Computer Organization and Architecture

Virtual Memory

Jeongseob Ahn

Department of Software & Computer Engineering
Ajou University

Adapted slides from CIS501@Upenn, ECE447@CMU, 6.823@MIT

Memory system in 1950's

- lw R1, 4(R3) // R1 <- M[R3+4]

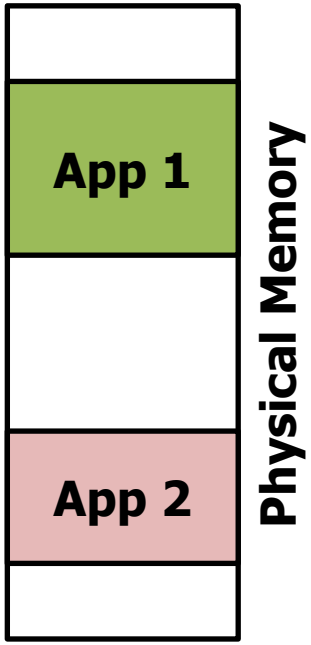
What is this address?
(called effective address)

- EDSAC, early 50's
  - Effective address = physical memory address

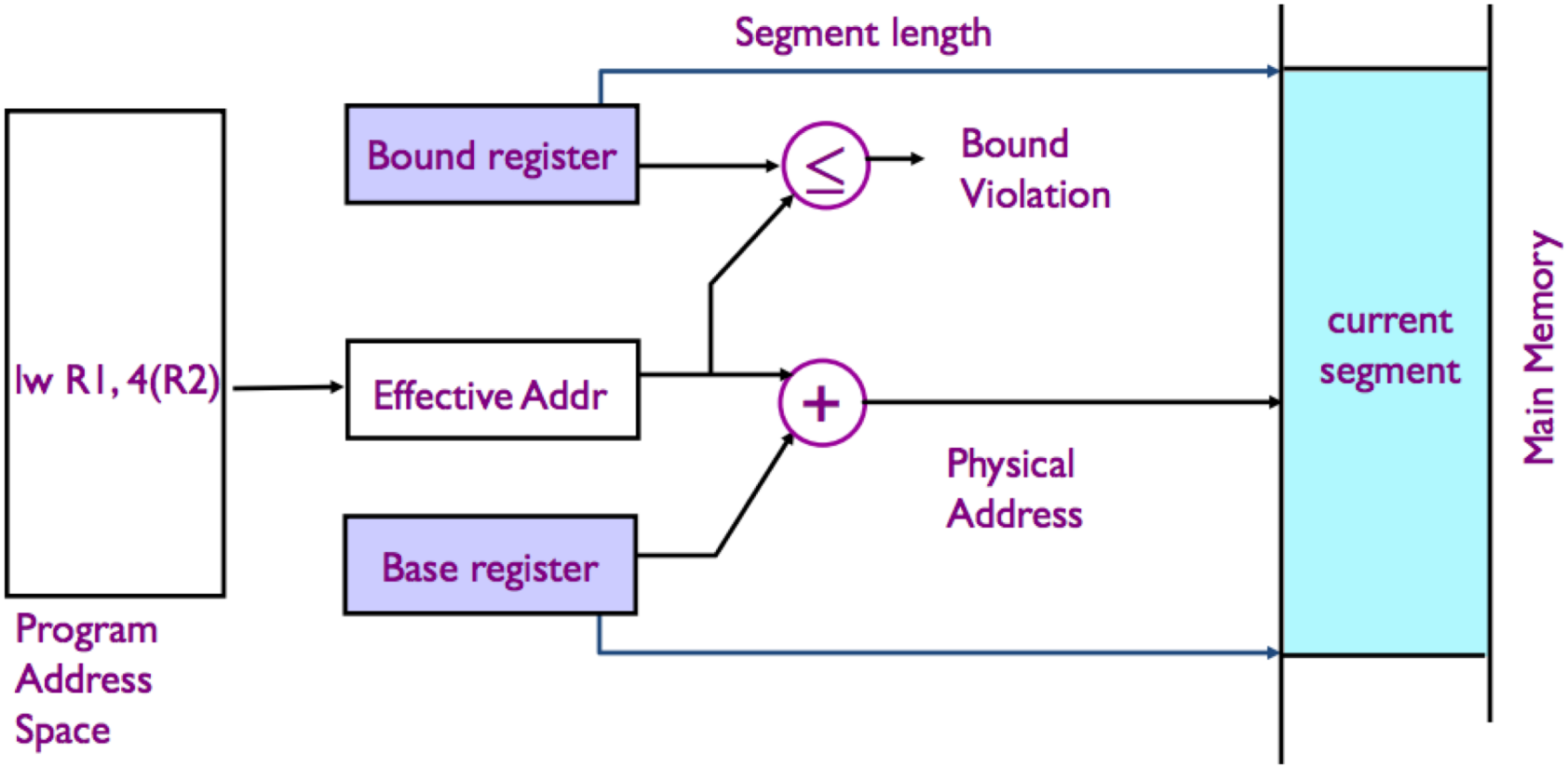
- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
  - Single-programmed execution environment

Multiprogramming

- Motivation
  - In the early machines, I/O operations were slow and each word transferred involved the CPU
  - Higher throughput if CPU and I/O of 2 or more programs were overlapped
  - Why?
    - Multiprogramming
- Location independent programs
  - Programming and storage management ease
    - Need for a base register
- Protection
  - Independent programs should not affect each other inadvertently
    - Need for a bound register

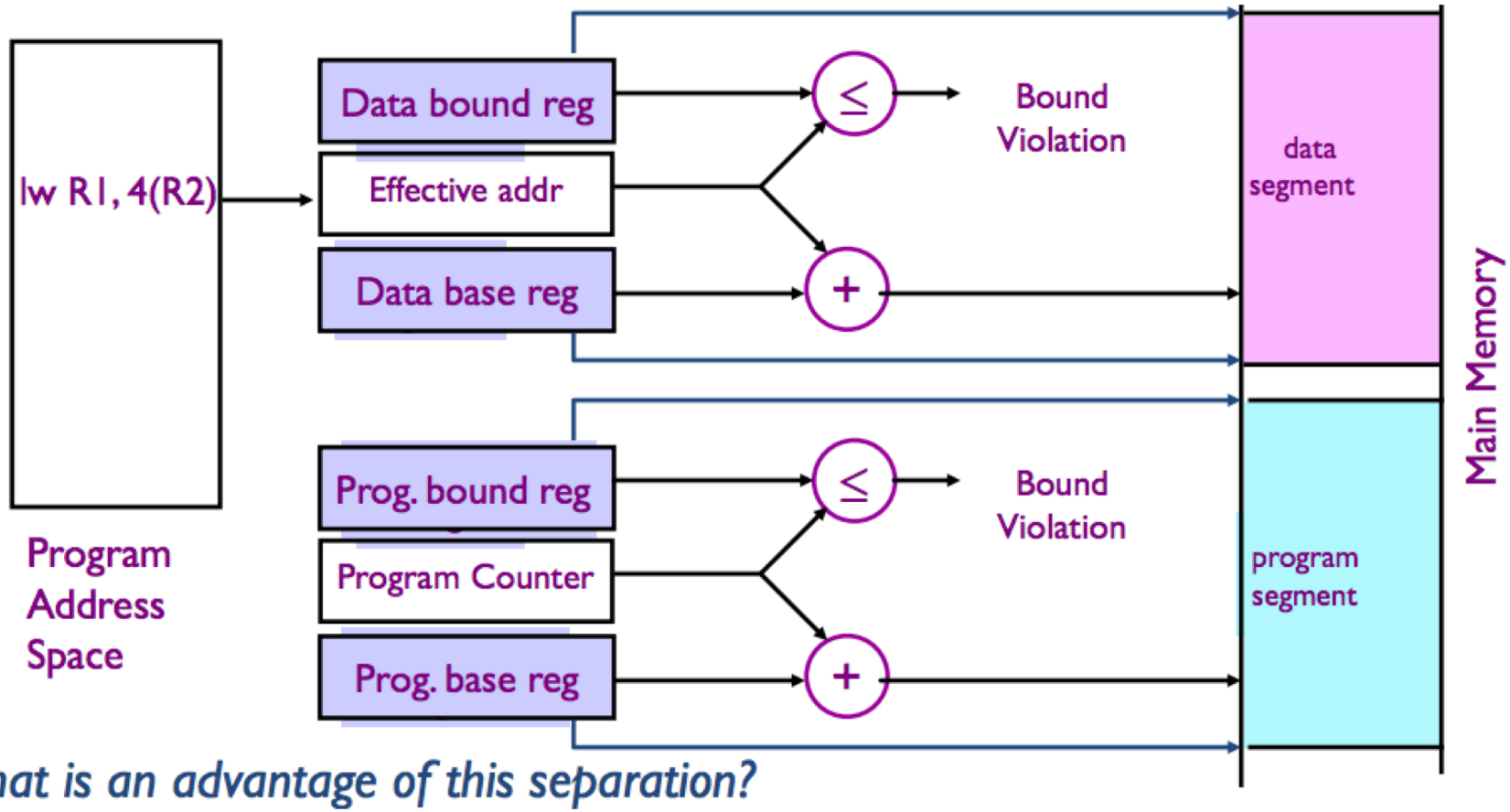


Simple Base and Bound Translation



- Base and bounds registers only visible/accessible when processor running in kernel (a.k.a supervisor) mode

