

Design of Digital Circuits

Lecture 23b: Virtual Memory

Prof. Onur Mutlu

ETH Zurich

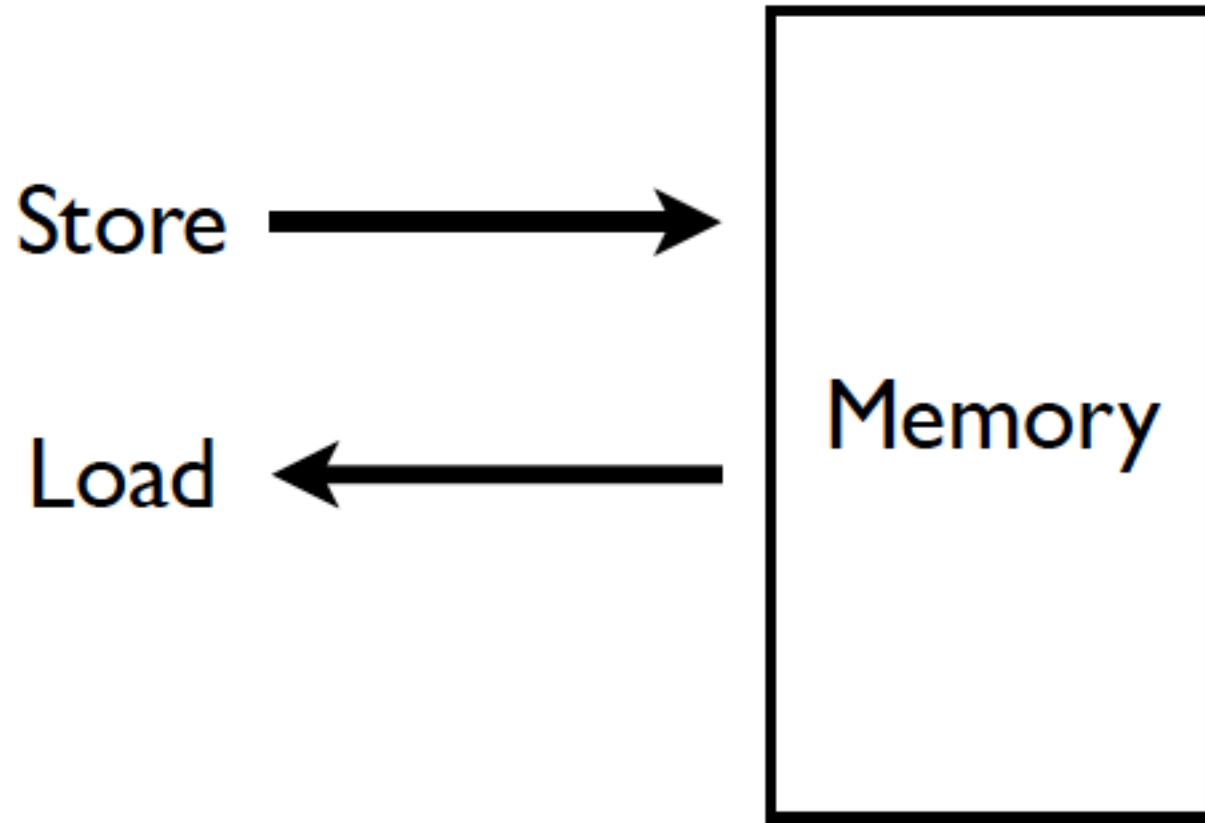
Spring 2019

17 May 2019

Readings

- Virtual Memory
- Required
 - H&H Chapter 8.4

Memory (Programmer's View)



Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

Abstraction: Virtual vs. Physical Memory

- **Programmer** sees **virtual memory**
 - Can assume the memory is “infinite”
 - Reality: **Physical memory** size is much smaller than what the programmer assumes
 - **The system** (system software + hardware, cooperatively) maps **virtual memory addresses** to **physical memory**
 - The system automatically manages the physical memory space **transparently to the programmer**
- + Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer
- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

Benefits of Automatic Management of Memory

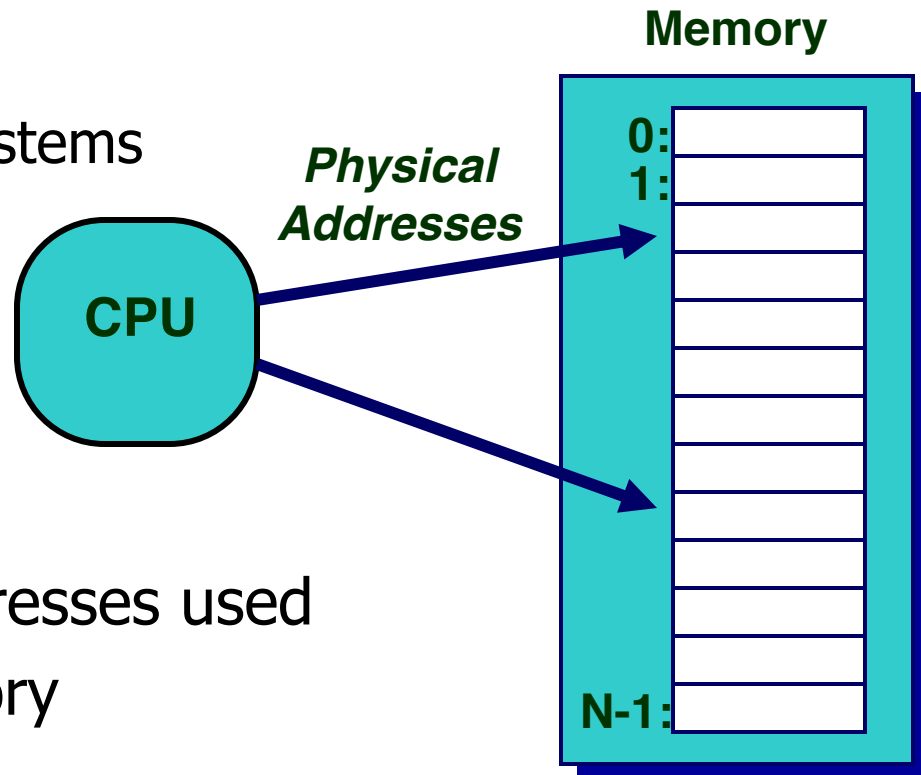
- Programmer does not deal with physical addresses
- Each process has its own mapping from virtual→physical addresses

- Enables
 - ❑ Code and data to be located anywhere in physical memory
(relocation)
 - ❑ Isolation/separation of code and data of different processes in physical memory
(protection and isolation)
 - ❑ Code and data sharing between multiple processes
(sharing)

A System with Physical Memory Only

■ Examples:

- ❑ most Cray machines
- ❑ early PCs
- ❑ nearly all embedded systems



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

The Problem

- Physical memory is of limited size (cost)
 - What if you need more?
 - Should the programmer be concerned about the size of code/data blocks fitting physical memory?
 - Should the programmer manage data movement from disk to physical memory?
 - Should the programmer ensure two processes (different programs) do not use the same physical memory?
- Also, ISA can have an address space greater than the physical memory size
 - E.g., a 64-bit address space with byte addressability
 - What if you do not have enough 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
 - Addresses are directly specified in the program
- Difficult to support multiple processes
 - Protection and isolation between multiple processes
 - Sharing of physical memory space
- Difficult to support data/code sharing across processes

