ECE 2300
Digital Logic & Computer Organization

Spring 2025

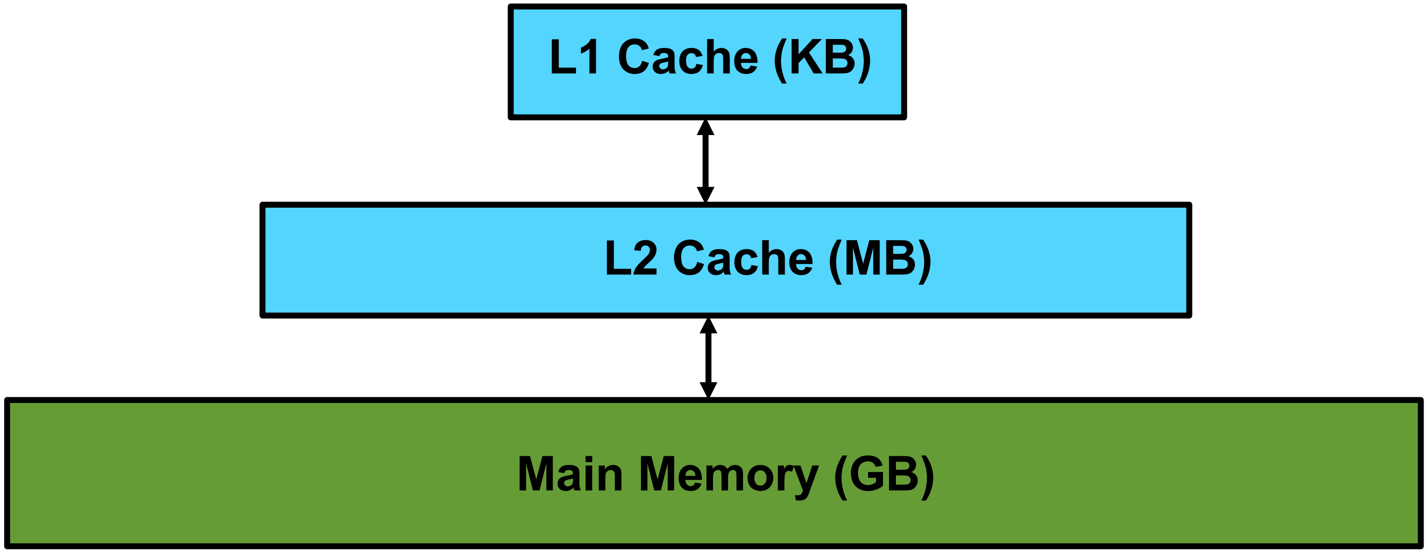
Virtual Memory

Cornell University

Announcements

- Lab 4 report due tomorrow

How Do We Extend and Share Main Memory?

