Virtual Memory: Concepts

15-213 / 18-213: Introduction to Computer Systems
16th Lecture, Mar. 20, 2012

Instructors:
Todd C. Mowry & Anthony Rowe

Today

- **Address spaces**
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

Recall: Byte-Oriented Memory Organization

- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Recall: Simple Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address

```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

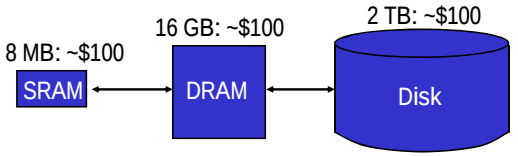
```
movl 8(%ebp),%edx
```

Why Virtual Memory?

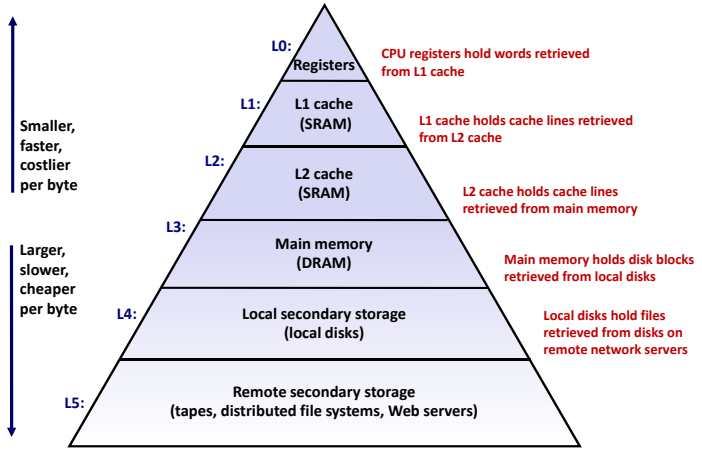
- **There are three motivations for Virtual Memory (VM):**
  1. Allow main memory (DRAM) to act as a "cache" for disk
  2. Simplifying memory management
  3. Protecting address spaces
- **But VM works very differently from SRAM caches. Why?**
  - To understand why, let's begin with the first motivation
  - (Once we understand that, the other aspects of VM will make more sense.)

Motivation 1: DRAM a "Cache" for Disk

- **The full address space is quite large:**
  - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
  - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- **Disk storage is ~100X cheaper than DRAM storage**
  - 2 TB of DRAM: ~ $10,000
  - 2 TB of disk: ~ $100
- **To access very large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk**

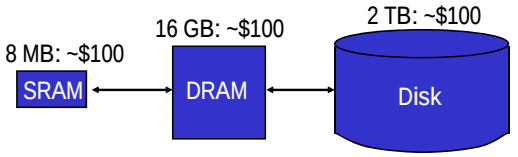


An Example Memory Hierarchy (Review)



DRAM vs. SRAM as a "Cache"

- **DRAM vs. disk is more extreme than SRAM vs. DRAM**
  - access latencies:
    - DRAM is ~10X slower than SRAM
    - disk is ~100,000X slower than DRAM
  - importance of exploiting spatial locality:
    - first byte is ~100,000X slower than successive bytes on disk
      - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
  - "cache" size:
    - main memory is ~1000X larger than an SRAM cache
  - addressing for disk is based on sector address, not memory address



Impact of These Properties on Design

- **If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?**
  - Line size?
  - Associativity?
  - Replacement policy (if associative)?
  - Write through or write back?
- **What would the impact of these choices be on:**
  - miss rate
  - hit time
  - miss latency
  - tag overhead

Locating an Object in a "Cache"

1. Search for matching tag

- SRAM cache

Object Name

X

= *X?*

"Cache"

	Tag	Data
0:	D	243
1:	X	17
N-1:	J	105

2. Use indirection to look up actual object location

- Virtual Memory

Object Name

X

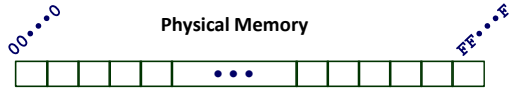
Lookup Table

	Location
D:	0
J:	N-1
X:	1

"Cache"

	Data
0:	243
1:	17
N-1:	105

Motivations for VM Revisited



- **#1: How does everything fit?** ***(We just talked about this.)***
  - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
  - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- **#2: How to decide which memory to use in your program?**
  - What about after a fork()?