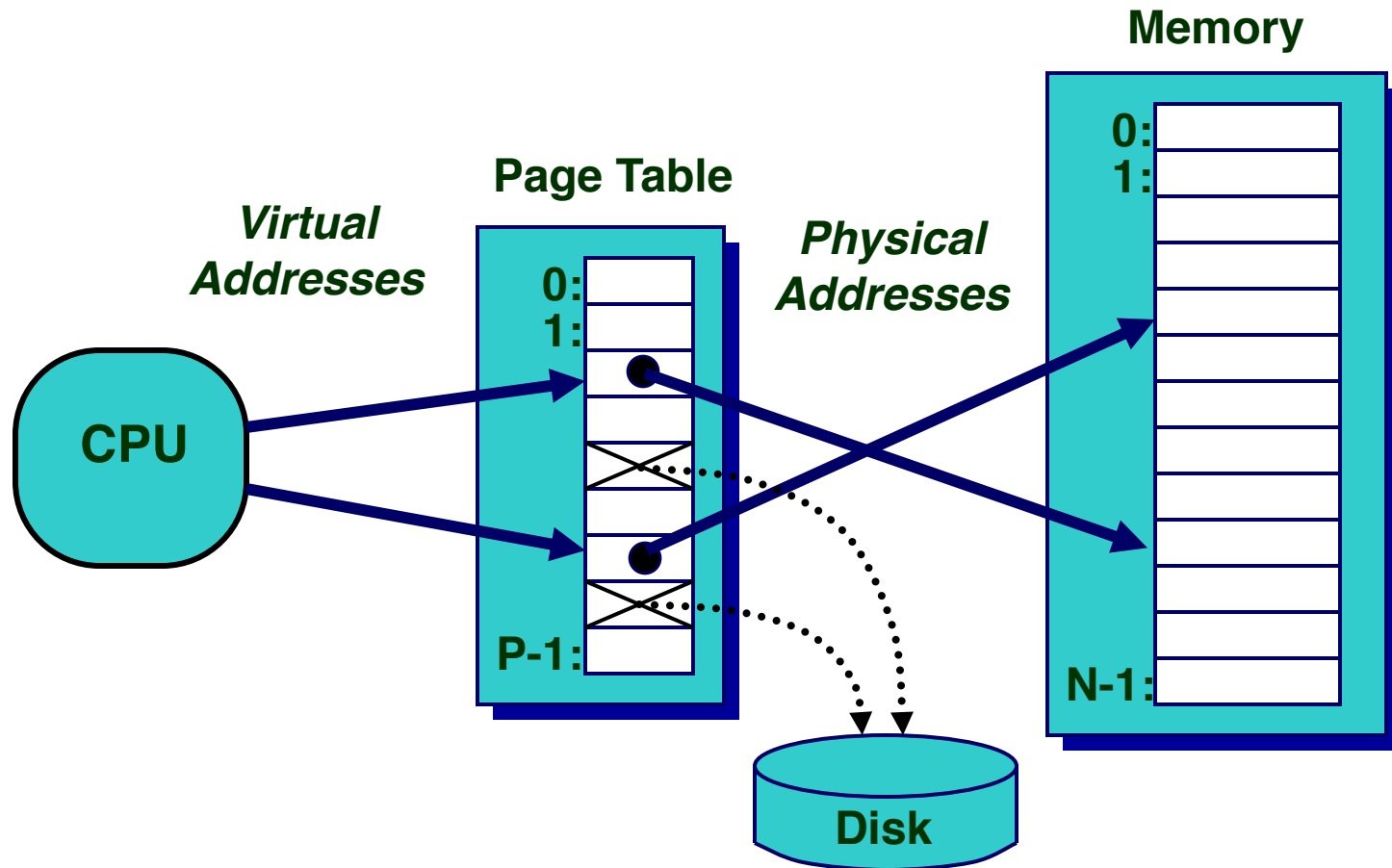
Virtual Memory

- 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

Basic Mechanism

- Indirection (in addressing)
- Address generated by each instruction in a program is a “virtual address”
 - i.e., it is not the physical address used to address main memory
 - called “linear address” in x86
- An “address translation” mechanism maps this address to a “physical address”
 - called “real address” in x86
 - Address translation mechanism can be implemented in hardware and software together

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)

Virtual Pages, Physical Frames

- Virtual address space divided into pages
- Physical address space divided into frames
- A virtual page is mapped to
 - A physical frame, if the page is in 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 frame and adjusts the mapping → this is called demand paging
- Page table is the table that stores the mapping of virtual pages to physical frames

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 potentially 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 cache?
 - Replacement: what page to remove to make room in cache?
 - Granularity of management: large, small, uniform pages?
 - Write policy: what do we do about writes? Write back?

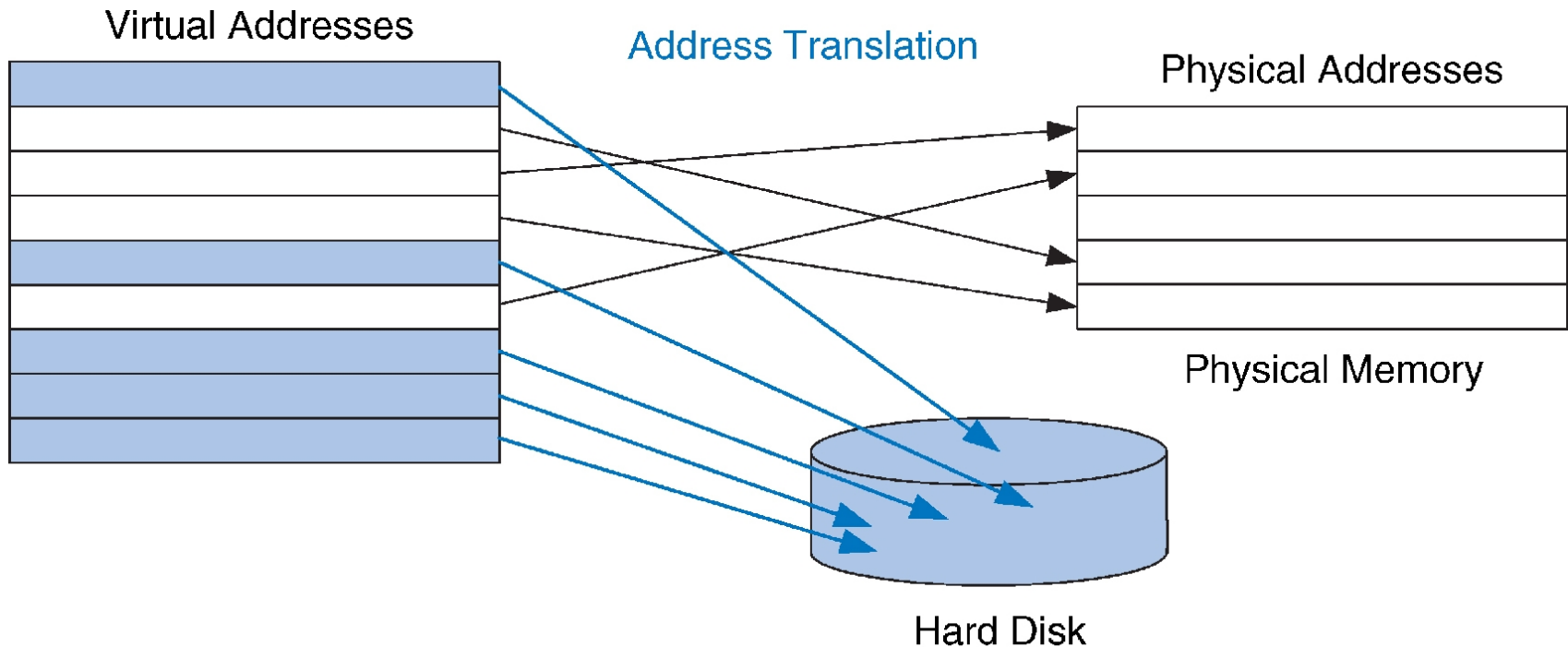
Cache/Virtual Memory Analogues

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Virtual Memory Definitions

- **Page size**: amount of memory transferred from hard disk to DRAM at once
- **Address translation**: determining the physical address from the virtual address
- **Page table**: lookup table used to translate virtual addresses to physical addresses (and find where the associated data is)

Virtual and Physical Addresses

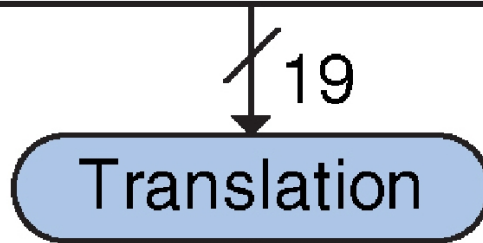
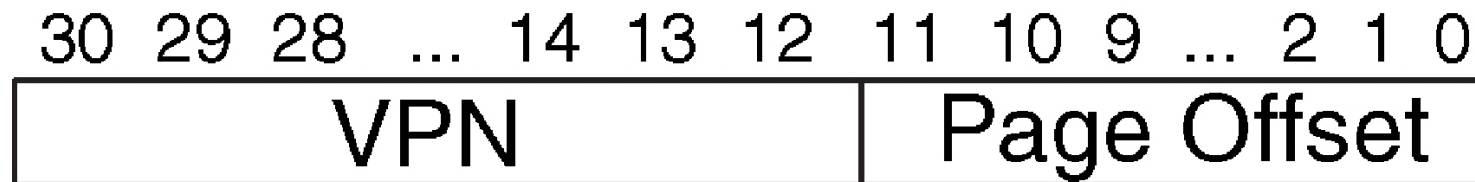


© 2007 Elsevier, Inc. All rights reserved

- Most accesses hit in physical memory
- But programs see the large capacity of virtual memory

Address Translation

Virtual Address



26 25 24 ... 13 12 11 10 9 ... 2 1 0

Physical Address

Virtual Memory Example

- System:

- Virtual memory size: 2 GB = 2^{31} bytes
- Physical memory size: 128 MB = 2^{27} bytes
- Page size: 4 KB = 2^{12} bytes