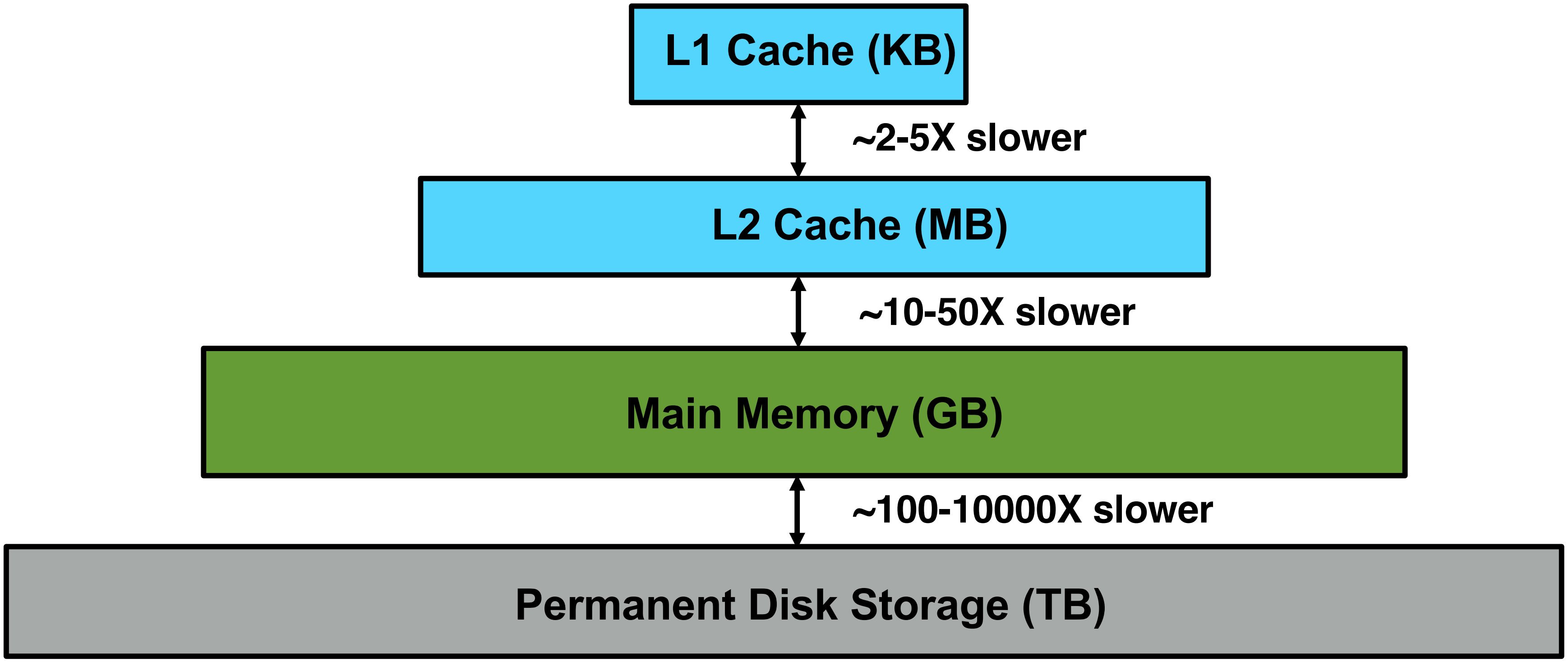


- What if one program needs more than the amount of installed main memory (i.e., physical memory)?

- How do multiple programs share the same main memory address space (multitasking) ?

Extending Memory Hierarchy

- Main memory (MM) is managed similar to a cache
  - – Data are brought into MM as requested
  - – If MM is full, older data get swapped out to disk



L1 Cache (KB)

~2-5X slower

L2 Cache (MB)

~10-50X slower

Main Memory (GB)

~100-10000X slower

Permanent Disk Storage (TB)

Data Transfer Granularity (Analogy)

- **Data transfer across the memory hierarchy increases in granularity**
  - **Bytes/words between RF and cache; blocks between cache and main memory; pages between disk and memory**

Registers / Cache (bytes or words)

Passing a pen between neighbors

Cache / MM (blocks)

Delivering boxes by car

MM / Disk (pages)

Transporting shipping containers by cargo ship

Data movement in the memory hierarchy is like transporting goods over different distances:

- Passing a few bytes between registers is like handing a pen to your neighbor—quick and lightweight
- Moving data blocks between main memory and cache is like delivering a box across town in a car
- Transferring pages between disk and memory is like shipping containers across the ocean—bulkier, slower, but more efficient per unit at that scale

Sharing Main Memory

- How to enable multiple programs to share the same physical MM?

Requirements

– *Transparency*: a program should not know other programs are sharing the same MM

– *Protection*: a program must not be able to corrupt other programs

Solutions (Virtualizing MM)

– Each program operates in its own virtual address space

– The set of physical MM addresses for each program is dynamically allocated and managed

Virtual Memory Intuition

Virtual Memory: Main Ideas

- The *hardware and software* mechanisms that dynamically manage the memory hierarchy

- Extends memory hierarchy to incorporate large permanent storage
  - Hide physical size of MM from software
  - Moves large blocks (in unit of <u>pages</u>) between MM and permanent storage as needed

- Allows multiple programs (via processes) to share main memory with protection and isolation
  - A <u>process</u> is an active running instance of a program
  - Processes run in virtual address space