- **#3: What if another process stores data into your memory?**
  - How could you debug your program?

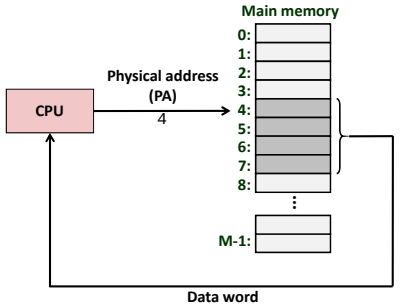
Indirection solves all three problems

- **Each process gets its own private image of memory**
  - appears to be a full-sized private memory range
- **This fixes "how to choose" and "others shouldn't mess w/yours"**
  - in addition to "making everything fit"
- **Implementation: translate addresses transparently**
  - add a mapping function
    - to map private (i.e. "virtual") addresses to physical addresses
  - do the mapping on every load or store
- **This mapping trick is the heart of *virtual memory***

Address Spaces

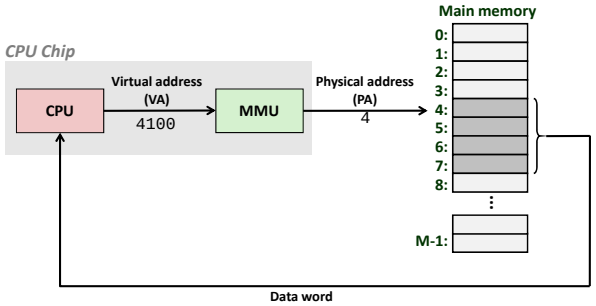
- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 … }
- **Virtual address space:** Set of N = 2^n virtual addresses
  {0, 1, 2, 3, …, N-1}
- **Physical address space:** Set of M = 2^m physical addresses
  {0, 1, 2, 3, …, M-1}
- **Clean distinction between data (bytes) and their attributes (addresses)**
- **Each datum can now have multiple addresses**
- **Every byte in main memory:**
  one physical address, one (or more) virtual addresses

A System Using Physical Addressing



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

A System Using Virtual Addressing



- **Used in all modern servers, desktops, and laptops**
- **One of the great ideas in computer science**

Why Virtual Memory? (Further Details)

(1) VM allows efficient use of limited main memory (RAM)

- Use RAM as a cache for the parts of a virtual address space
