CSCI2467: Systems Programming Concepts

Slideset 10: Virtual Memory

Source: CS:APP Chapter 9, Bryant & O'Hallaron

Course Instructors:

Matthew Toups

Caitlin Boyce

Course Assistants:

Saroj Duwal

David McDonald

Spring 2020

Today

- Class notes

1. Virtual Memory
   - Address spaces
   - VM as a tool for caching
   - VM as a tool for memory management
   - VM as a tool for memory protection
   - How it works: address translation and memory mapping
   - Summary

Addressing Memory

- We know that each byte of memory (RAM) in a computer has an *address*.
- We know that memory at an address can contain code (machine instructions) or data (bytes in some data format)
- We know (from previous class) that both code and data *may* be stored in a cache to speed up future accesses (thanks to locality)
- We've seen this first hand with a running program using GDB

Example using bomblab

```
matoups1@math209:~/2467/bomb3$ objdump -d bomb -M intel

400da7: e8 a0 09 00 00     call   40174c <initialize_bomb>
400dac: bf b8 25 40 00     mov    edi,0x4025b8
400db1: e8 1a fd ff ff     call   400ad0 <puts@plt>
400db6: bf f8 25 40 00     mov    edi,0x4025f8
400dbb: e8 10 fd ff ff     call   400ad0 <puts@plt>
400dc0: e8 e9 07 00 00     call   4015ae <read_line>
400dc5: 48 89 c7           mov    rdi,rax
400dc8: e8 b4 04 00 00     call   401281 <phase_1>
400dcd: e8 4e 06 00 00     call   401420 <phase_defused>
400dd2: bf 28 26 40 00     mov    edi,0x402628
400dd7: e8 f4 fc ff ff     call   400ad0 <puts@plt>

(gdb) x/s 0x402628
0x402628:  "Phase 1 defused.  How about the next one?"
```

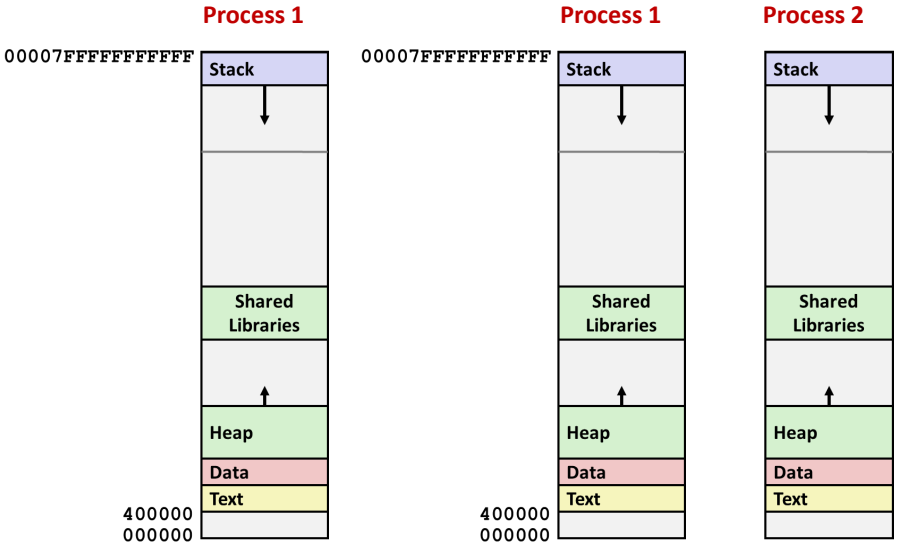
Example using bomblab

```
matoups2@math209:~/2467/bomb46$ objdump -d bomb -M intel
```

```
400da7: e8 71 09 00 00     call   40171d <initialize_bomb>
400dac: bf 98 25 40 00     mov    edi,0x402598
400db1: e8 1a fd ff ff     call   400ad0 <puts@plt>
400db6: bf d8 25 40 00     mov    edi,0x4025d8
400dbb: e8 10 fd ff ff     call   400ad0 <puts@plt>
400dc0: e8 19 08 00 00     call   4015de <read_line>
400dc5: 48 89 c7           mov    rdi,rax
400dc8: e8 e4 04 00 00     call   4012b1 <phase_1>
400dcd: e8 7e 06 00 00     call   401450 <phase_defused>
400dd2: bf 08 26 40 00     mov    edi,0x402608
400dd7: e8 f4 fc ff ff     call   400ad0 <puts@plt>
```

```
(gdb) x/s 0x402608
0x402608:  "Phase 1 defused.  How about the next one?"
```