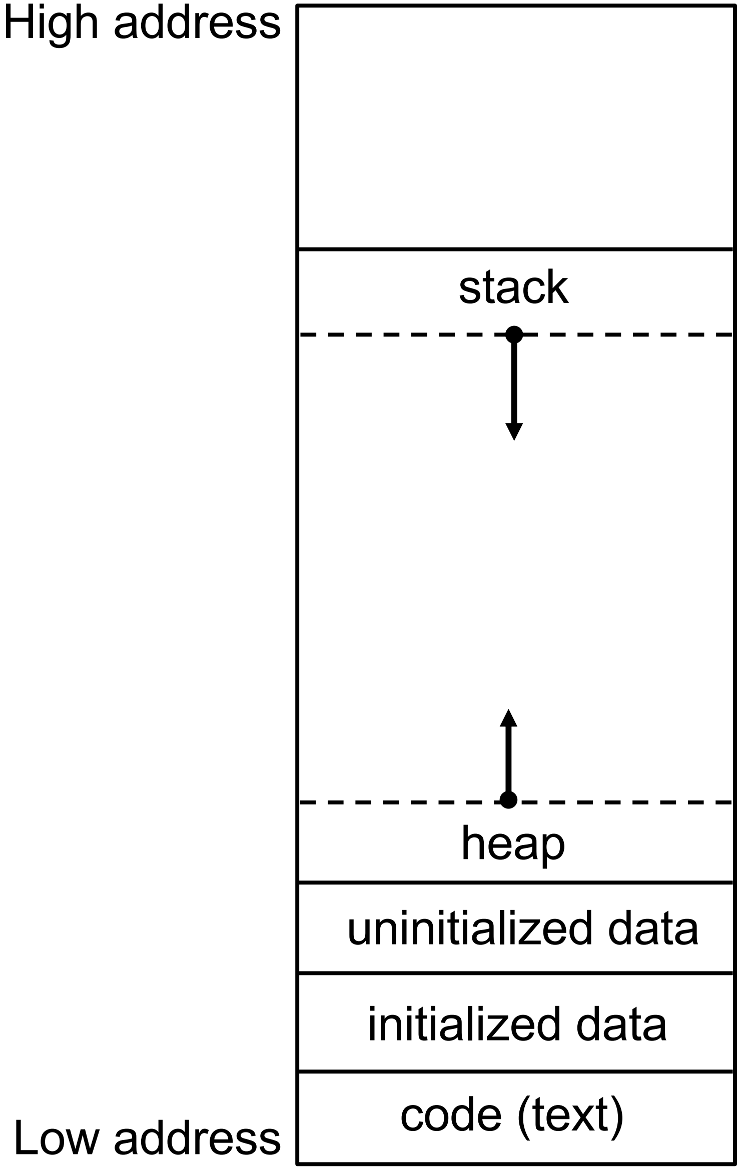
Virtual and Physical Addresses

- When a program is compiled, the instruction and data addresses are *virtual*
  - They need to translated to the *physical* addresses

- Virtual addresses refer to the addresses used by the programs
  - With a N-bit virtual address, the size of the virtual address space is 2^N bytes

- Physical addresses refer to the real addresses used by hardware to access the physical MM
  - With a M-bit physical address, the size of the physical address space is 2^M bytes (typically, $M < N$)

Virtual Address Space of a Program (Process)*



* Supplementary material
(not included in the final exam)

Code Section (or Text) contains the executable code of the program, i.e., instructions to be executed by the processor.