Separate Areas for Program and Data

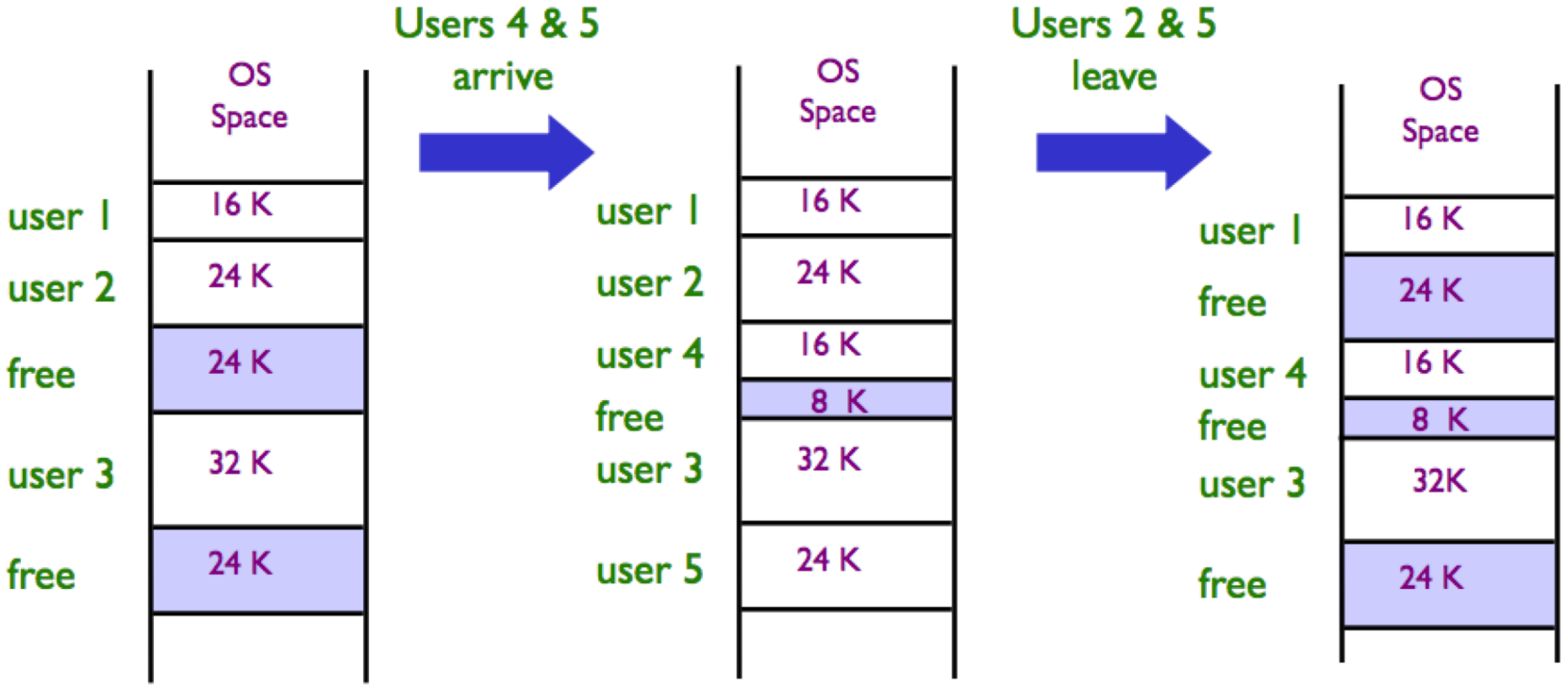


What is an advantage of this separation?

→ **Permit sharing of program segments**

Memory Fragmentation

- What is the problem with segmentation? Fragmentation!



- As users come and go, the storage is "fragmented".
  - Therefore, at some stage programs have to be moved around to compact the storage.

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
  - **Goal: each application thinks it has infinite memory**
- One app may want more memory than is in the system
  - App's insn/data footprint may be larger than main memory
- Computer system (OS and HW) automatically manages the physical memory space transparently to the programmer
  - Programmers do not need to know the physical size of memory nor manage it
  - More complex system software and architecture

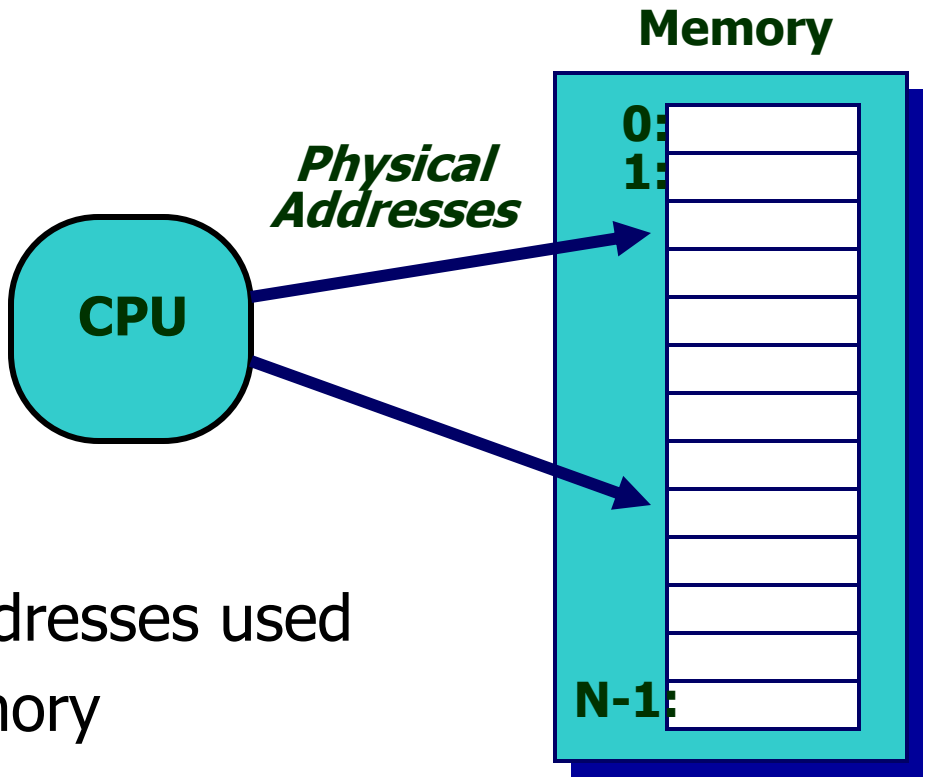
Benefits of Virtual Memory

- Programmer does not need to deal with physical addresses
- Each process has its own mapping from virtual → physical addresses

- Enables
  - Code and data to be located anywhere in physical memory
  - Isolation/separation of code and data of different processes in physical processes
  - Code and data sharing between multiple processes

A System with Physical Memory Only

- Examples:
  - early PCs



CPU's load or store addresses used directly to access memory

The Problem

- Physical memory is not infinite
  - Should the programmer be concerned about the size of code/data blocks fitting physical memory?
  - Should the programmer ensure two processes do not use the same physical memory?

Difficulties of Direct Physical Addressing

- Programmer needs to manage physical memory space
  - Inconvenient & hard
  - Harder when you have multiple processes
- Difficult to support code and data relocation
- Difficult to support multiple processes
  - Protection and isolation between multiple processes
- Difficult to support data/code sharing across processes