Virtual Memory Example

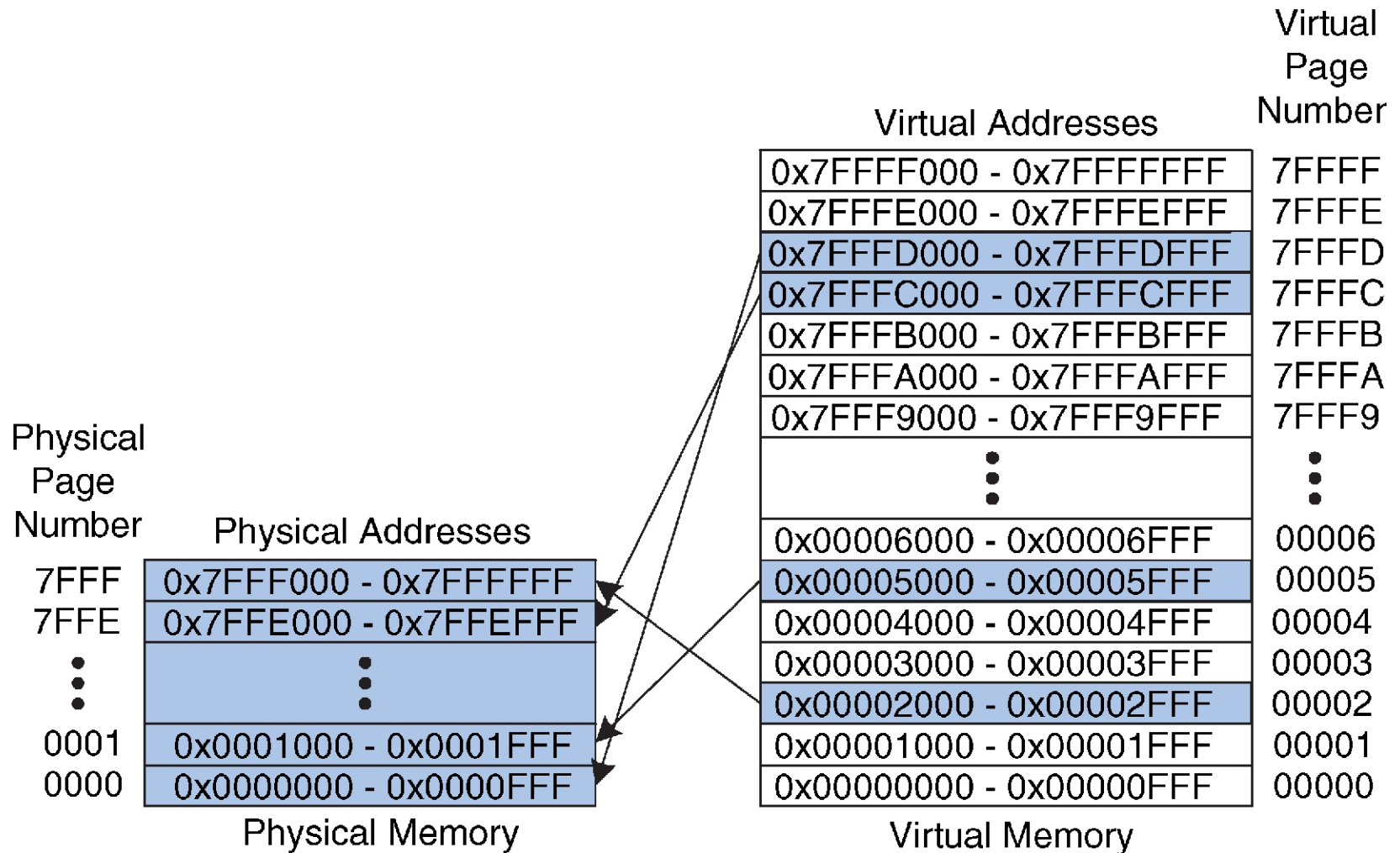
■ System:

- ❑ Virtual memory size: 2 GB = 2^{31} bytes
- ❑ Physical memory size: 128 MB = 2^{27} bytes
- ❑ Page size: 4 KB = 2^{12} bytes

■ Organization:

- ❑ Virtual address: **31** bits
- ❑ Physical address: **27** bits
- ❑ Page offset: **12** bits
- ❑ # Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
- ❑ # Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)

Virtual Memory Example



How Do We Translate Addresses?

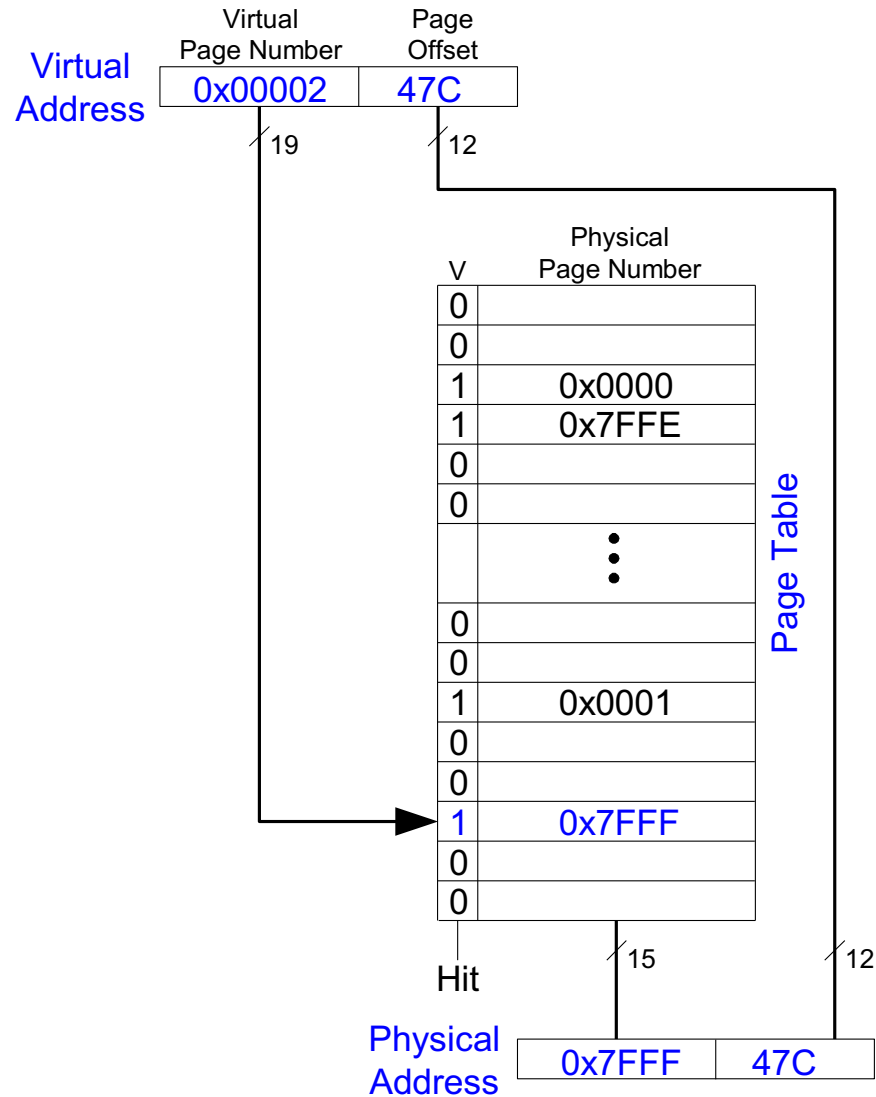
- **Page table**

- Has entry for each virtual page

- Each **page table entry** has:

- **Valid bit**: whether the virtual page is located in physical memory (if not, it must be fetched from the hard disk)
- **Physical page number**: where the page is located
- (Replacement policy, dirty bits)

Page Table Example



Page Table Example 1

- What is the physical address of virtual address 0x5F20?