How does this work?



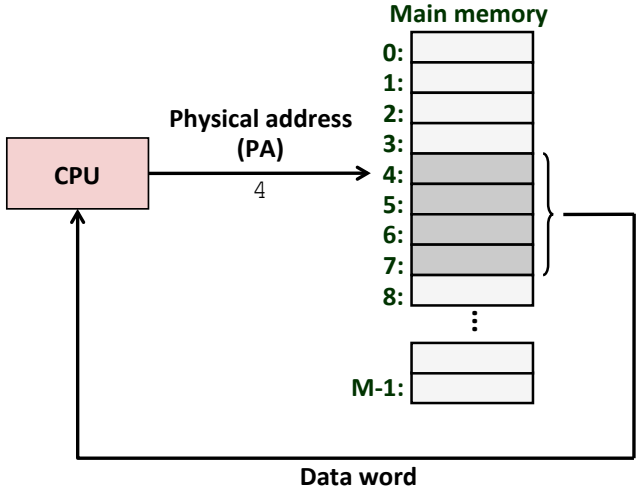
Making processes and address spaces work together

How does this work?

Using a crucial system called *Virtual memory*

Let's introduce VM by starting with what we had before: *physical memory addressing*

A system using *physical* addressing

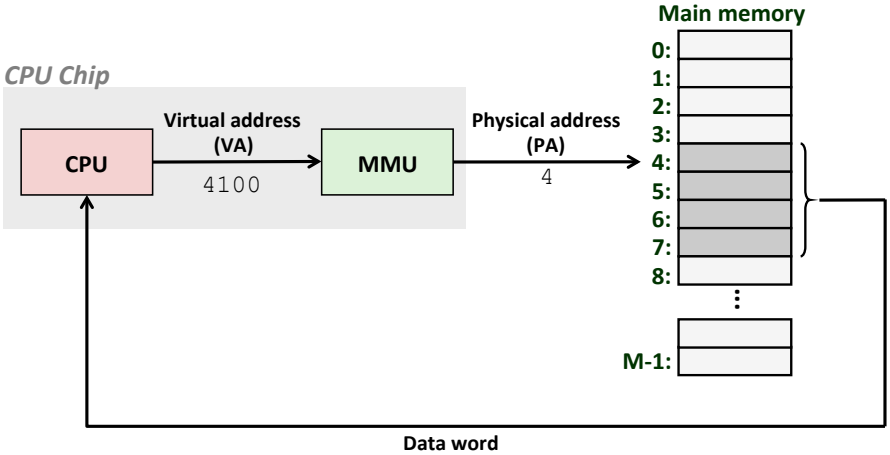


Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames.

A system using *virtual* addressing

Used in all modern servers, laptops, and smart phones

One of the key ideas in computer systems!



Address spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- **Virtual address space:** Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

Why virtual memory (VM)?