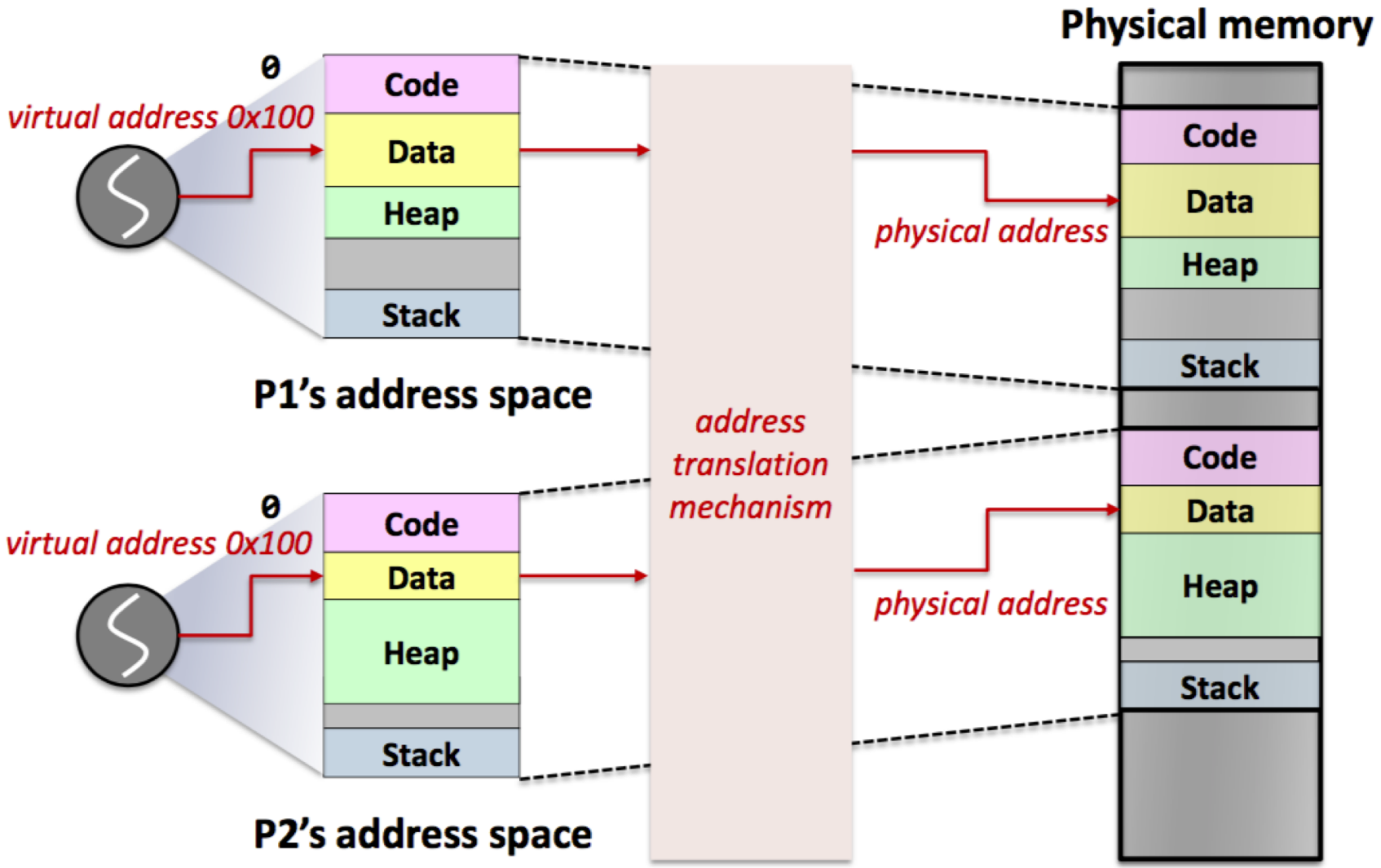
Virtual Memory

- Idea: Give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory
- Programmer can assume he/she has "infinite" amount of physical memory
- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

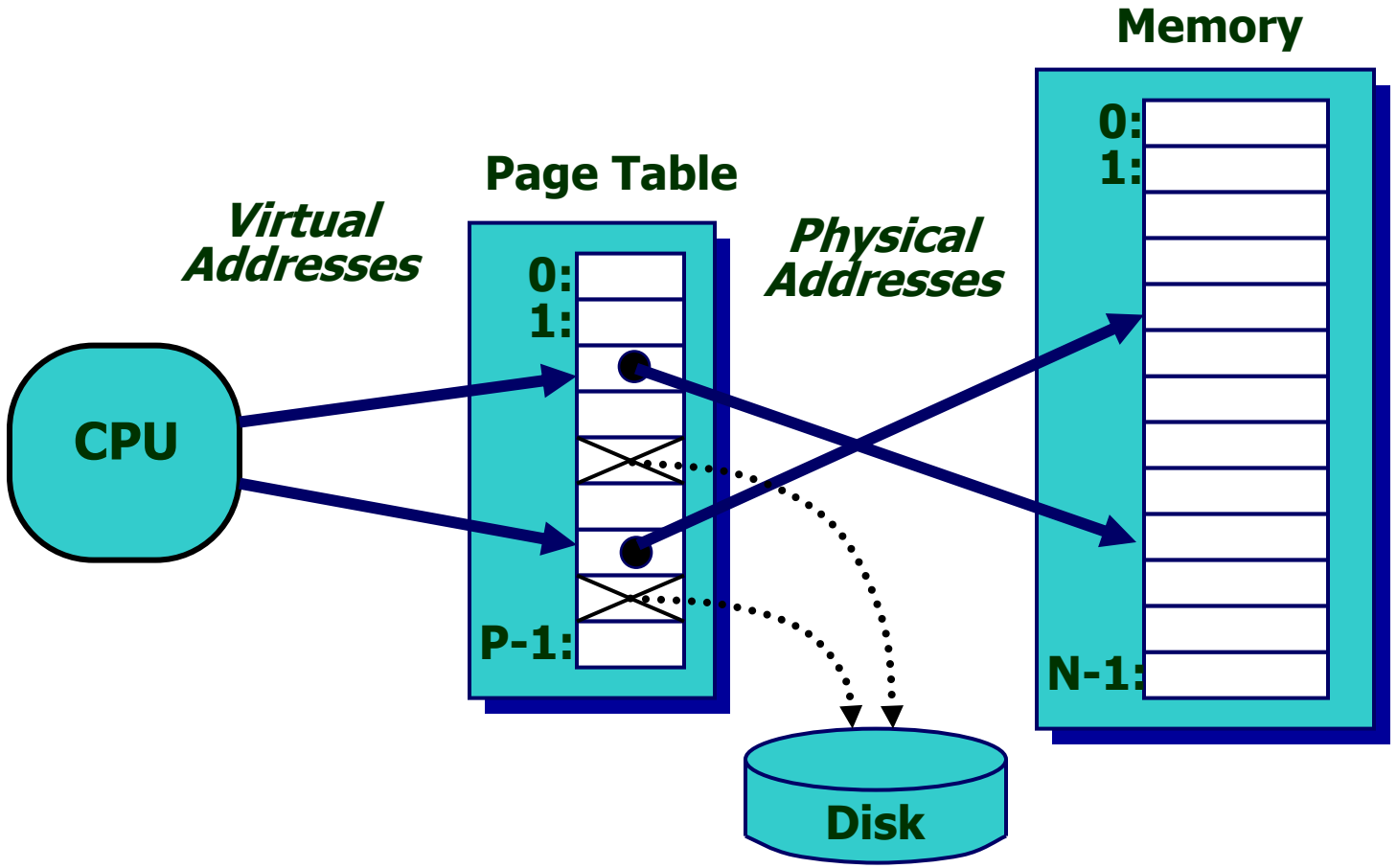
Virtual Memory

- Level of indirection
- Address generated by each instruction in a program is a **virtual addresses (VAs)**
- Memory accessed using **physical addresses (PAs)**
- VAs translated to PAs at some coarse granularity (page)
- OS controls VA to PA mapping for itself and all other processes

Virtual Memory



A System with Virtual Memory (Page based)



- Address Translation: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

Demand Paging

- A virtual page is mapped to
  - A physical page if the page is in the physical memory
  - A location in disk, otherwise
- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical page and adjust the mapping
- Page table is the table that stores the mapping of virtual pages to physical pages

Physical Memory as a Cache

- In other words…
- Physical memory is a cache for pages stored on disk
  - In fact, it is a fully associative cache in modern systems (a virtual page can be mapped to any physical frame)
- Similar caching issues exist as we have covered earlier:
  - Placement: where and how to place/find a page in memory?
  - Replacement: what page to remove to make room in memory?
  - Granularity of management: large, small, uniform pages?
  - Write policy: what do we do about writes? Write back?

Supporting Virtual Memory

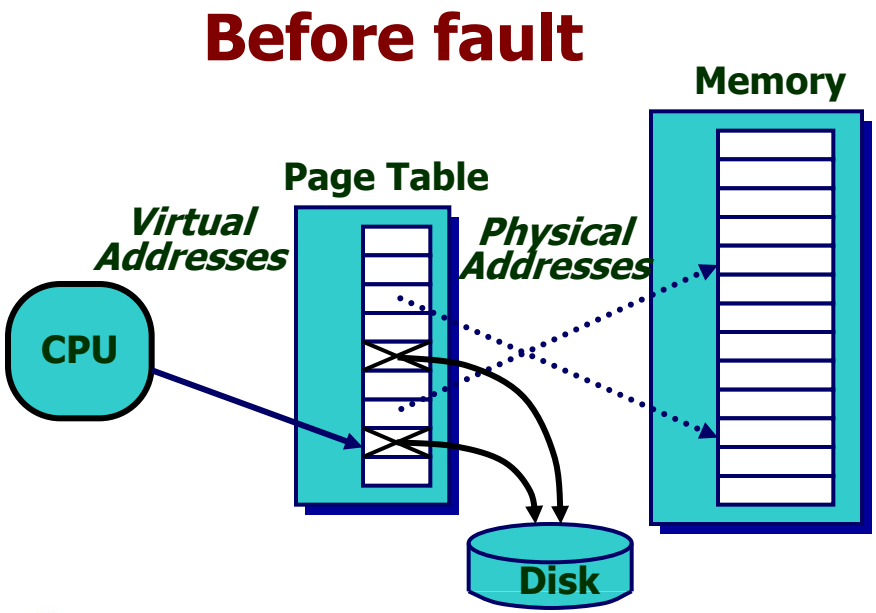
- Virtual memory requires both HW + SW support
  - Page table is in memory
  - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)
- The hardware component is called the MMU (Memory Management Unit)
  - Includes Page Table Base Register(s), TLBs, page walkers
- It is the job of the OS to leverage the MMU to
  - Populate page tables, decide what to replace in physical memory
  - Change the Page Table Register on context switch (to use the running thread's page table)
  - Handle page faults and ensure correct mapping