V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table

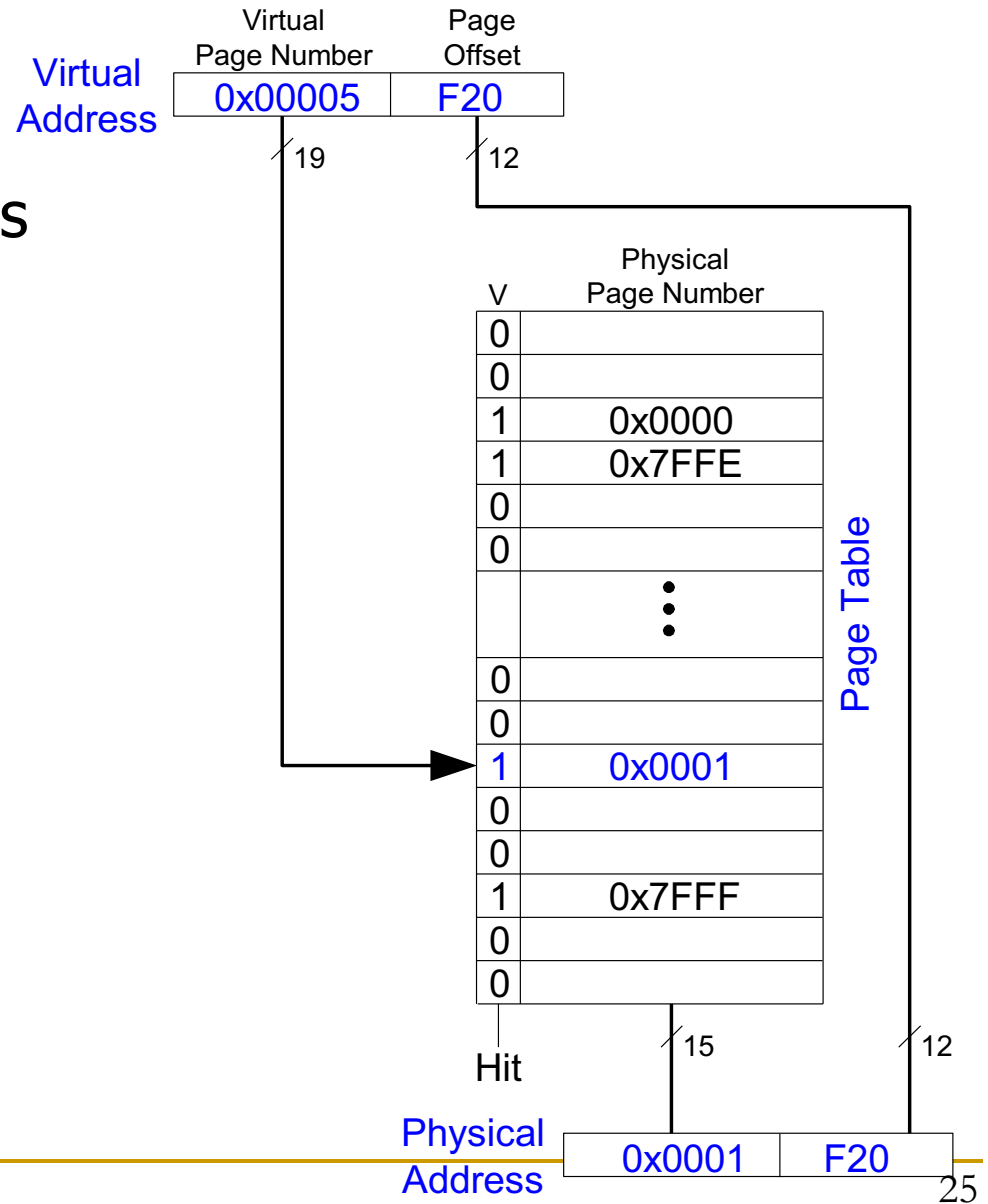
Hit

15

Page Table Example 1

- What is the physical address of virtual address 0x5F20?

- VPN = 5
- Entry 5 in page table indicates VPN 5 is in physical page 1
- Physical address is 0x1F20



Page Table Example 2

- What is the physical address of virtual address 0x73E0?

V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

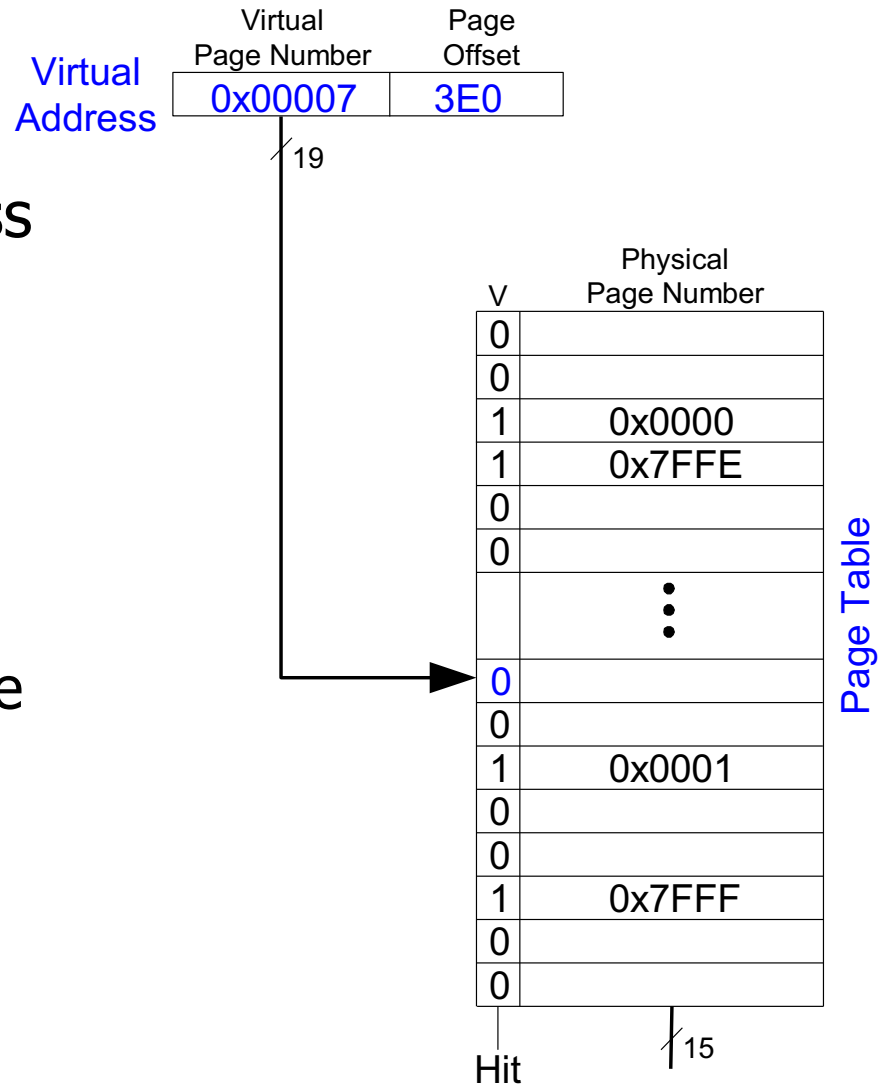
Page Table

Hit

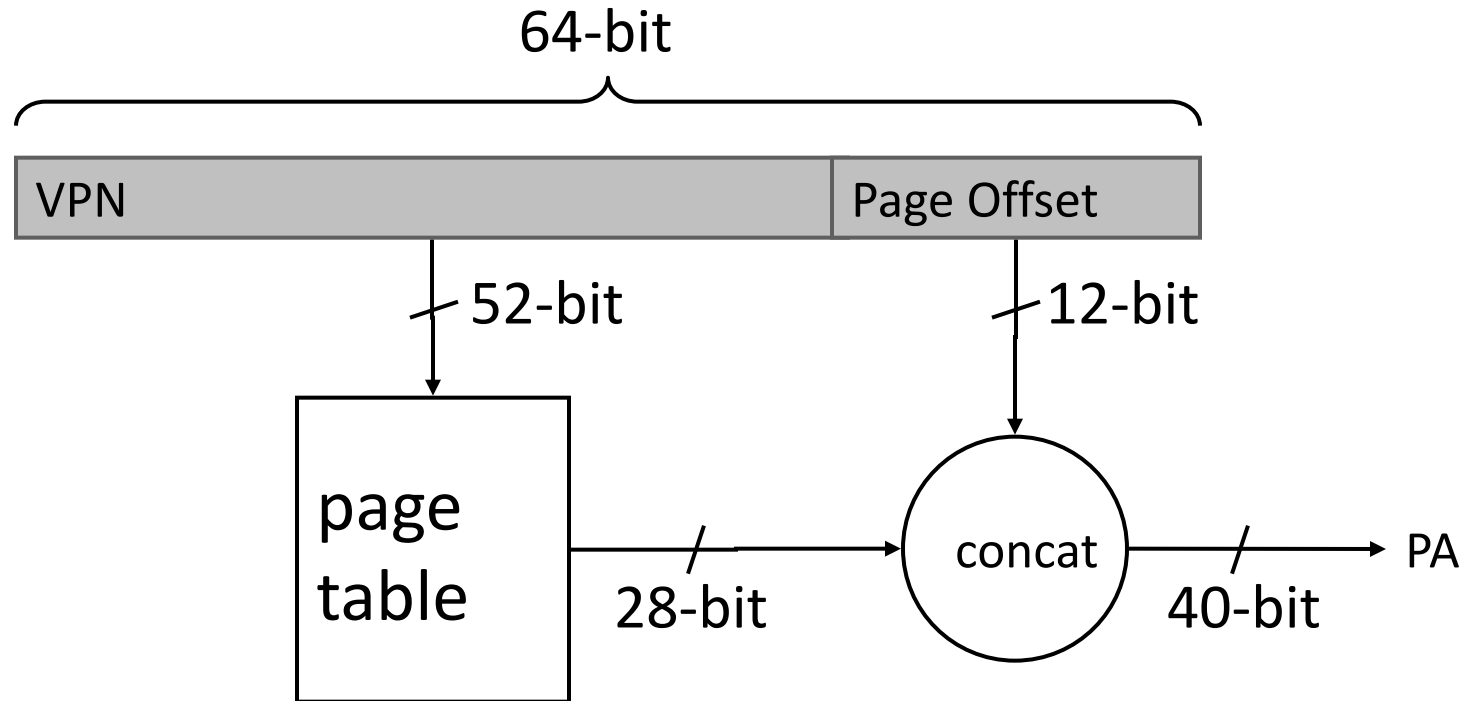
15

Page Table Example 2

- What is the physical address of virtual address 0x73E0?
 - VPN = 7
 - Entry 7 in page table is invalid, so the page is not in physical memory
 - The virtual page must be swapped into physical memory from disk



Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
 - **2^{52} entries x ~4 bytes $\approx 2^{54}$ bytes**
and that is for just one process!
and the process may not be using the entire VM space!