  - some non-cached parts stored on disk
  - some (unallocated) non-cached parts stored nowhere
- Keep only active areas of virtual address space in memory
  - transfer data back and forth as needed

(2) VM simplifies memory management for programmers

- Each process gets a full, private linear address space

(3) VM isolates address spaces

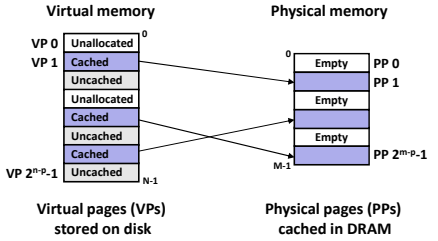
- One process can't interfere with another's memory
  - because they operate in different address spaces
- User process cannot access privileged information
  - different sections of address spaces have different permissions

Today

- Address spaces
- **(1) VM as a tool for caching**
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation

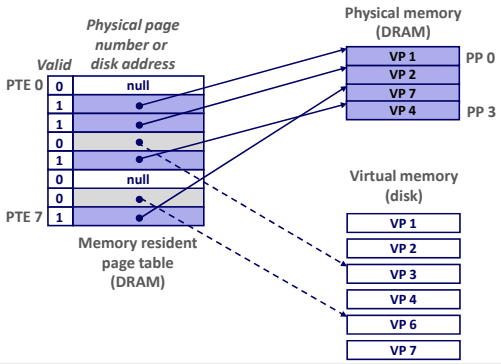
(1) VM as a Tool for Caching

- ***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)



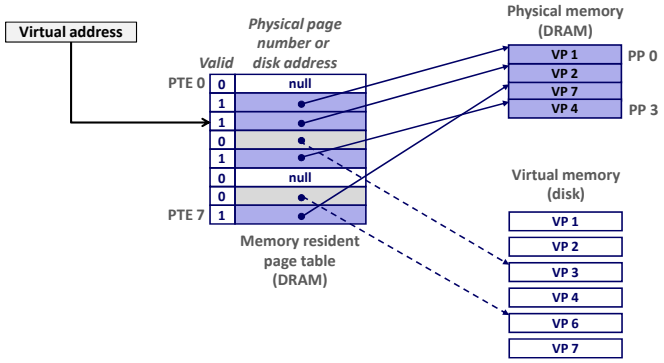
Enabling data structure: Page Table

- **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



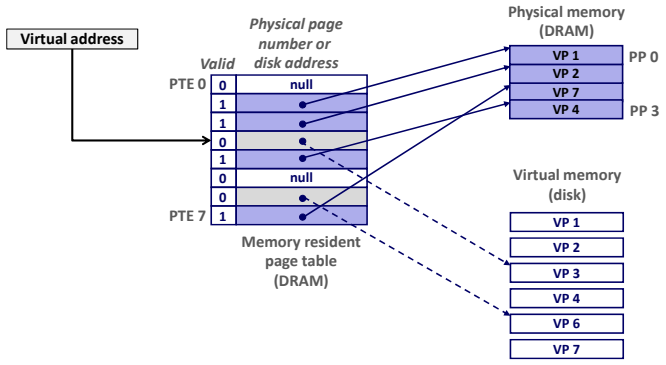
Page Hit

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



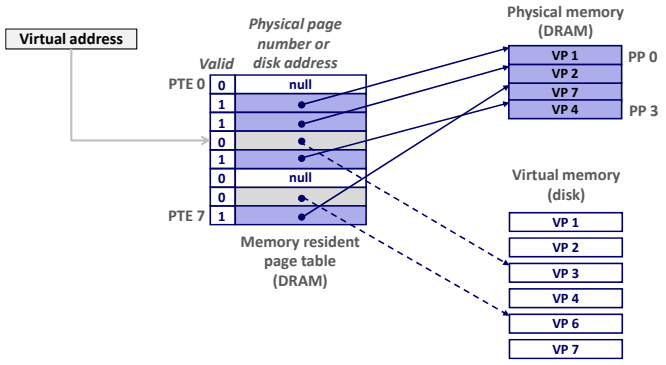
Page Fault

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



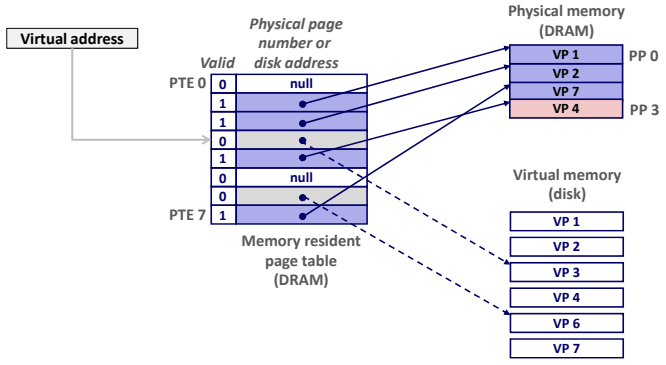
Handling Page Fault

- Page miss causes page fault (an exception)



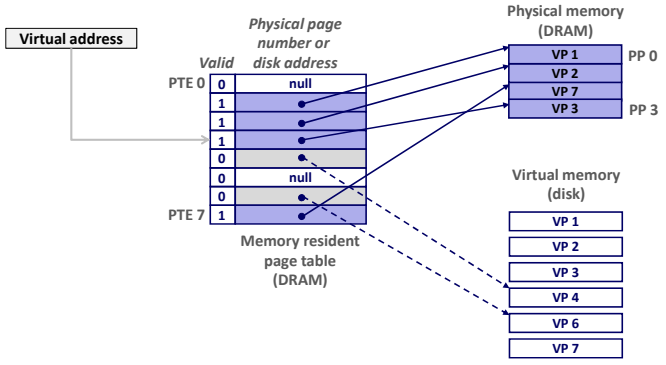
Handling Page Fault

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



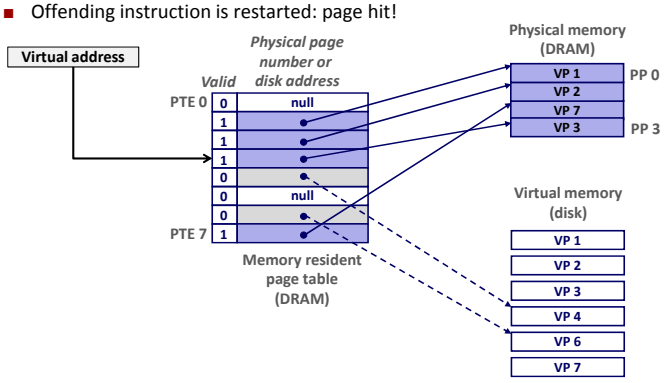
Handling Page Fault

- 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!