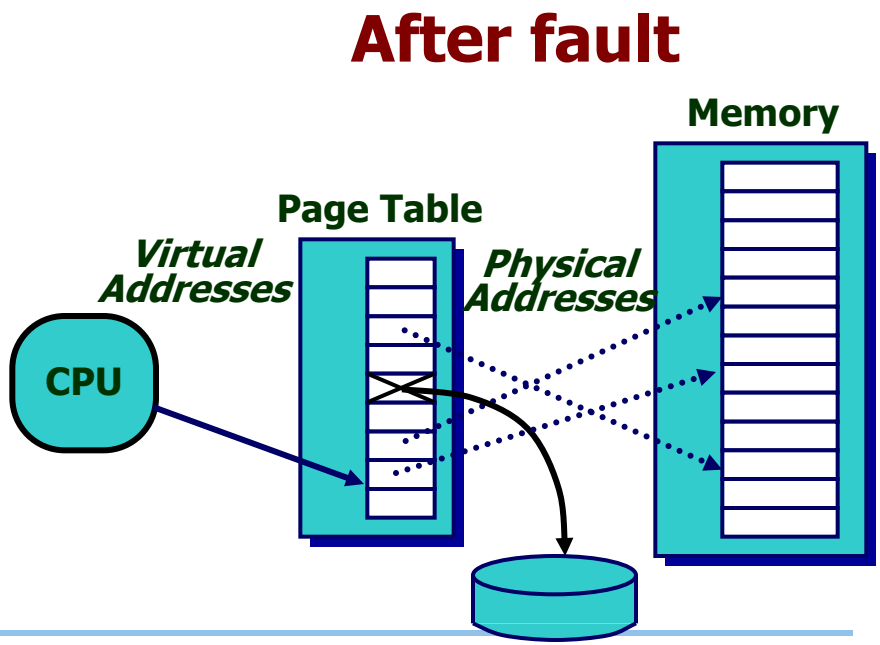
Some OS Jobs for VM

- Keeping track of which physical pages are free
- Allocating free physical pages to virtual pages
- Page replacement policy
  - When no physical page is free, what should be swapped out?
- Sharing pages between processes
  - Copy-on-write optimization

Page Fault: A Miss in Physical Memory

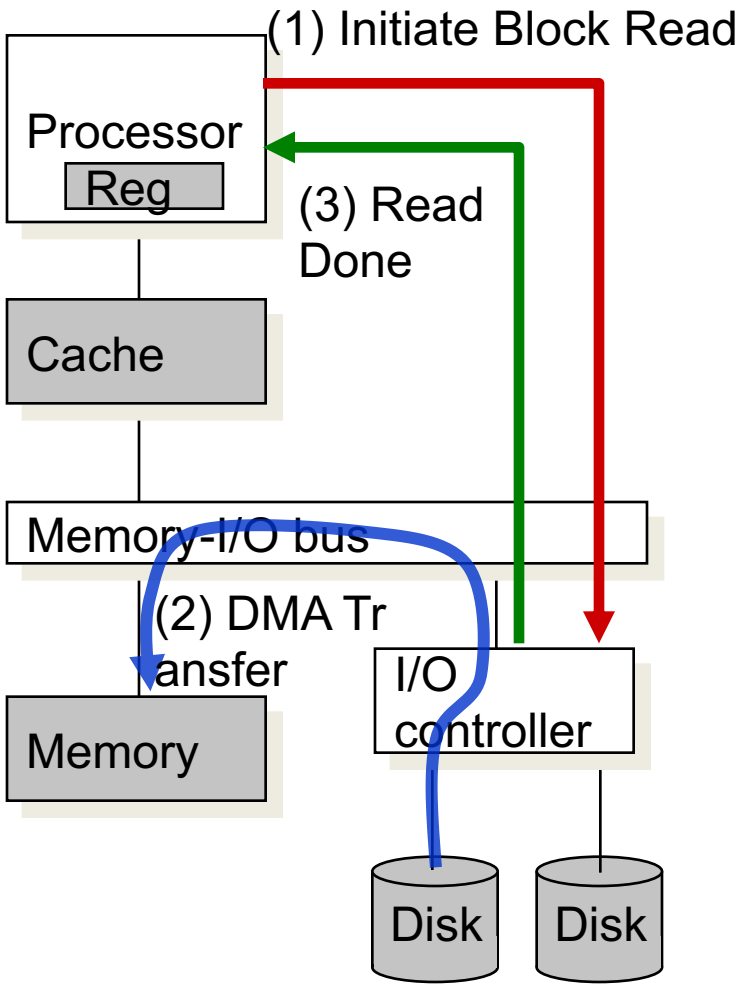
- If a page is not in physical memory but disk
  - Page table entry indicates that the page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Other processes can continue executing
    - OS has full control over placement





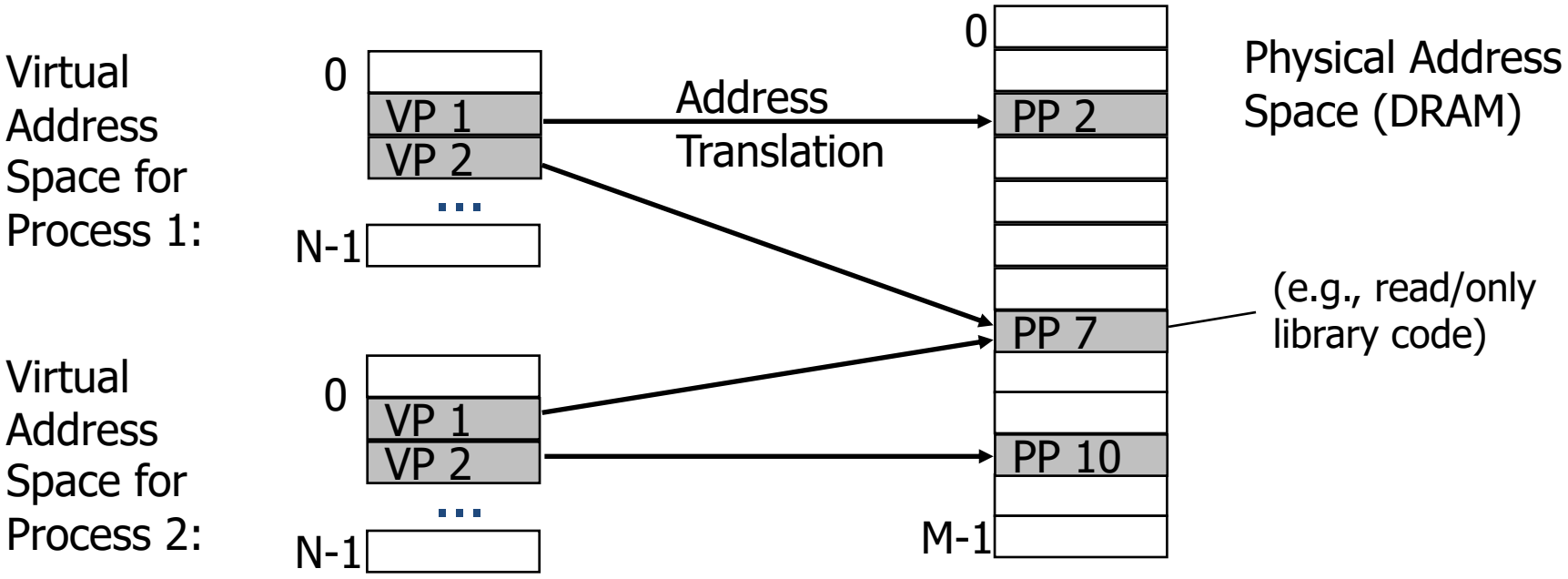
Servicing a Page Fault

- (1) OS makes the I/O request
  - Read block of length P starting at disk address X and store starting at memory address Y
- (2) Read occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- (3) Controller signals completion
  - Interrupt processor
  - OS resumes suspended process



Page Table is Per Process

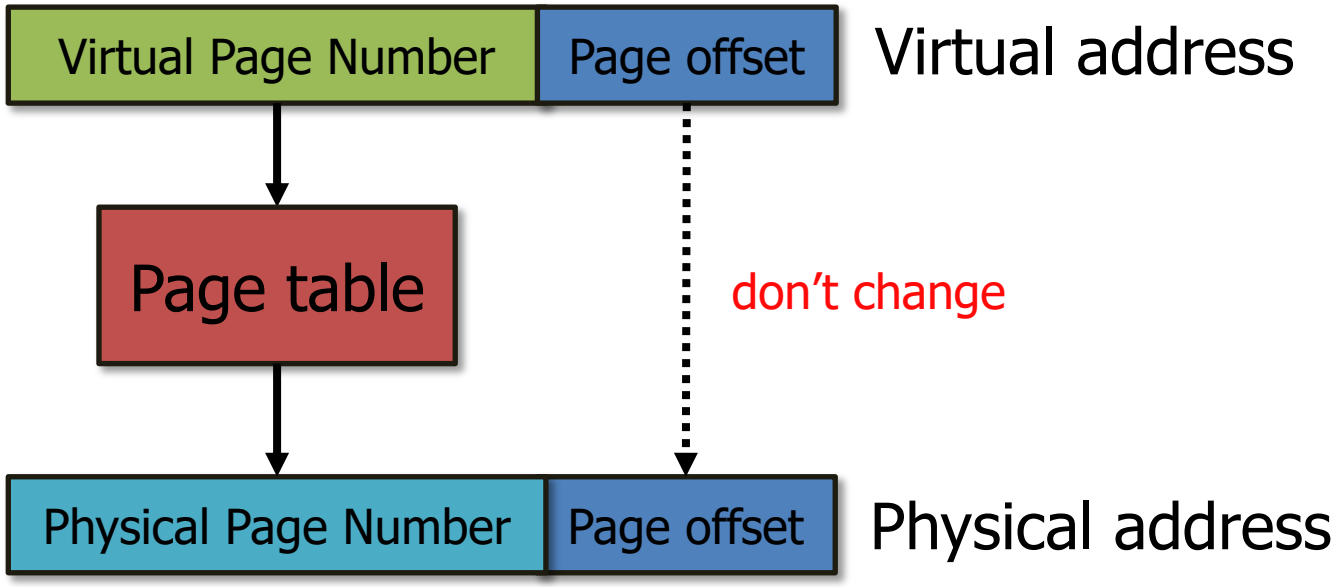
- Each process has its own virtual address space
  - Full address space for each program
  - Simplifies memory allocation, sharing, linking and loading.