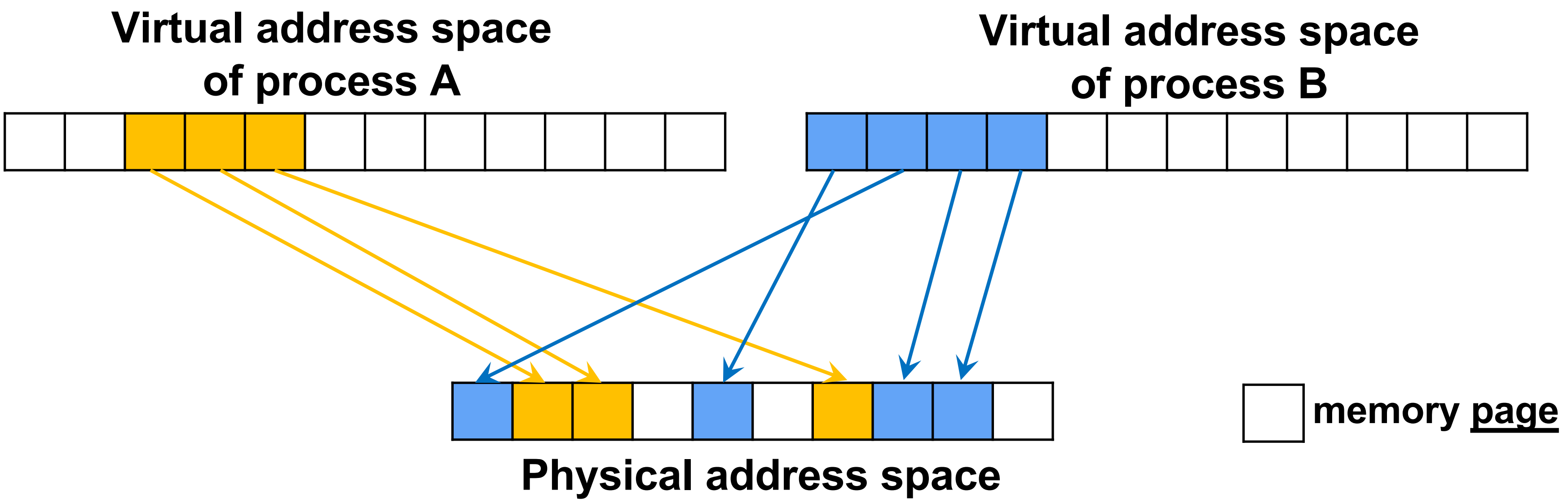
Initialized Data Section contains initialized global & static variables used by the program.

Uninitialized Data Section contains uninitialized global & static variables.

Heap is a memory area where memory is dynamically allocated & deallocated at runtime.

Stack is used for local variables and function call information, which grows & shrinks dynamically as functions are called & return.

Physical Memory Sharing with Virtual Memory



Virtual address space of process A