Page Table Challenges

- Page table is large
 - at least part of it needs to be located in physical memory
- Each load/store requires at least two memory accesses:
 1. one for translation (page table read)
 2. one to access data with the physical address (after translation)
- Two memory accesses to service a load/store greatly degrades load/store execution time
 - Unless we are clever...

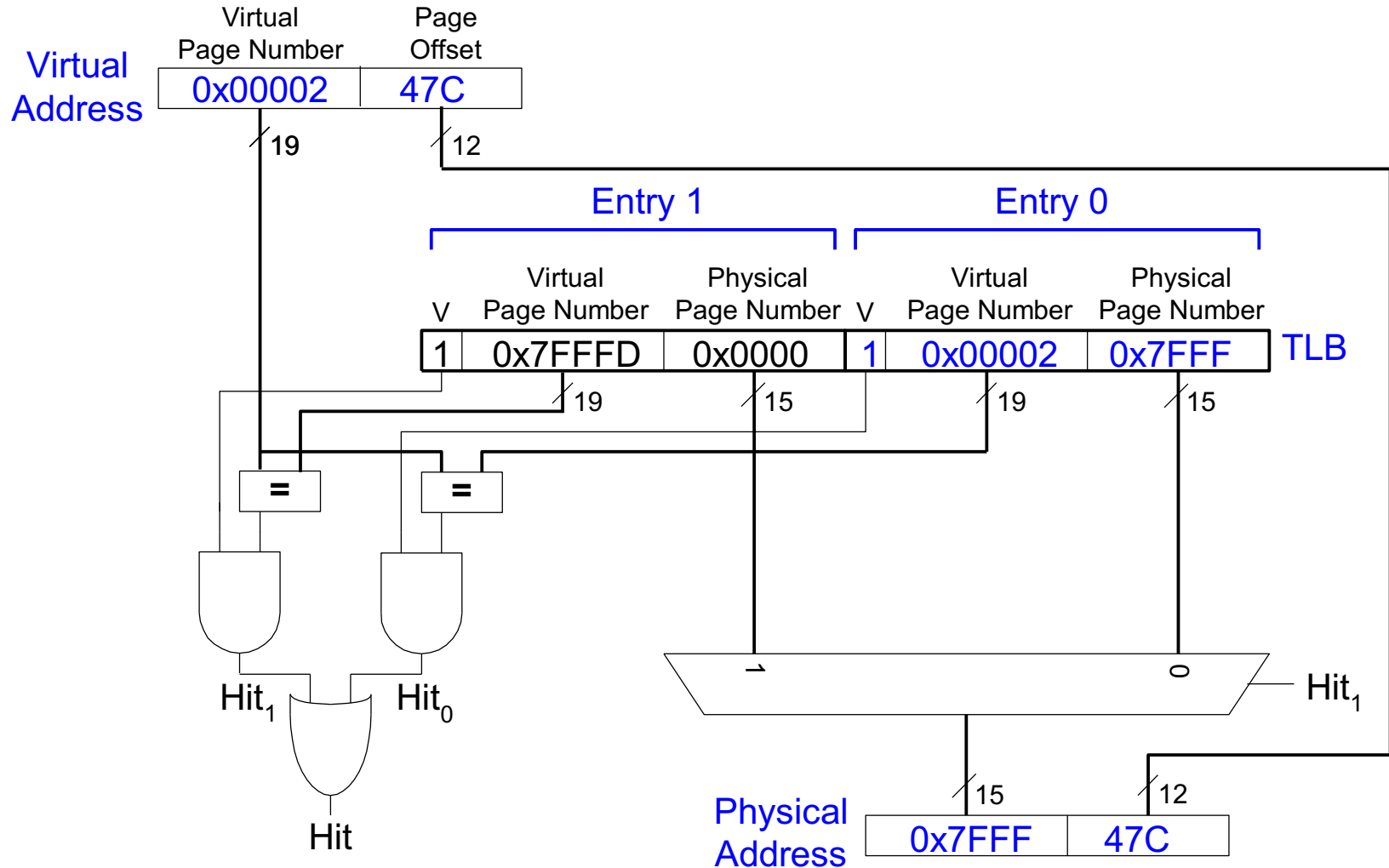
Translation Lookaside Buffer (TLB)

- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor
- Translation lookaside buffer (TLB)
 - Small cache of most recently used translations (PTEs)
 - Reduces number of memory accesses required for *most* loads/stores to only one

Translation Lookaside Buffer (TLB)

- Page table accesses have a lot of temporal locality
 - Data accesses have temporal and spatial locality
 - Large page size (say 4KB, 8KB, or even 1-2GB), so consecutive loads/stores likely to access same page
- TLB
 - Small: accessed in < 1 cycle
 - Typically 16 - 512 entries
 - High associativity
 - $> 99\%$ hit rates typical (depends on workload)
 - Reduces # of memory accesses for most loads and stores to only 1

Example Two-Entry TLB



Design of Digital Circuits

Lecture 23b: Virtual Memory

Prof. Onur Mutlu

ETH Zurich

Spring 2019

17 May 2019

We did not cover the following slides in lecture.
These are for your benefit.

Memory Protection

- Multiple programs (*processes*) run at once
 - Each process has its own page table
 - Each process can use entire virtual address space without worrying about where other programs are
- A process can only access physical pages mapped in its page table – cannot overwrite memory of another process
 - Provides protection and isolation between processes
 - Enables access control mechanisms per page

Virtual Memory Summary

- Virtual memory gives the illusion of “infinite” capacity
- A subset of virtual pages are located in physical memory
- A **page table** maps virtual pages to physical pages – this is called address translation
- A **TLB** speeds up address translation
- Using different page tables for different programs provides **memory protection**

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 software** 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

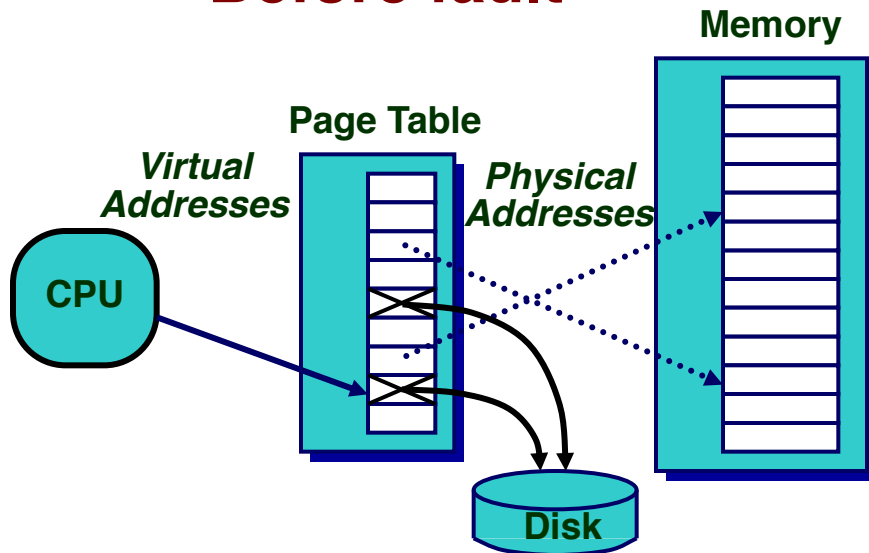
Some System Software Jobs for VM

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

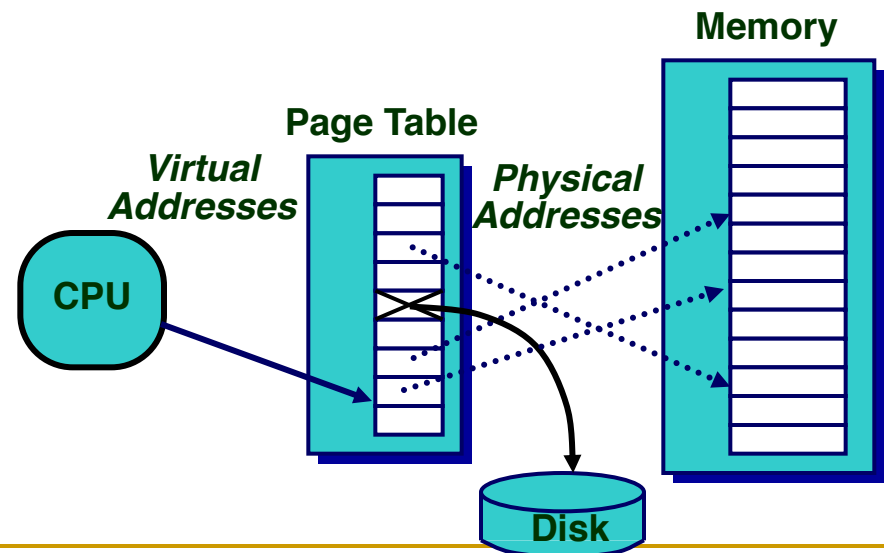
Page Fault (“A Miss in Physical Memory”)