Address Translation

- How to obtain the physical address from a virtual address?
- Page size specified by the ISA
  - Today: 4KB, 2MB, 1GB in x86
  - Trade-offs? (remember cache lectures)
- Page table contains an entry for each virtual page
  - Called Page Table Entry

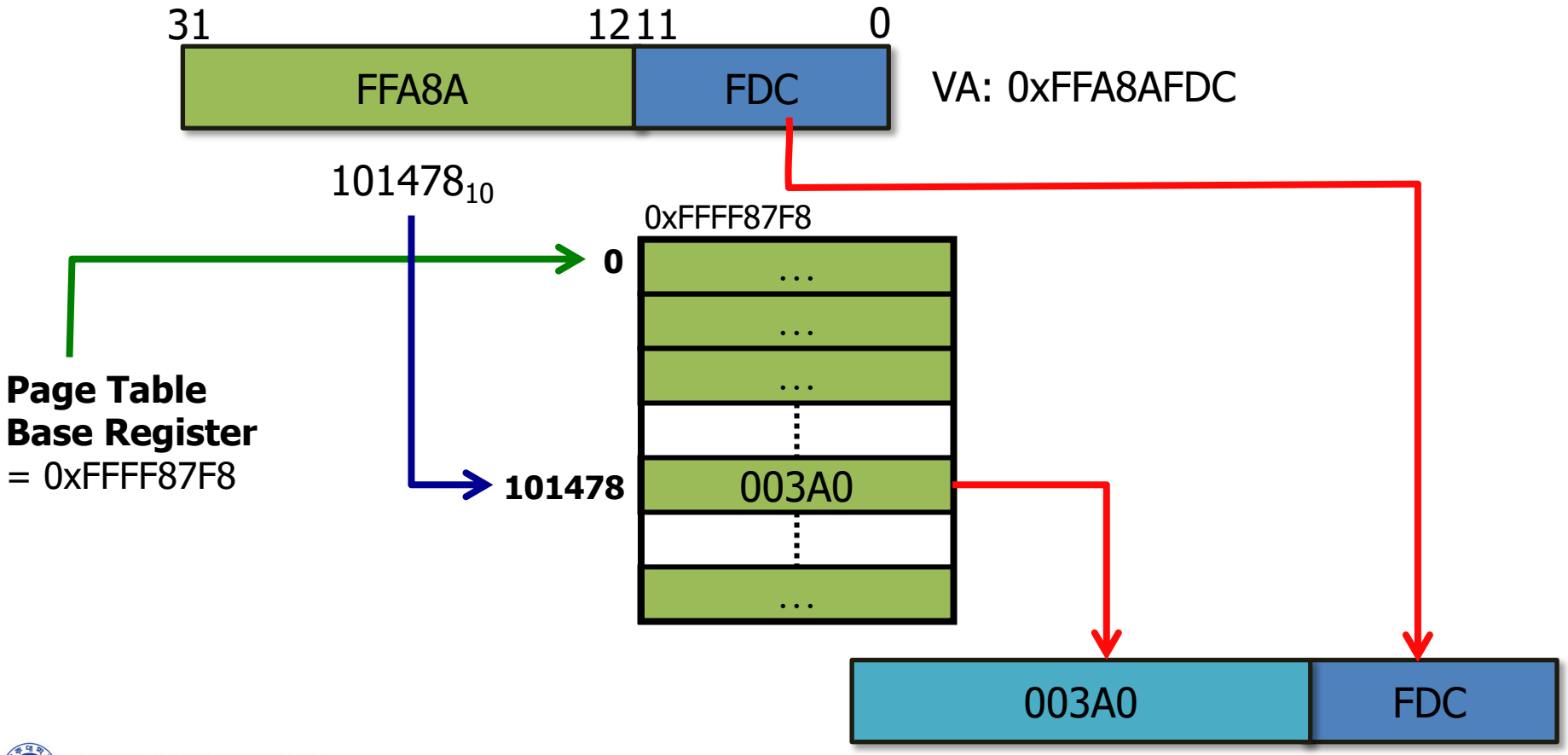
Address Translation



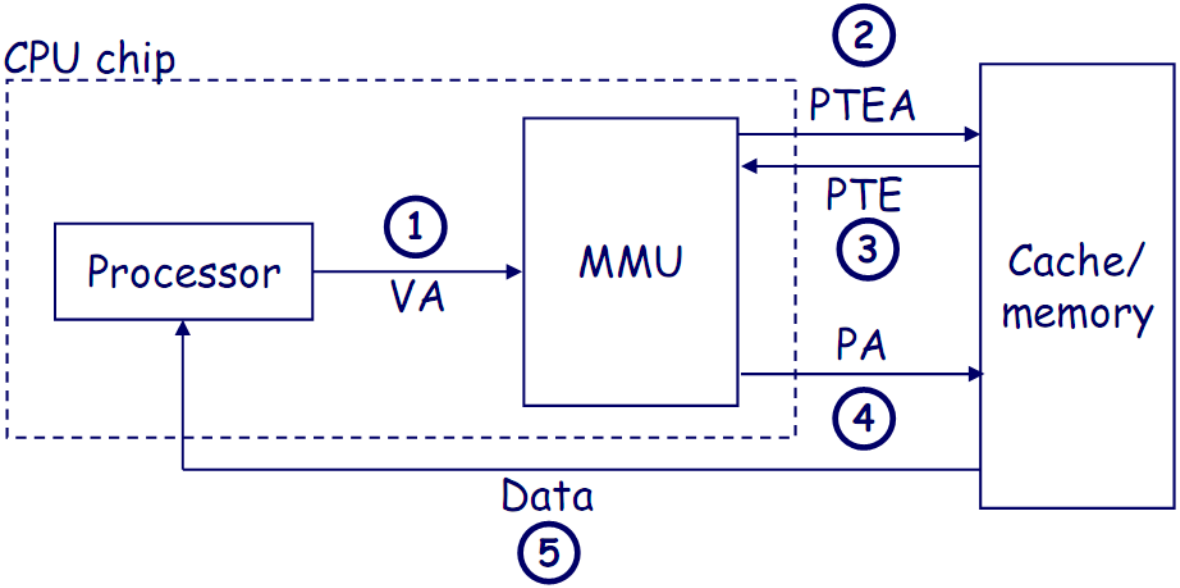
- VA → PA mapping called address translation
  - Split VA into virtual page number (VPN) & page offset
  - Translate VPN into physical page number (PPN) by page table

Address Translation Example

- Example: Memory access at address 0xFFA8AFDC
  - Page size: 4KB



Address Translation: Page Hit



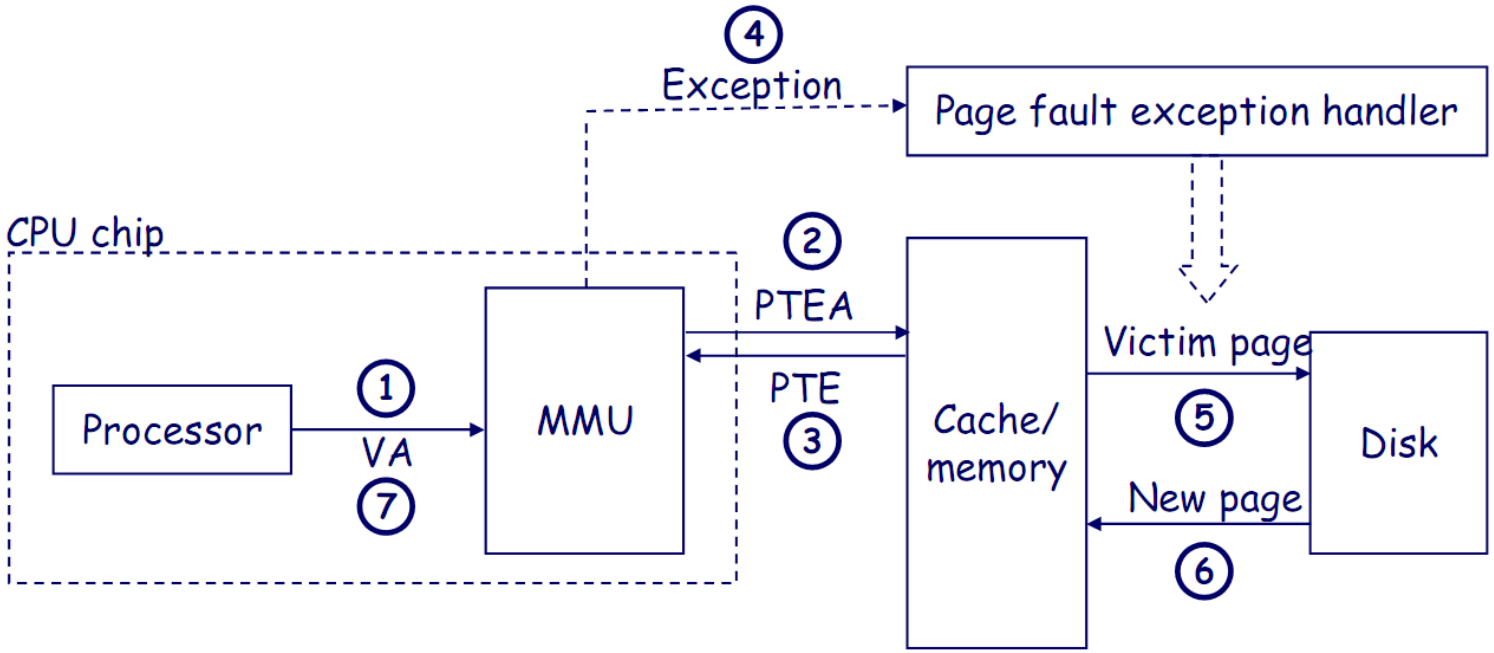
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to L1 cache

5) L1 cache 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.