Virtual address space of process B

memory page

Physical address space

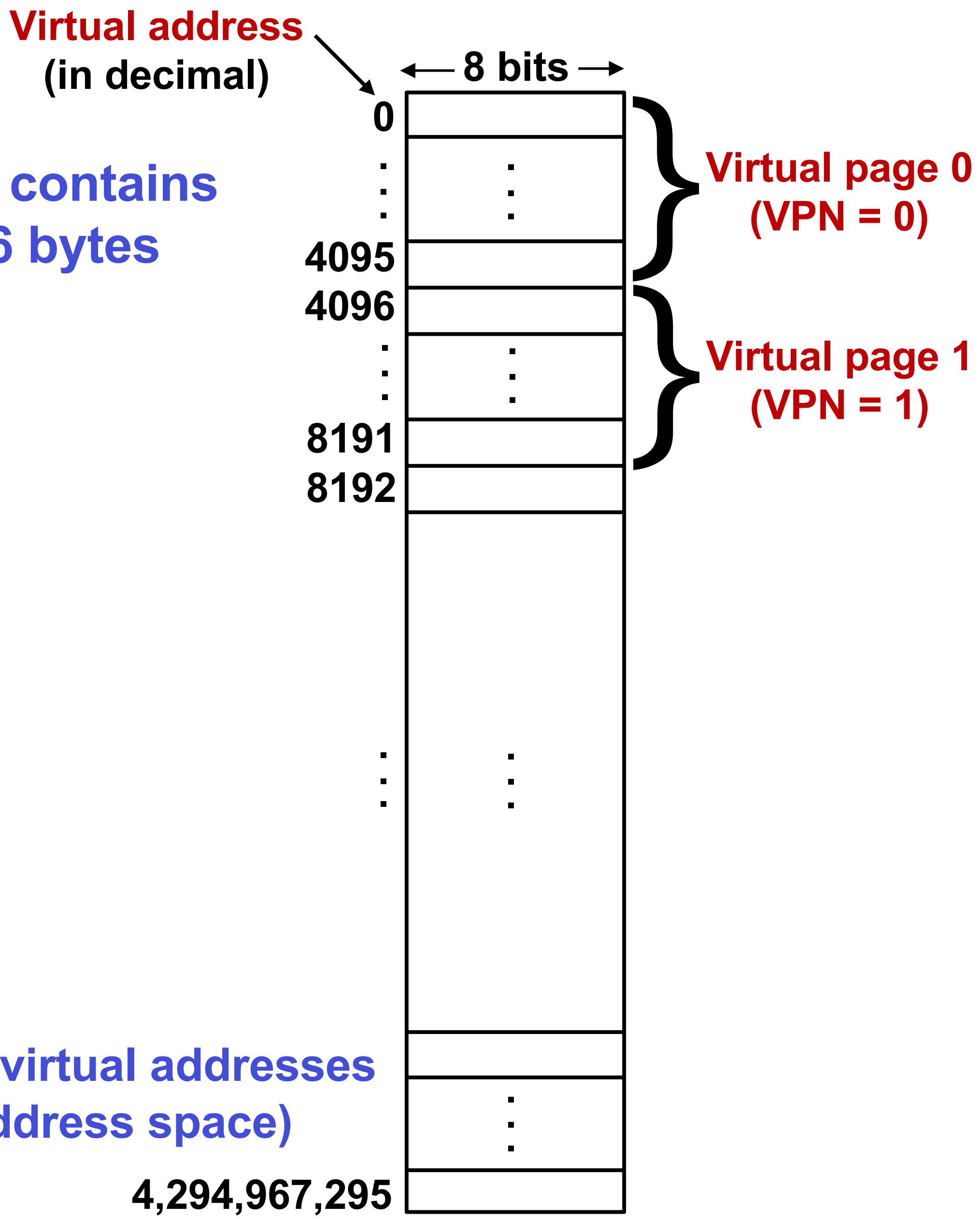
- **Each process (program) has its own virtual address space**
  - Allows developers to write software as if it owns all of the computer's memory
  - At any given time, only portions of each process's virtual memory need to reside in physical MM, due to data locality
- **The OS allocates pages and "multiplexes" the physical MM across processes**
  - Without virtual memory, could have only one process in MM at a time