- If a page is not in physical memory but disk
 - ❑ Page table entry indicates virtual 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

Before fault

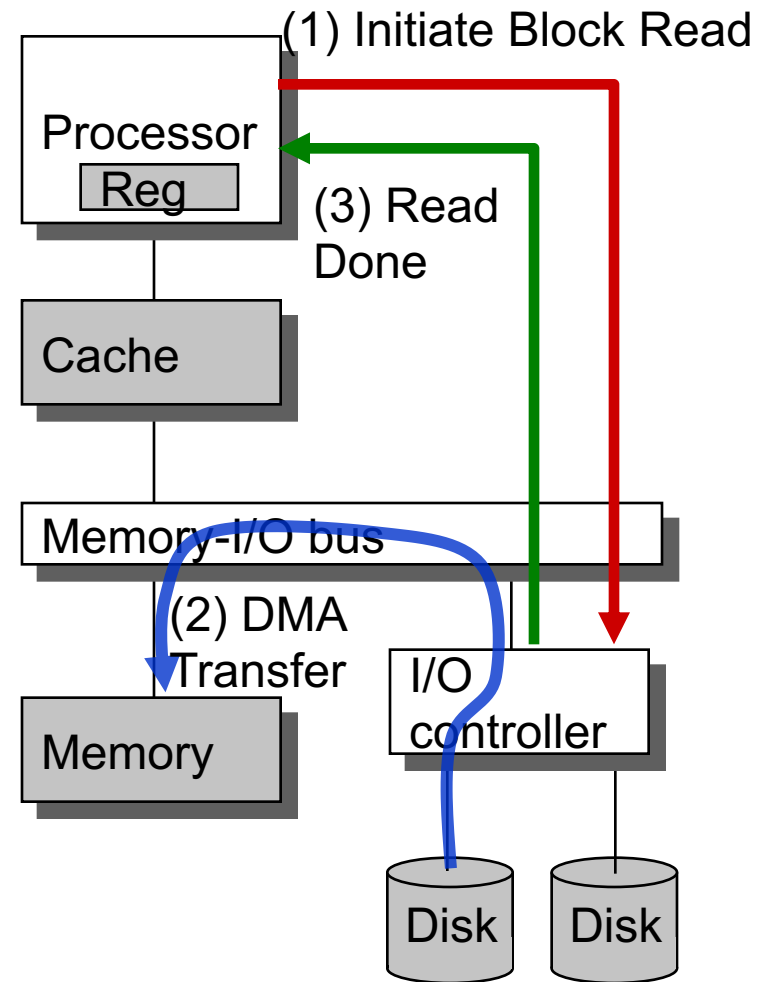


After fault



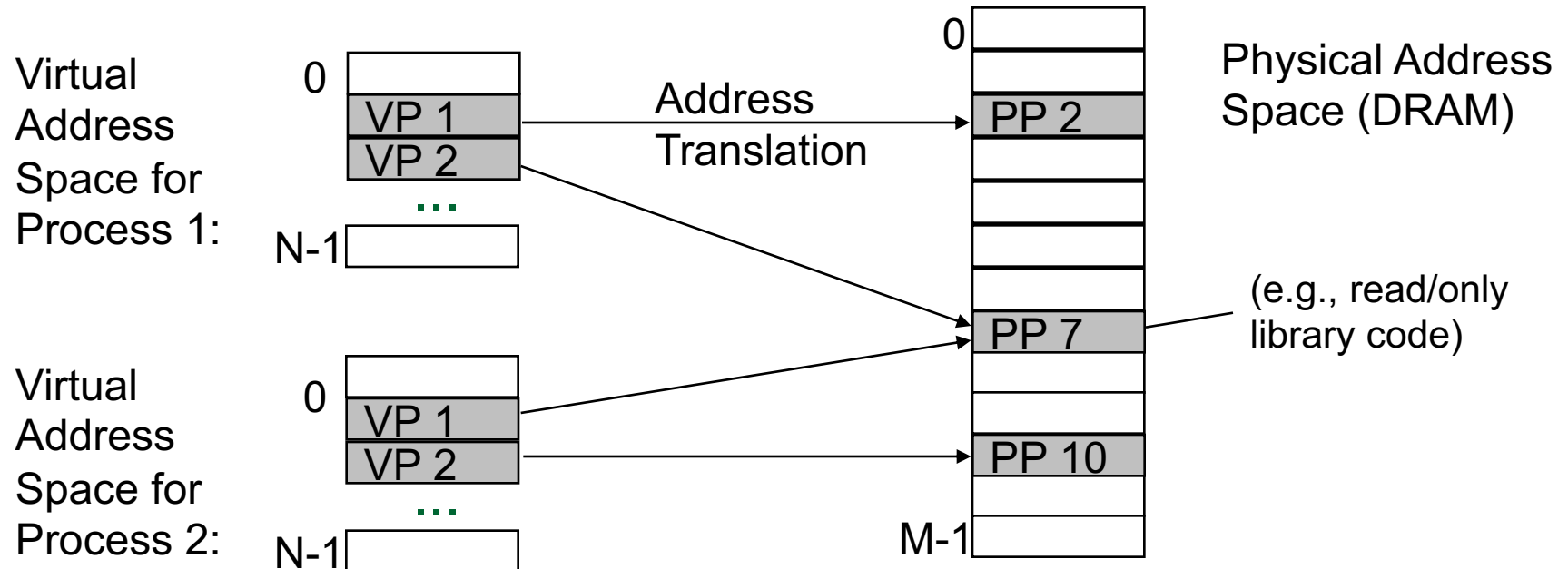
Servicing a Page Fault

- (1) Processor signals controller
 - ❑ 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

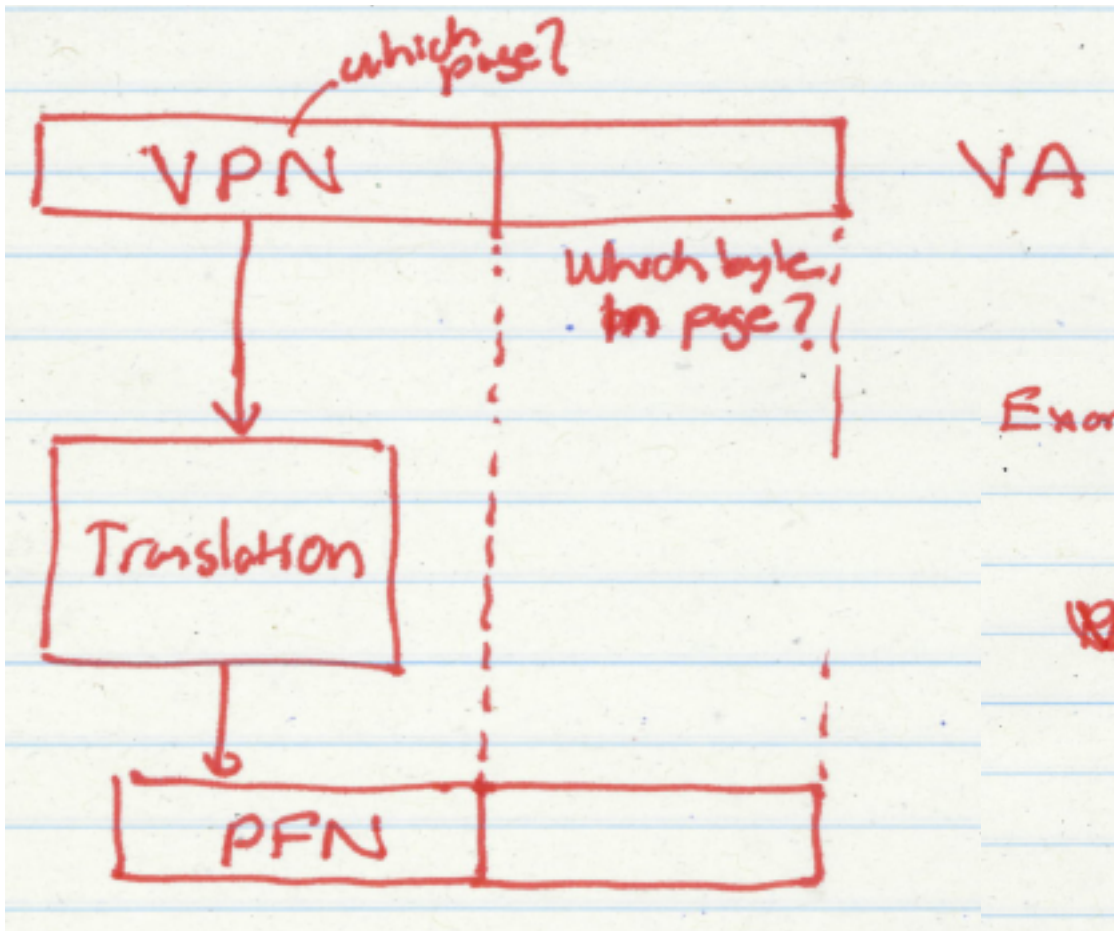
- Each process has its own virtual address space
 - Full address space for each program
 - Simplifies memory allocation, sharing, linking and loading.