Locality to the Rescue Again!

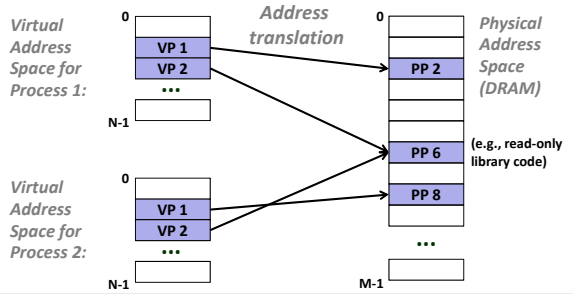
- **Virtual memory works because of locality**
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets
- **If (working set size < main memory size)**
  - Good performance for one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

Today

- Address spaces
- (1) VM as a tool for caching
- **(2) VM as a tool for memory management**
- (3) VM as a tool for memory protection
- Address translation

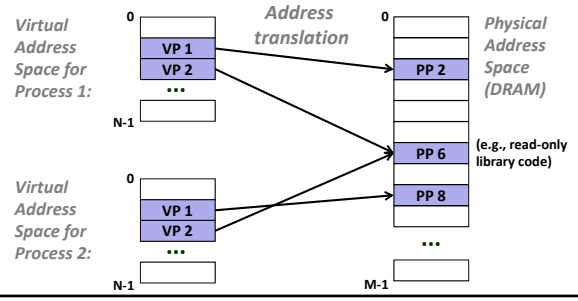
(2) VM as a Tool for Memory Management

- **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 simplify memory allocation and management



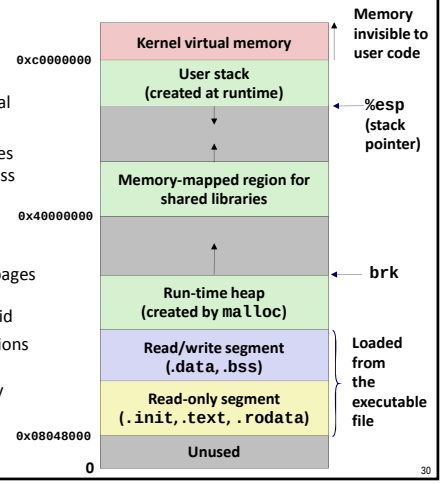
Simplifying allocation and sharing

- **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 multiple virtual pages to the same physical page (here: PP 6)



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

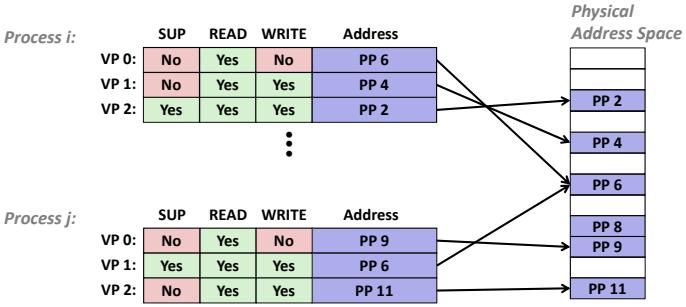


Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- **(3) VM as a tool for memory protection**
- Address translation

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)



Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- **Address translation**

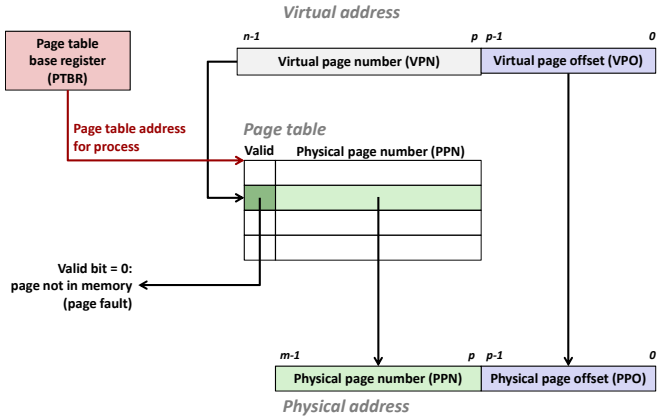
VM Address Translation

- **Virtual Address Space**