- **Uses main memory efficiently**
  - 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 and code

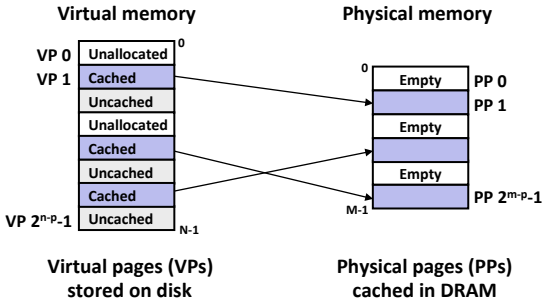
Today

- Class notes

1. Virtual Memory
   - Address spaces
   - VM as a tool for caching
   - VM as a tool for memory management
   - VM as a tool for memory protection
   - How it works: address translation and memory mapping
   - Summary

VM as a tool for caching

- **Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.**
- **The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - These cache blocks are called *pages* (size is P = 2^p bytes)



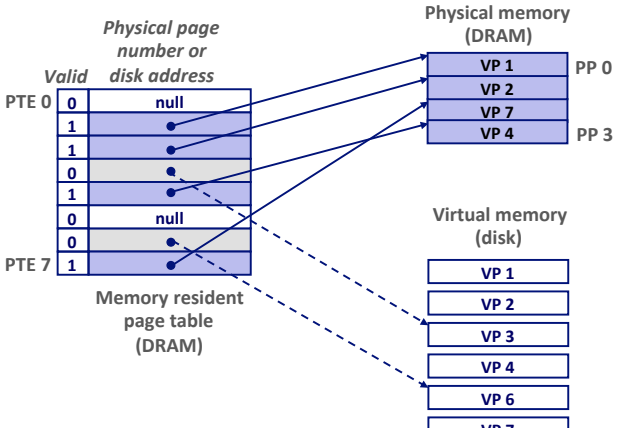
Address spaces

- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- **Consequences**
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