Paging

- Virtual/physical address space is divided into equal sized pages
  - A page contains N bytes where N is a power of 2
    - N = 4096 is a typical size
  - A whole page is read or written during data transfer between MM and disk
  - Each page in virtual memory space has a unique index called *virtual page number* (VPN)
  - Similarly, each page in physical memory space has a unique *physical page number* (PPN)

View of Virtual Memory with 32b Address

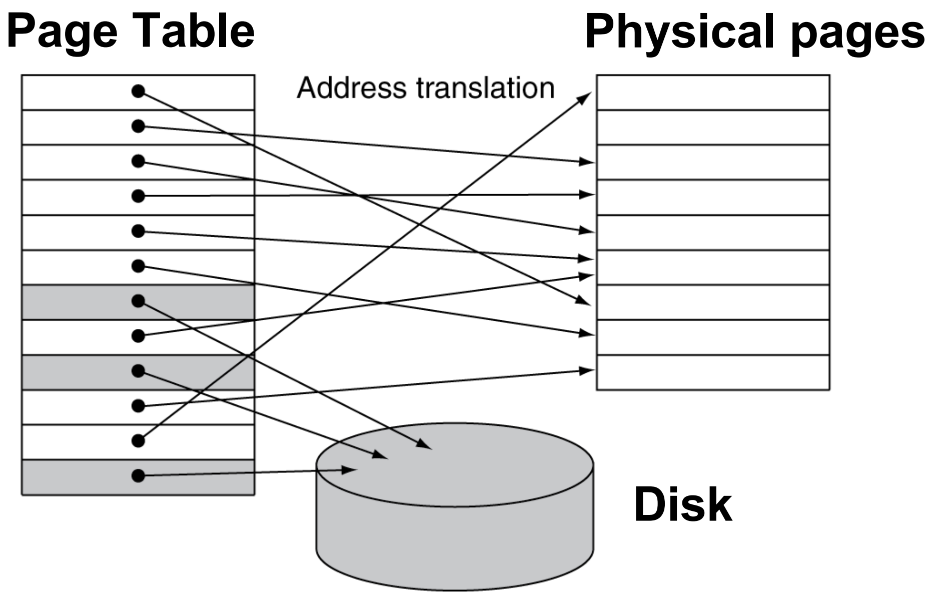
Each page contains
4KB = 4096 bytes



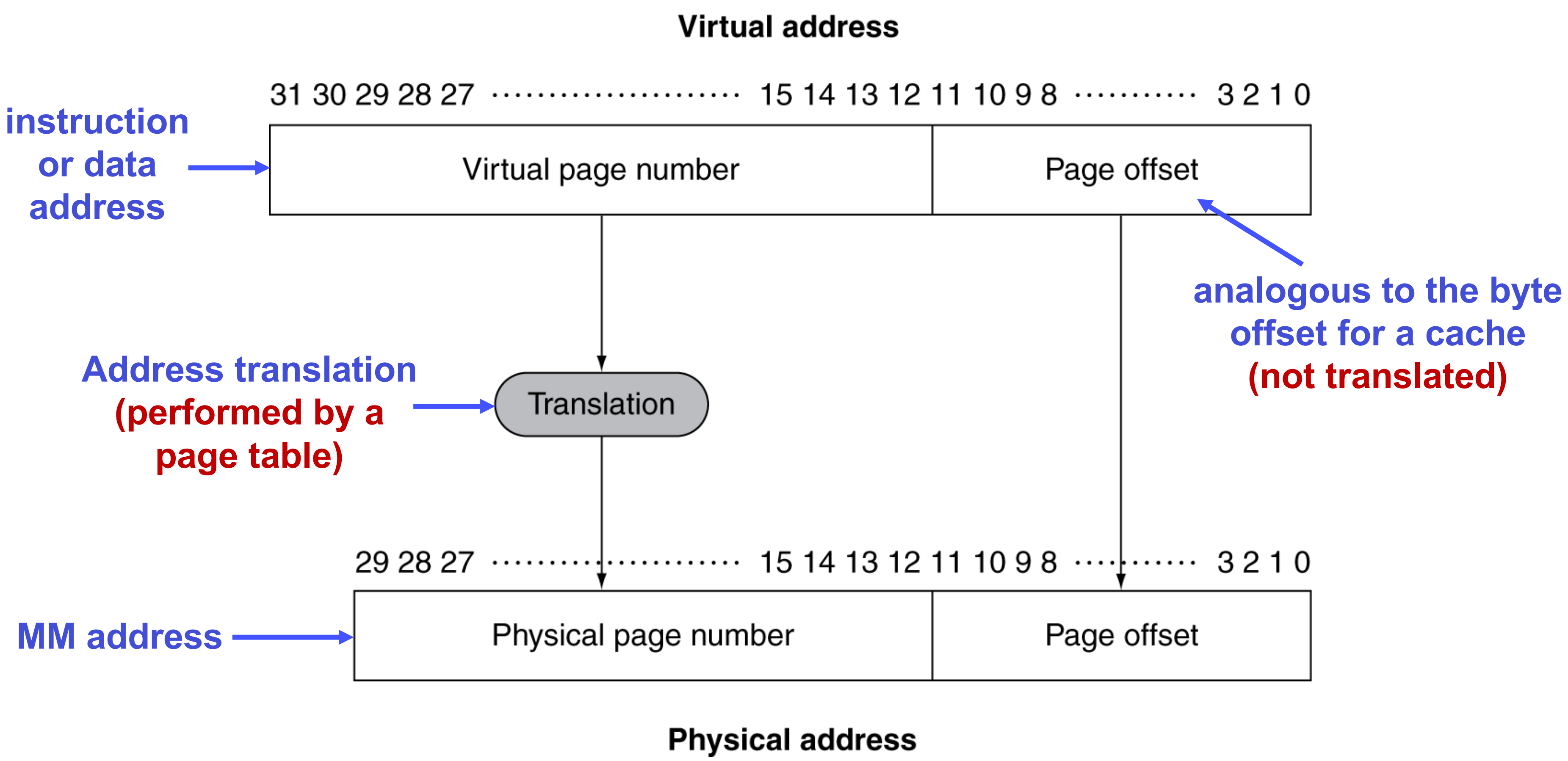
Assuming 32-bit virtual addresses
(4GB of virtual address space)

Virtual Memory and Physical MM

- During a program execution, only a subset of its virtual pages need to be in physical main memory (MM) at a time
  - When requested, if the page is not already in MM, the OS loads an entire page from disk into a physical memory location
  - The mapping between virtual to physical pages is saved in a directory called page table (which is stored in physical MM)
    - When the same virtual address is encountered, it is *translated* using this saved mapping information in the directory



Address Translation



Virtual address

31 30 29 28 27 ······················ 15 14 13 12 11 10 9 8 ··········· 3 2 1 0

Virtual page number

Page offset

instruction or data address

analogous to the byte offset for a cache (not translated)

