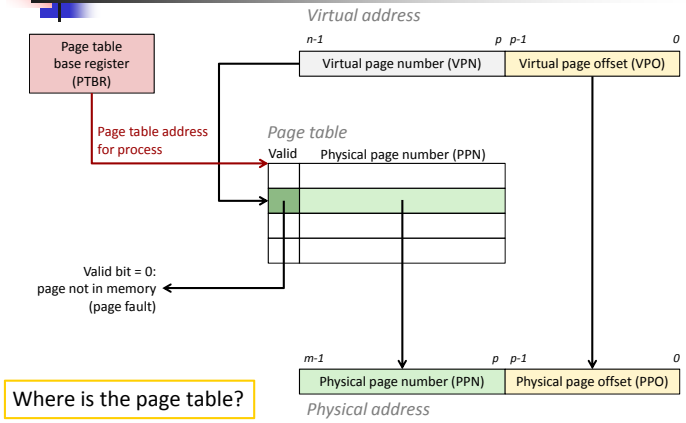
- Virtual address must be translated into physical address for data accesses
- When does it happen?
  - Compile time? Link time? Load time? Runtime?

⇨ It must be fast!

⇨ It must be protected!

19

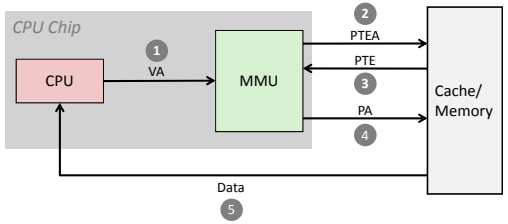
Address Translation With a Page Table



Where is the page table?

20

Address Translation: Page Hit



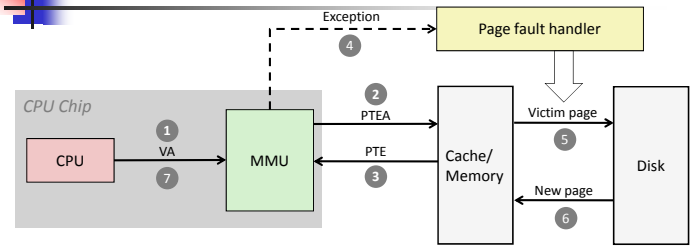
1) Processor sends virtual address to MMU

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

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)

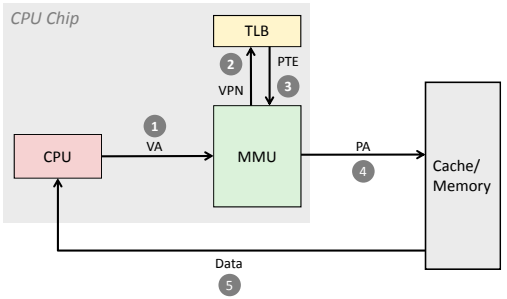
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Speeding up Translation with a TLB

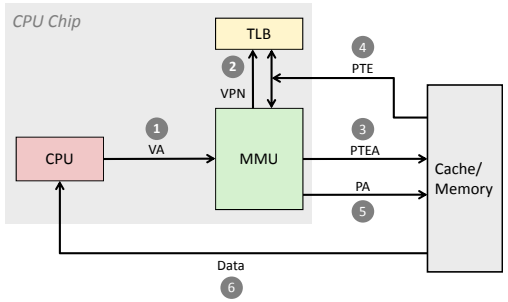
- Page table entries (PTEs) are in memory
  - One access to memory becomes two!
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

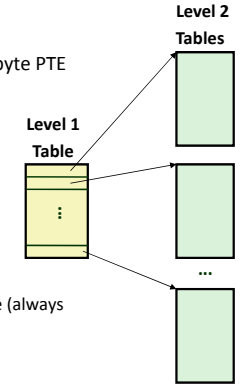


A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare in practice. Why?

25

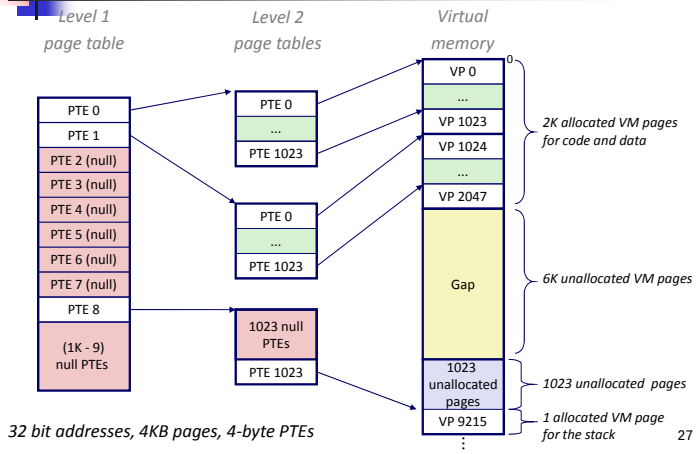
Multi-Level Page Tables

- 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



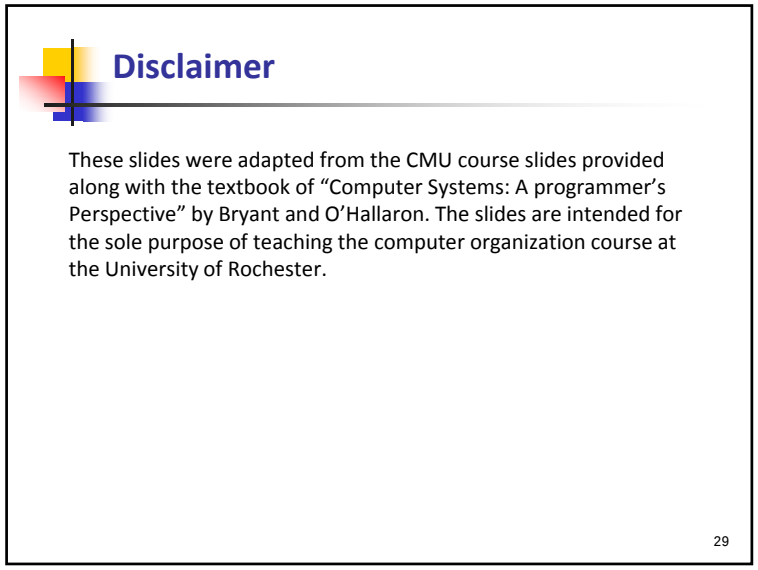
32 bit addresses, 4KB pages, 4-byte PTEs

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