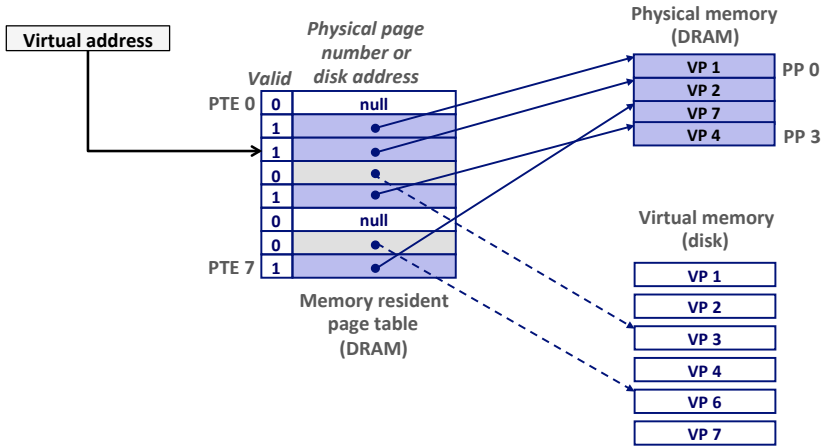
Enabling data structure: page table

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



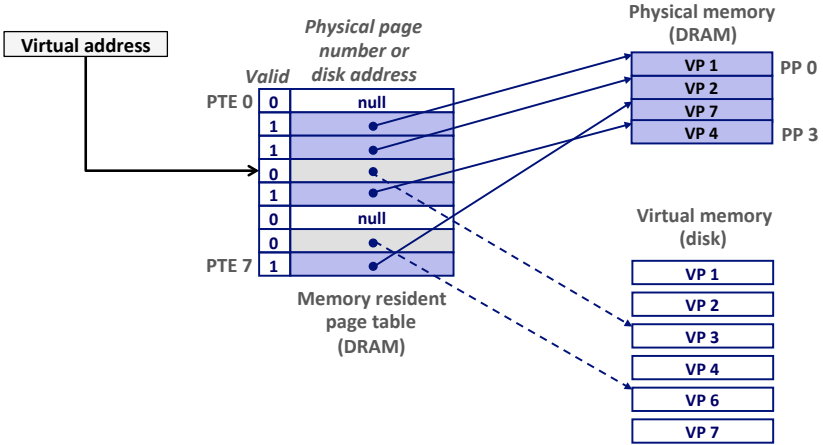
Page hit

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



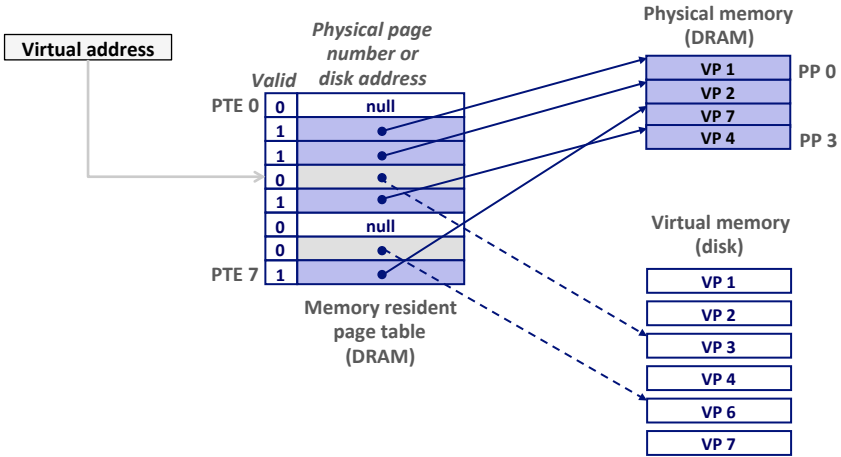
Page fault

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



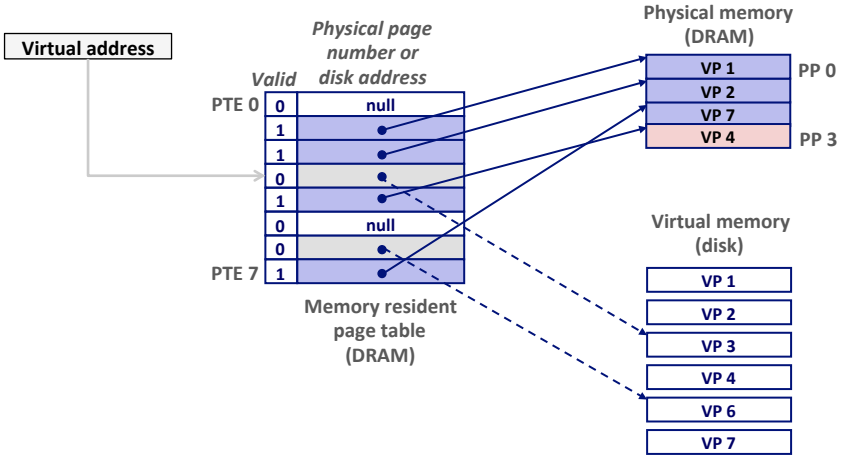
Handling page fault

- Page miss causes page fault (an exception)