Address Translation

- How to obtain the physical address from a virtual address?
- Page size specified by the ISA
 - VAX: 512 bytes
 - Today: 4KB, 8KB, 2GB, ... (small and large pages mixed together)
 - Trade-offs? (remember cache lectures)
- Page Table contains an entry for each virtual page
 - Called Page Table Entry (PTE)
 - What is in a PTE?

Address Translation (II)



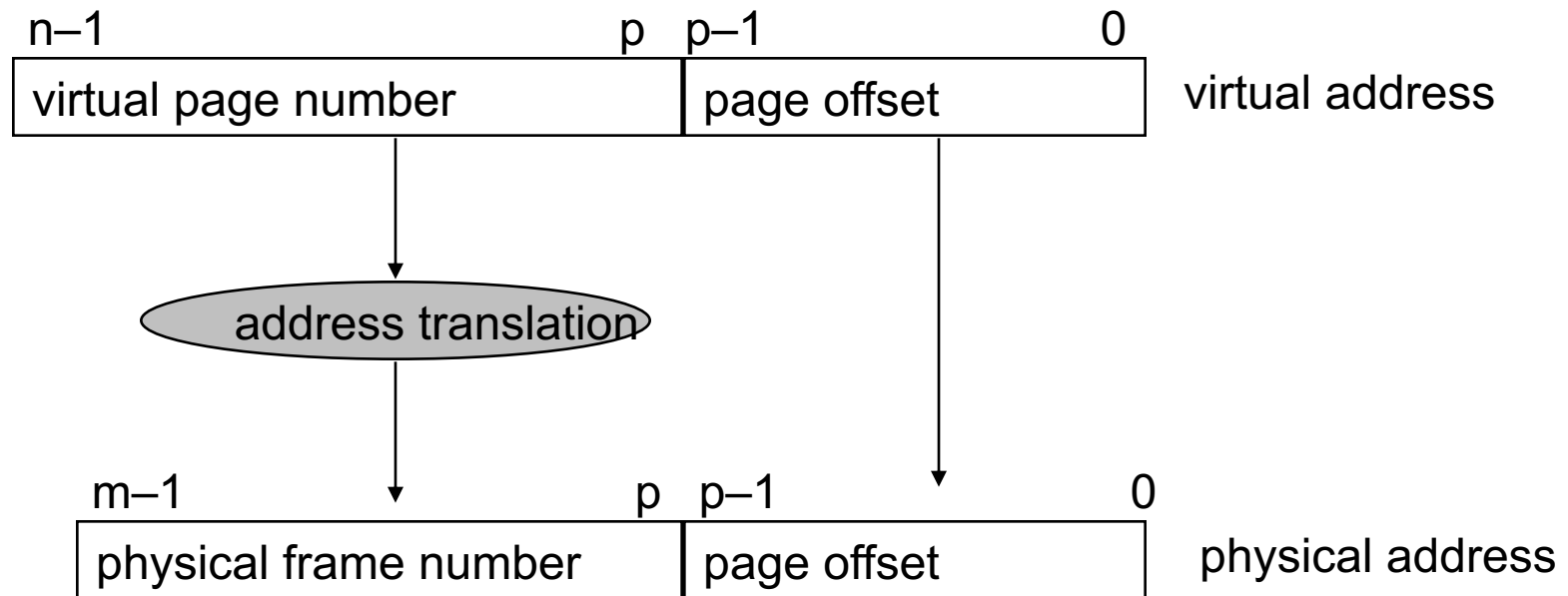
Example: 8K page size
32-bit virtual address space

VPN \rightarrow 19 bits
 $\rightarrow 2^{19}$ virtual pages
 $\rightarrow 2^{19}$ PTEs in page table
(for each process)

Address Translation (III)

■ Parameters

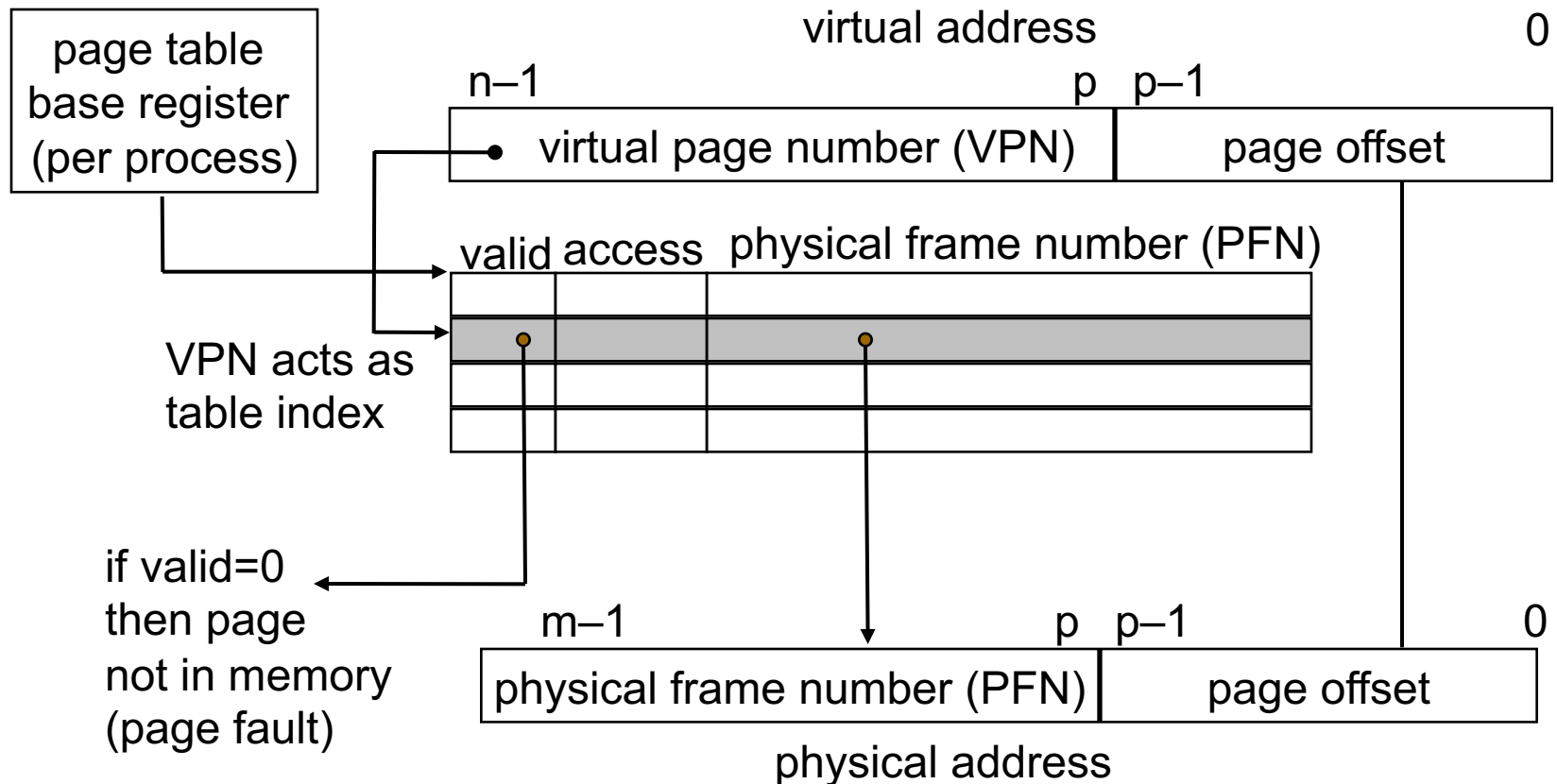
- ❑ $P = 2^p =$ page size (bytes).
- ❑ $N = 2^n =$ Virtual-address limit
- ❑ $M = 2^m =$ Physical-address limit