Page Size Trade Offs

- What's the benefit of large pages?
  - Can reach more virtual address space
  - Reduce the # of TLB misses because the coverage becomes large
  - Keep more TLB entries
  - Reduce the HW cost for tagging
- Disadvantage of large pages
  - Waste of space within a page (internal fragmentation)
  - Waste of space within the entire physical memory (external fragmentation)
  - Transfer size from disk to memory

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages
- Idea: Store access control information on a page basis in the process's page table
- Enforce access control at the same time as translation

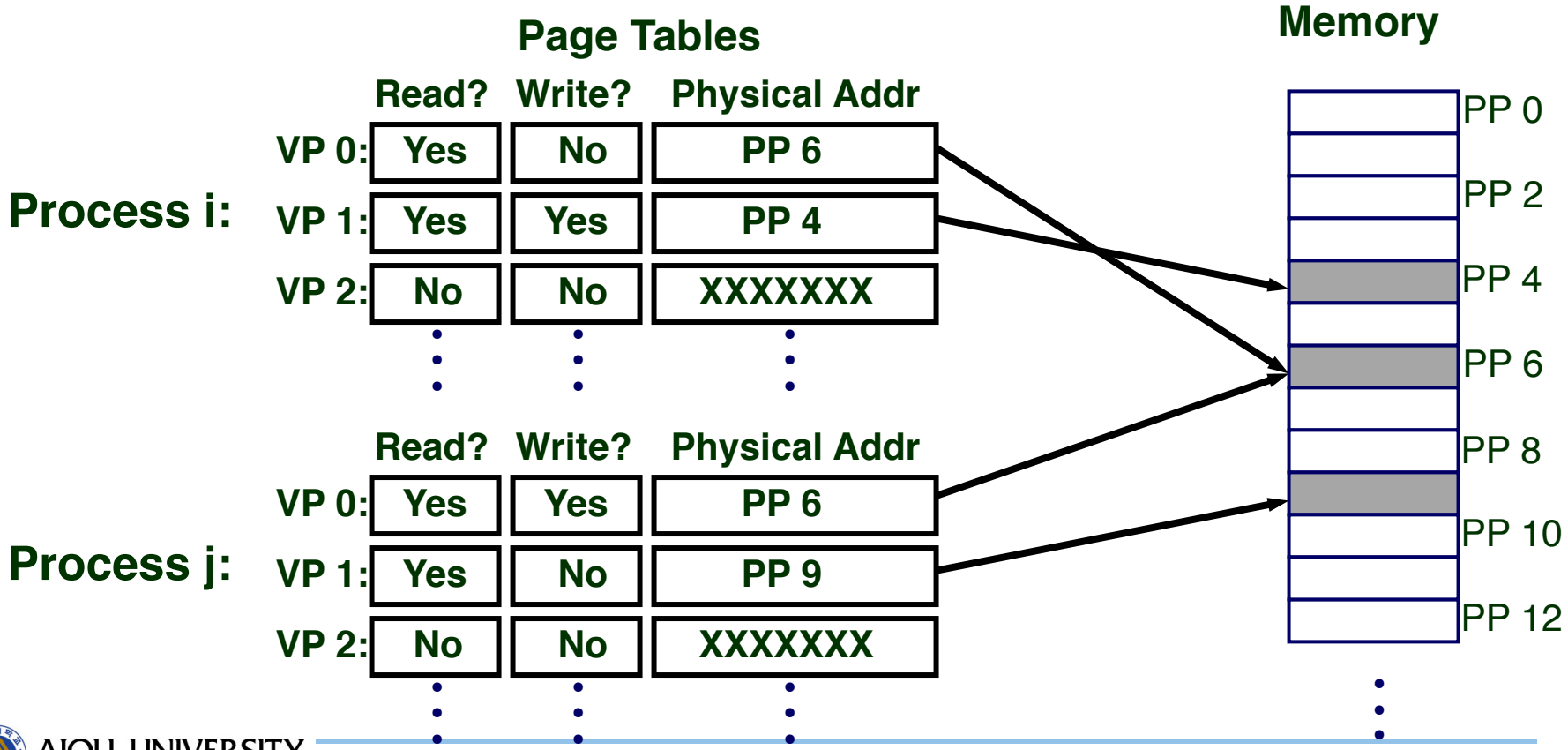
→ Virtual memory system serves two functions today

Address translation (for illusion of large physical memory)

Access control (protection)

VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
  - If violated, generate exception



Three Major Issues

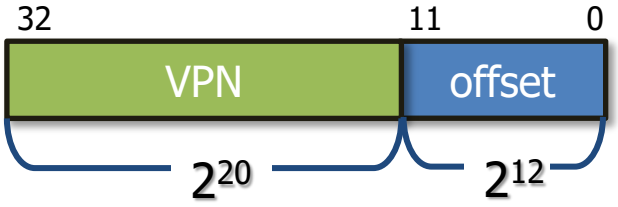
- How large is the page table and how do we store and access it?
- How can we speed up translation & access control check?
- When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - …

Virtual Memory Issue I

- How large is the page table?
- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)
- How can we store it efficiently without requiring physical memory that can store all page tables?
  - Idea: multi-level page tables
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)

Issue: Page Table Size

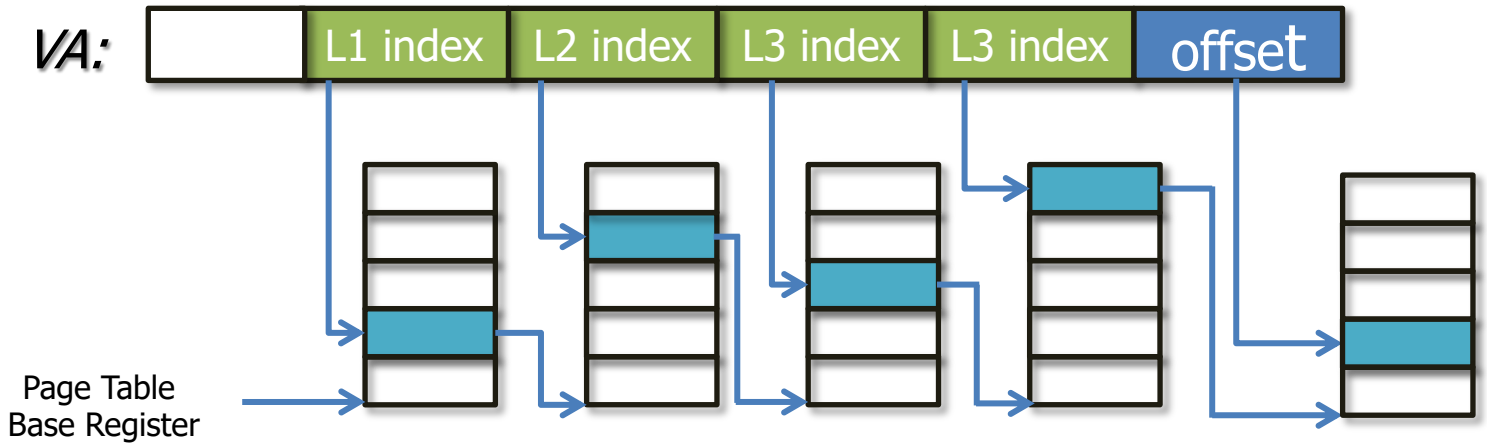
- Memory required to hold page tables can be large
  - 32-bit address space with 4KB page = $2^{32}/2^{12} = 2^{20}$ Pages



  - Flat page table: 2^{20} x 4 byte entry = 2^{22} = ?? bytes
  - Page tables are required for each process

- Observation
  - Process may not be using the entire VM space
    - E.g., daemon background service