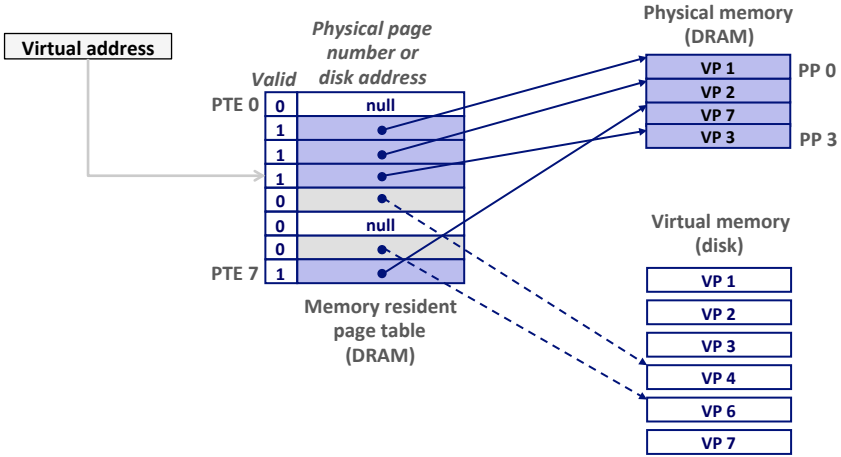
Handling page fault

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



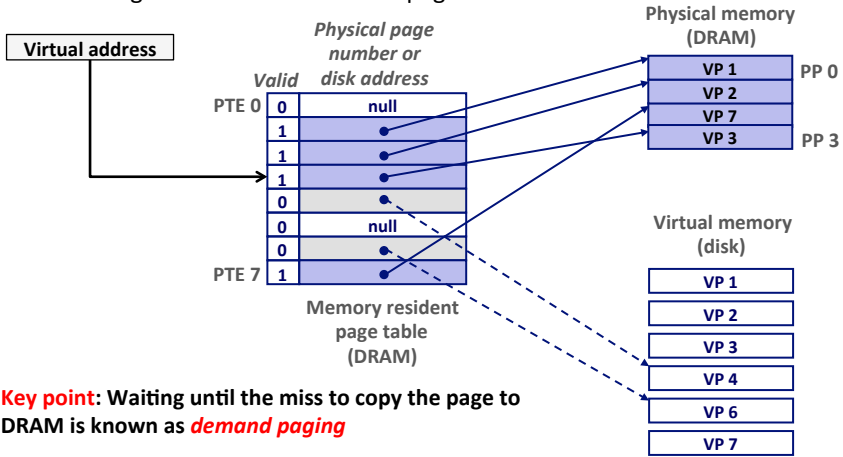
Handling page fault

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



Handling page fault

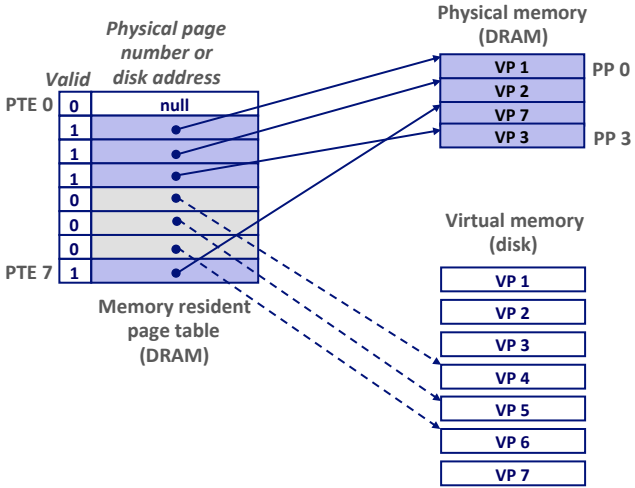
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Allocating pages

- **Allocating a new page (VP 5) of virtual memory.**



Locality to the rescue! (again)

- **Virtual memory seems terribly inefficient, but it 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

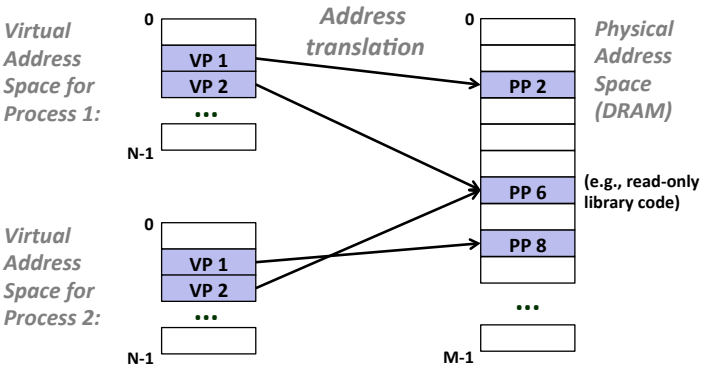
Today

1. Class notes

1. Virtual Memory
   - Address spaces
   - VM as a tool for caching
   - VM as a tool for memory management
   - VM as a tool for memory protection
   - How it works: address translation and memory mapping
   - Summary