  - $V = \{0, 1, \ldots, N-1\}$
- **Physical Address Space**
  - $P = \{0, 1, \ldots, M-1\}$
- **Address Translation**
  - ***MAP:*** $V \rightarrow P \cup \{\varnothing\}$
  - For virtual address ***a***:
    - ***MAP(a)*** $= a'$ if data at virtual address ***a*** is at physical address ***a'*** in ***P***
    - ***MAP(a)*** $= \varnothing$ if data at virtual address ***a*** is not in physical memory
      - Either invalid or stored on disk

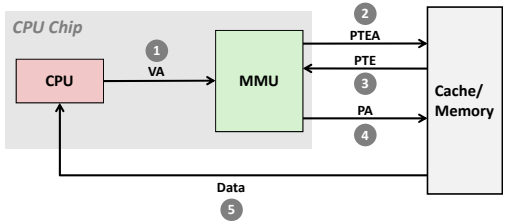
Summary of Address Translation Symbols

- **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

Address Translation With a Page Table



Address Translation: Page Hit



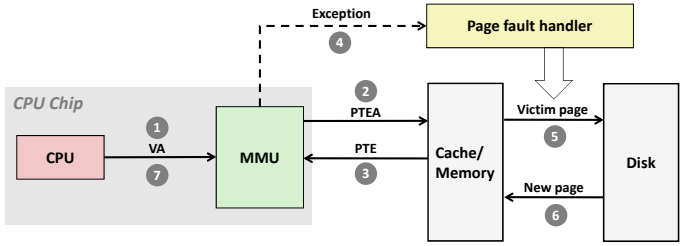
1) Processor sends virtual address to MMU

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

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

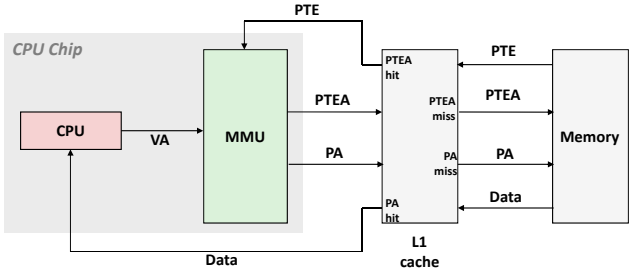
7) Handler returns to original process, restarting faulting instruction

Views of virtual memory

- **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

Integrating VM and Cache

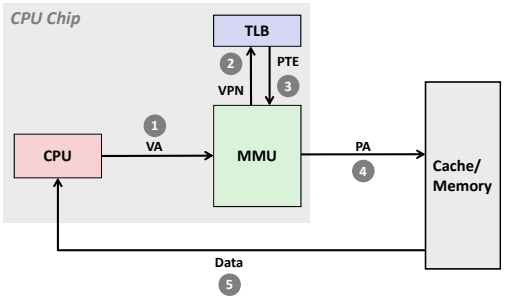


VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

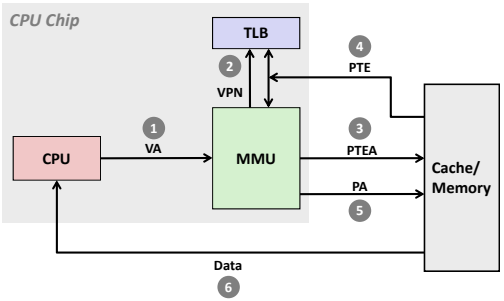
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page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