Solution: Multi-level Page Table

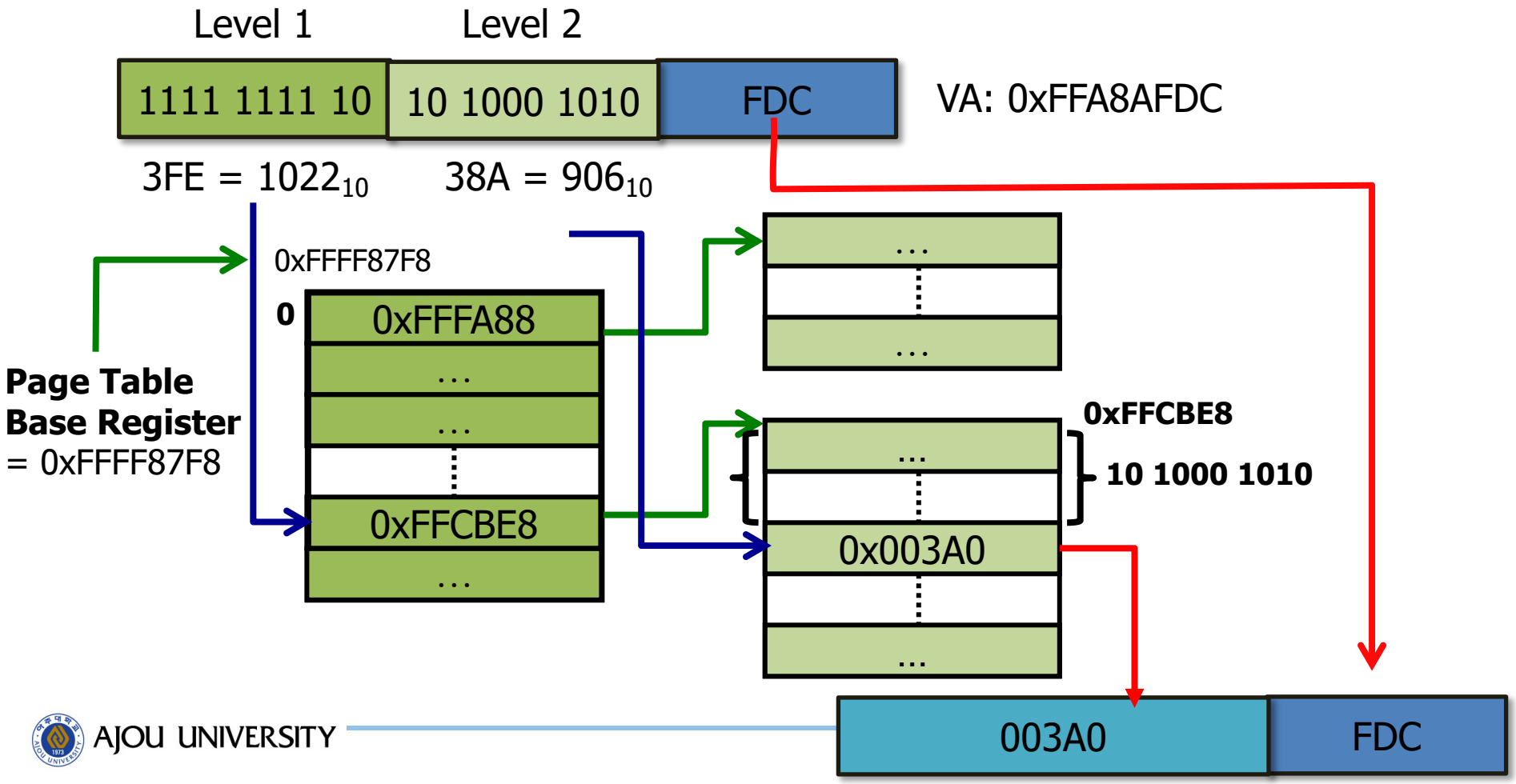


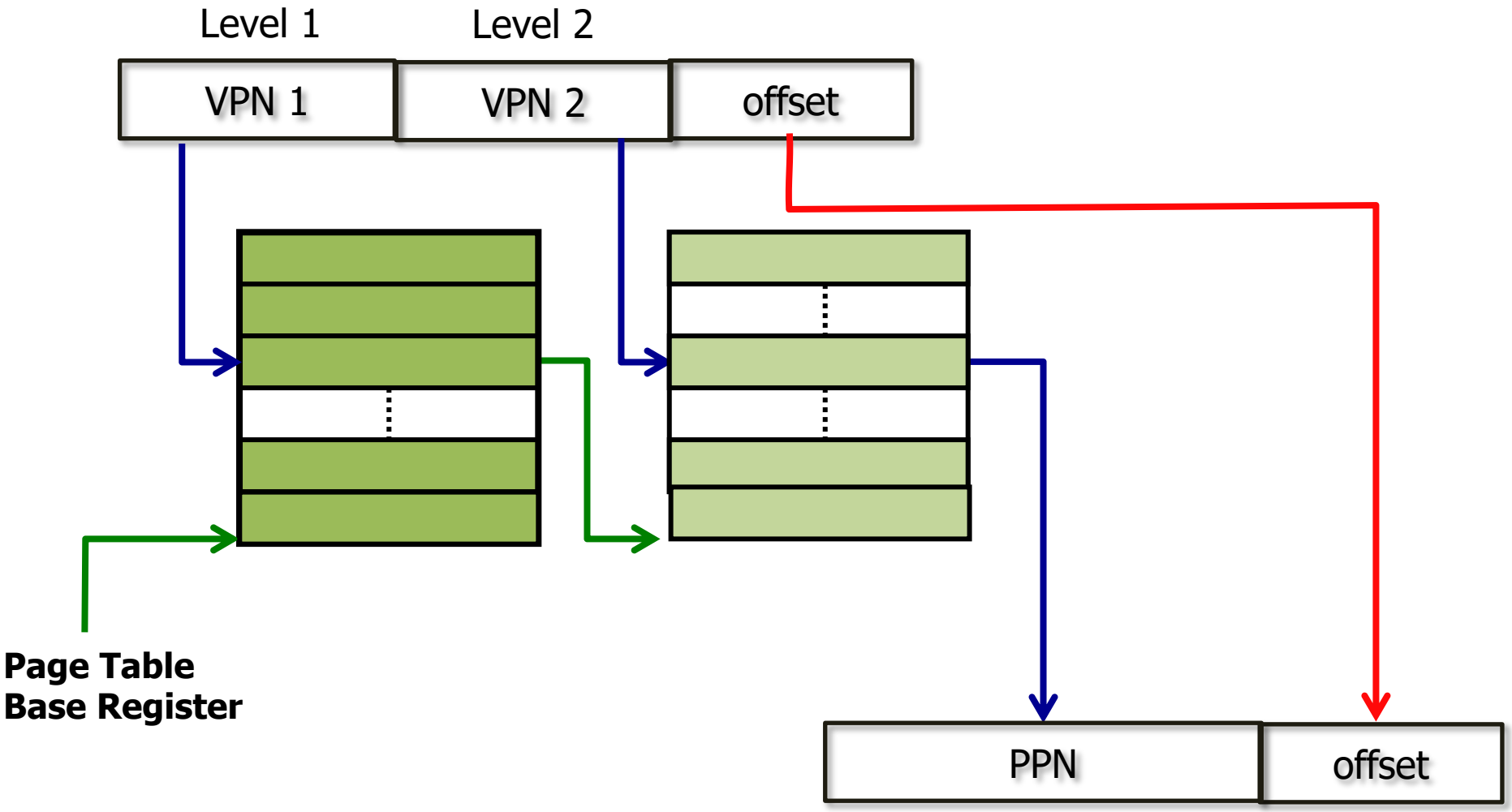
- Lowest-level tables hold PTEs
- Upper-level tables hold pointers to lower-level tables
- Different parts of VPN used to index different levels