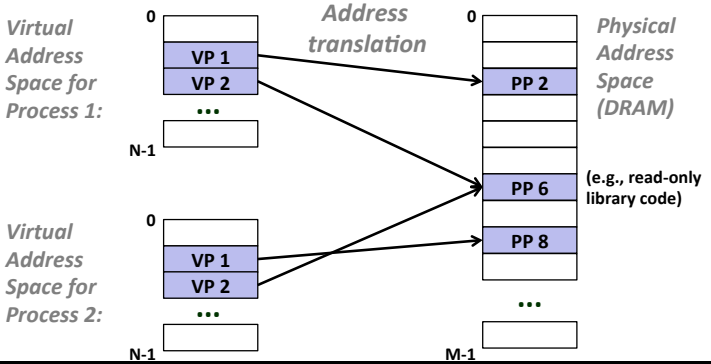
VM as a tool for memory management

- **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
    - Well-chosen mappings can improve locality



VM as a tool for memory management

- **Simplifying memory allocation**
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- **Sharing code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)



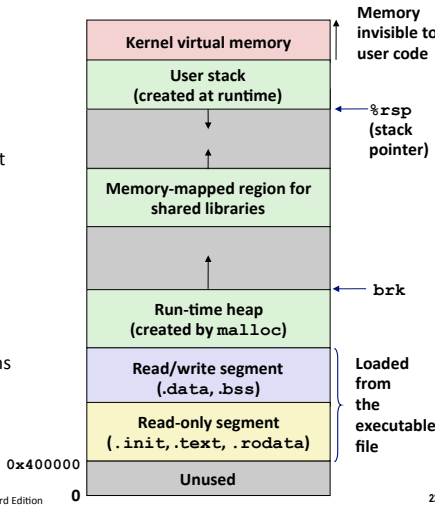
Simplifying linking and loading

Linking

- Each program has similar virtual address space
- Code, data, and heap always start at the same addresses.

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

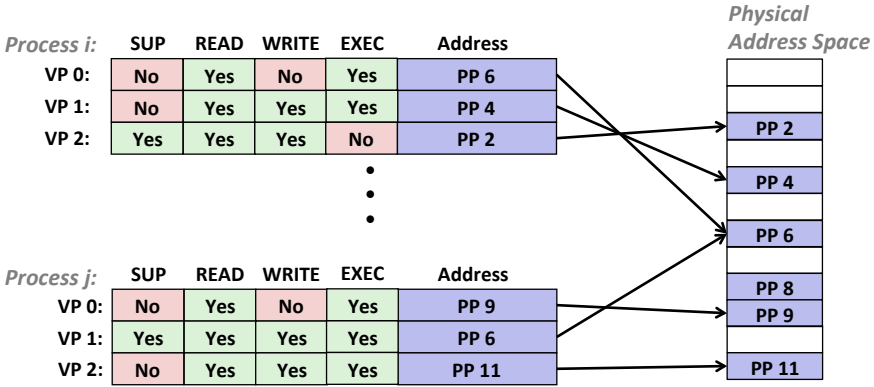


Today

1. Class notes
2. Virtual Memory
   - Address spaces
   - VM as a tool for caching
   - VM as a tool for memory management
   - VM as a tool for memory protection
   - How it works: address translation and memory mapping
   - Summary

VM as a tool for memory protection

- **Extend PTEs with permission bits**
- **MMU checks these bits on each access**