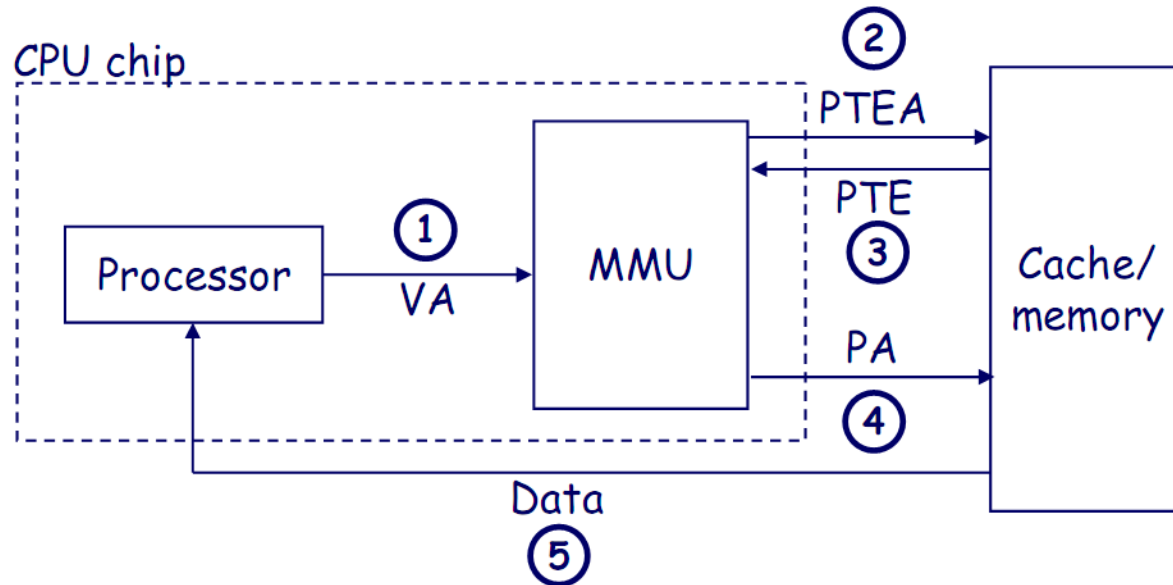
Page offset bits don't change as a result of translation

Address Translation (IV)

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page

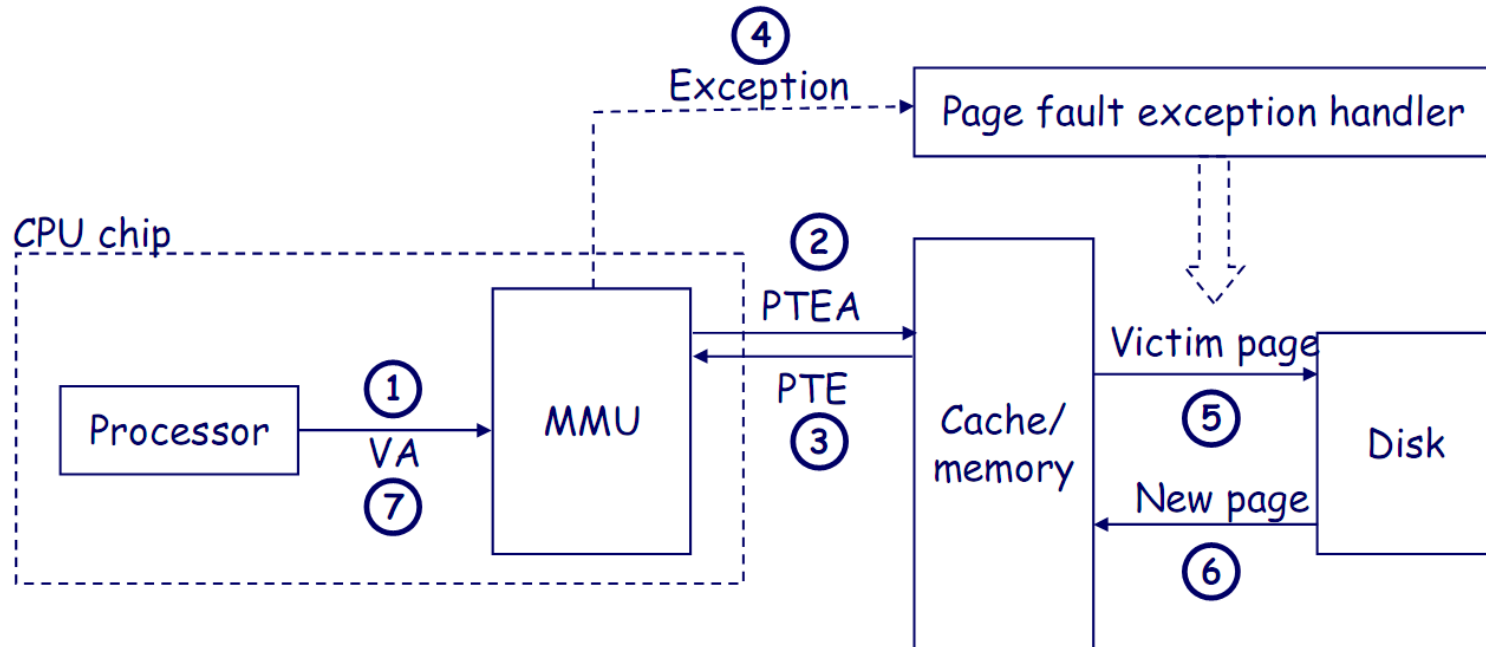


Address Translation: Page Hit



- 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

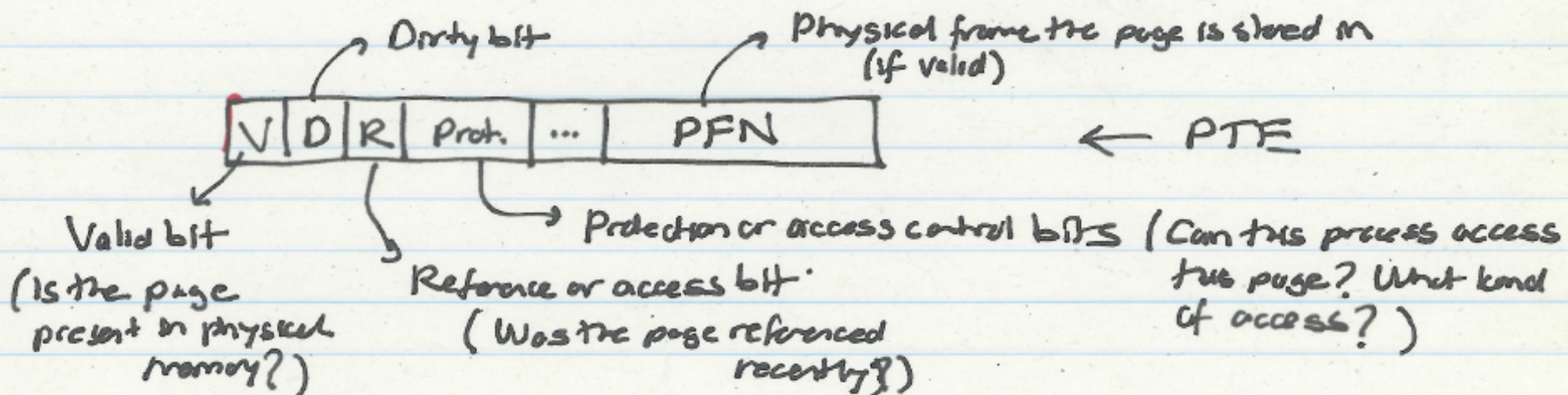
Address Translation: Page Fault



- 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.

What Is in a Page Table Entry (PTE)?

- Page table is the “tag store” for the physical memory data store
 - A mapping table between virtual memory and physical memory
- PTE is the “tag store entry” for a virtual page in memory
 - Need a **valid** bit → to indicate validity/presence in physical memory
 - Need **tag** bits (PFN) → to support translation
 - Need bits to support **replacement**
 - Need a **dirty** bit to support “write back caching”
 - Need **protection bits** to enable access control and protection



Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
 - Usually managed by system software via the virtual memory subsystem
- Page replacement is similar to cache replacement
- Page table is the “tag store” for physical memory data store
- What is the difference?
 - Required speed of access to cache vs. physical memory
 - Number of blocks in a cache vs. physical memory
 - “Tolerable” amount of time to find a replacement candidate (disk versus memory access latency)
 - Role of hardware versus software

Page Replacement Algorithms

- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?
- Is True LRU feasible?
 - 4GB memory, 4KB pages, how many possibilities of ordering?
- Modern systems use approximations of LRU
 - E.g., the CLOCK algorithm
- And, more sophisticated algorithms to take into account “frequency” of use
 - E.g., the ARC algorithm
 - Megiddo and Modha, “[ARC: A Self-Tuning, Low Overhead Replacement Cache](#),” FAST 2003.

CLOCK Page Replacement Algorithm

- Keep a **circular list of physical frames** in memory
- Keep a **pointer** (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
 - ❑ During traversal, clear the R bits of examined frames
 - ❑ Set the hand pointer to the next frame in the list