→ But, it takes longer to traverse than flat page table

Multi-level Address Translation Example

- Example: Memory access at address 0xFFA8AFDC
  - Page size: 4KB

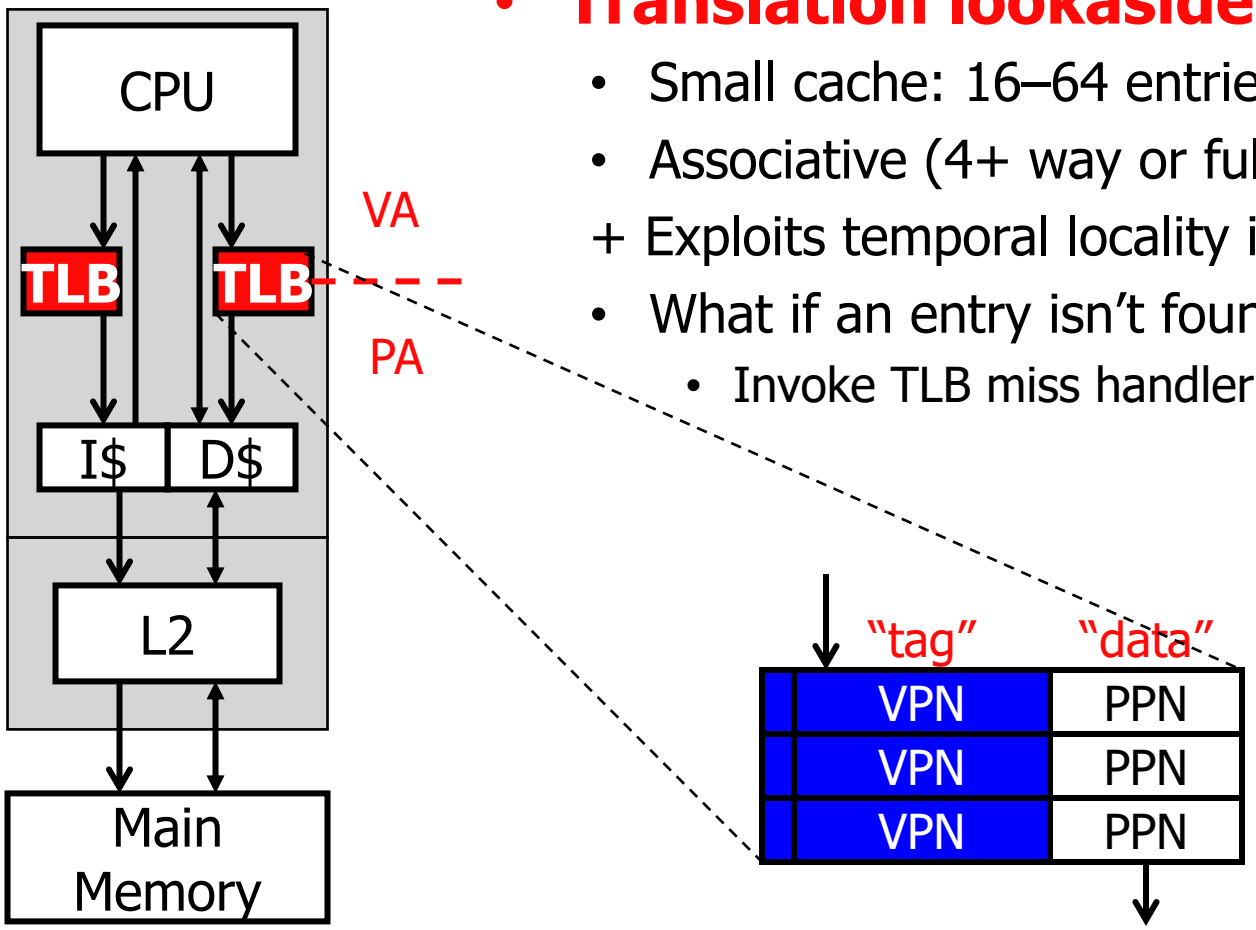



Level 1
Level 2
VPN 1
VPN 2
offset
Page Table
Base Register
PPN
offset

Virtual Memory Issue II

- How fast is the address translation?
  - How can we make it fast?
- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer
- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?
- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

Translation Lookaside Buffer



- **Translation lookaside buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  + Exploits temporal locality in page table
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler

Translation Lookaside Buffer

- TLB: cache for page table entries
- Who should fill TLBs?
  - SW-managed TLB: Alpha
    - Invoke the handler, which will traverse the page table and fill TLB entry
    - Invoking the handler costs performance (draining pipelines)
  - HW-managed TLB: x86
    - HW controller find PTE and fill TLB (transparent to OS)
    - HW controller needs fixed PTE format
- Multi-level TLBs
  - Small fully associative L1 TLB (e.g., 32 entries)
  - Large n-way associative L2 TLB (e.g., 4-way 512 entries)
- Support multiple page sizes

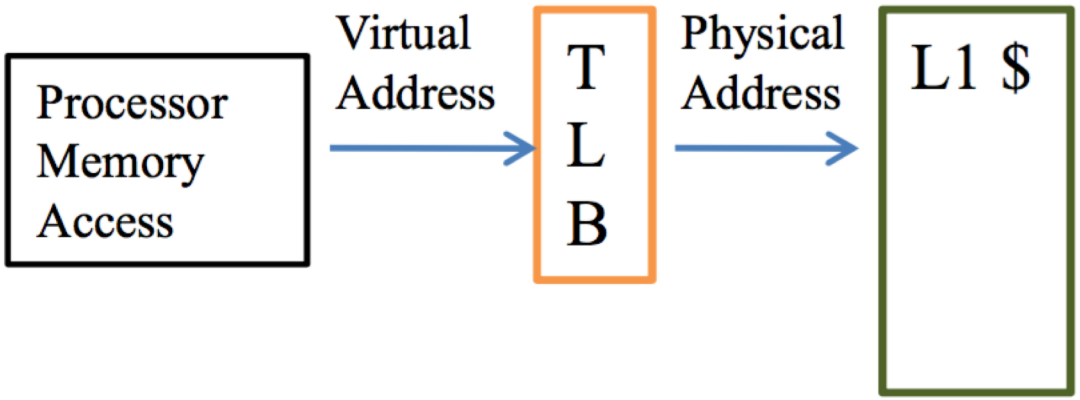
Virtual Memory Issue III

- When do we do the address translation?
  - Before or after accessing the L1 cache?

TLB and Cache Accesses

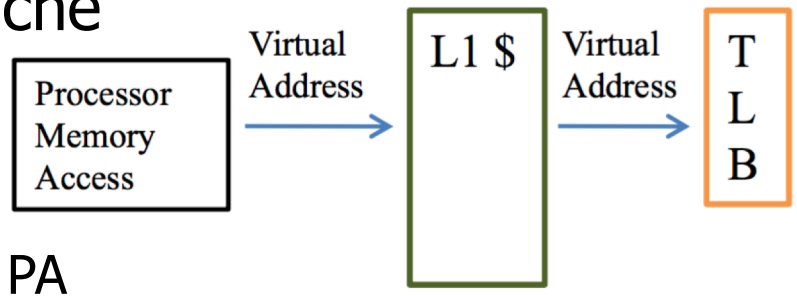
- When to translate virtual address (VAs) to physical address (PAs)

- Address translation **before** L1 cache
  - L1 sees only physical addresses (PAs)
  - *Physically-indexed, physically-tagged (PIPT)*
  - Slow L1 access



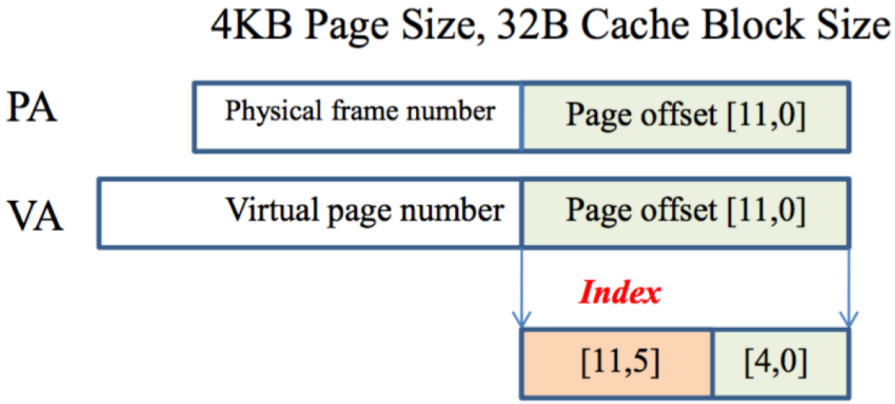
Virtually-Indexed, Virtually-Tagged Cache

- Address translation **after** L1 cache
- Fast L1 access, but…
- *Aliasing* problem
  - Multiple VAs mapped to the same PA
  - Multiple blocks for the same physical address in the cache
  - Consistency for duplicate blocks is complex
  - A hardware solution → check all possible locations on a miss, and remove synonyms
  - How to know if two VAs are synonym → must know the Pas
- Need to flush the cache for context switch
- Get more complicated in multiprocessors because cache coherence is done in physical address



Parallel TLB and Cache Access

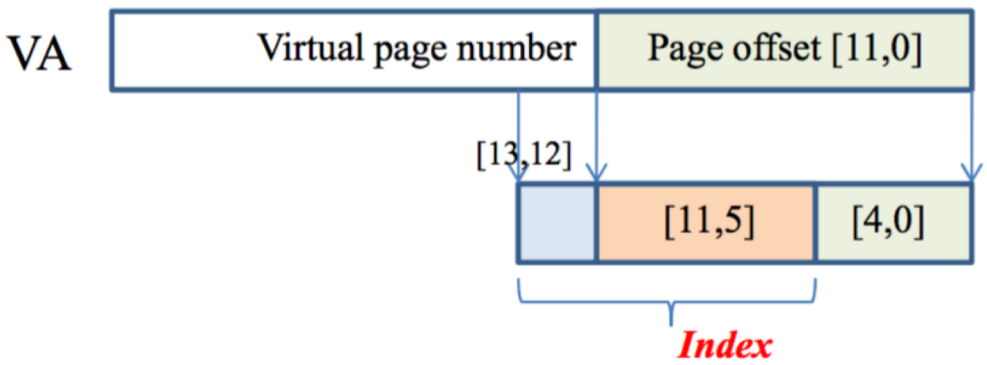
- Page offset is equal in PA and VA → Use only the common bits as index



- 7 bits for index → 128 sets
- Maximum cache size
  Direct-mapped: 128×32B = 4KB
  2-way: 128×2×32B = 8KB
  4-way: 128×4×32B = 16KB
  8-way: 128×8×32B = 32KB

- Number of sets limited by page size
  - To increase cache size, must increase associativity → can make cache slower

Virtually-Indexed, Physically-Tagged Cache



- 9 bits for index → 512 sets
- Maximum cache size
  Direct-mapped: 512×32B = 16KB
  2-way: 512×2×32B = 32KB
  4-way: 512×4×32B = 64KB
  8-way: 512×8×32B = 128KB

- *Aliasing* problem
  - Multiple sets for a PA (alias problem)
  - In the example, 2 extra bits are used → 4 possible locations for a PA
  - For a cache miss, search all possible locations, and invalidate duplicate blocks
  - Finding synonyms is easier than virtually-index, virtually-tagged cache
- No flushing needed for context switch
- Coherence probes search all possible locations
- Common in high performance microprocessors

Summary

- Virtual memory (paging)
- Address translation
- Virtually-indexed, physically-tagged caches enable parallel TLB and cache accesses with moderate overheads