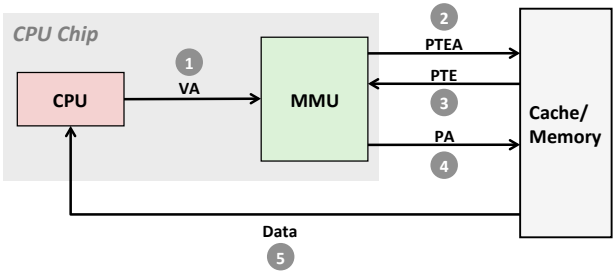


Today

- Class notes

1. Virtual Memory
   - Address spaces
   - VM as a tool for caching
   - VM as a tool for memory management
   - VM as a tool for memory protection
   - How it works: address translation and memory mapping
   - Summary

Address translation: page hit



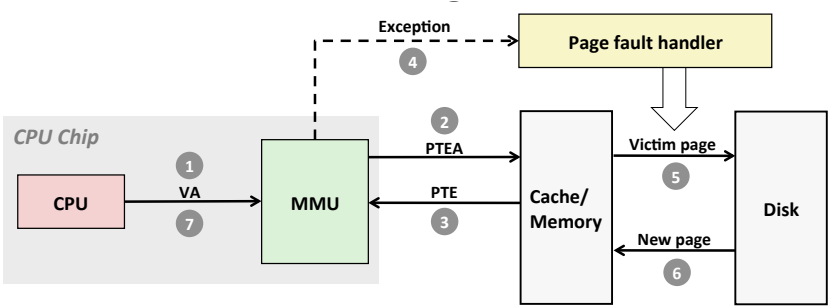
1) Processor sends virtual address to MMU

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

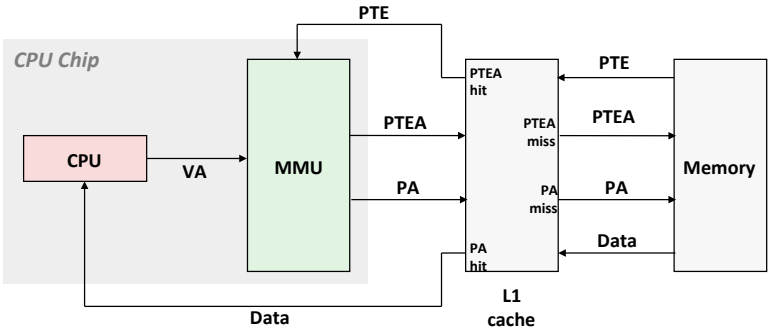
4) Valid bit is zero, so MMU triggers page fault exception

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

Integrating VM and cache

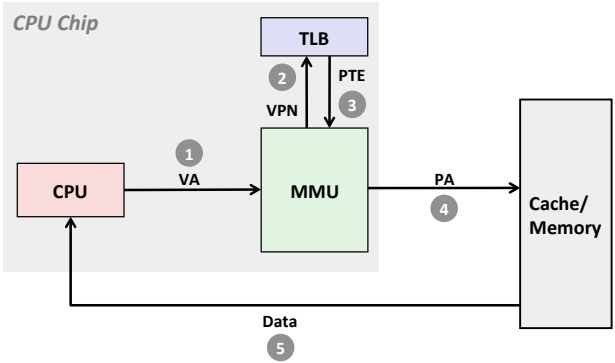


VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up translation with a TLB

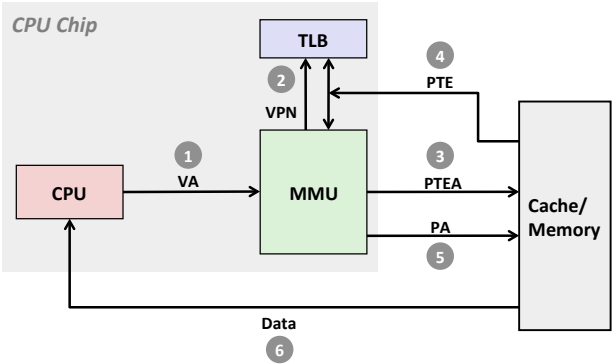
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to 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. Why?

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