Address translation (performed by a page table)

Translation

29 28 27 ······················ 15 14 13 12 11 10 9 8 ··········· 3 2 1 0

MM address

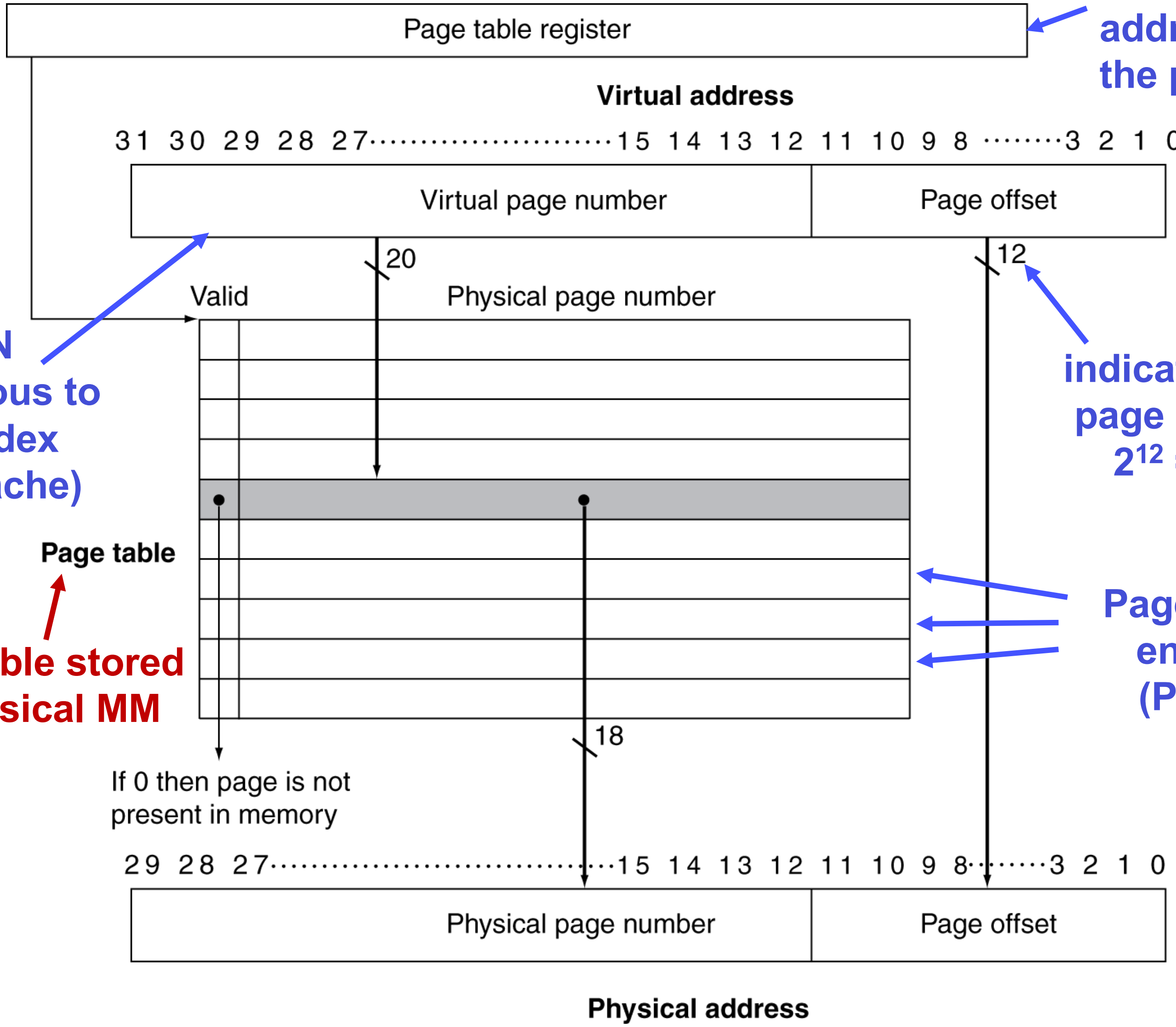
Physical page number

Page offset

Physical address

Assuming 1GB physical memory here (30-bit physical address)

Address Translation Using a Page Table



A special CPU register that holds a physical address for locating the page table

VPN (analogous to the index for a cache)

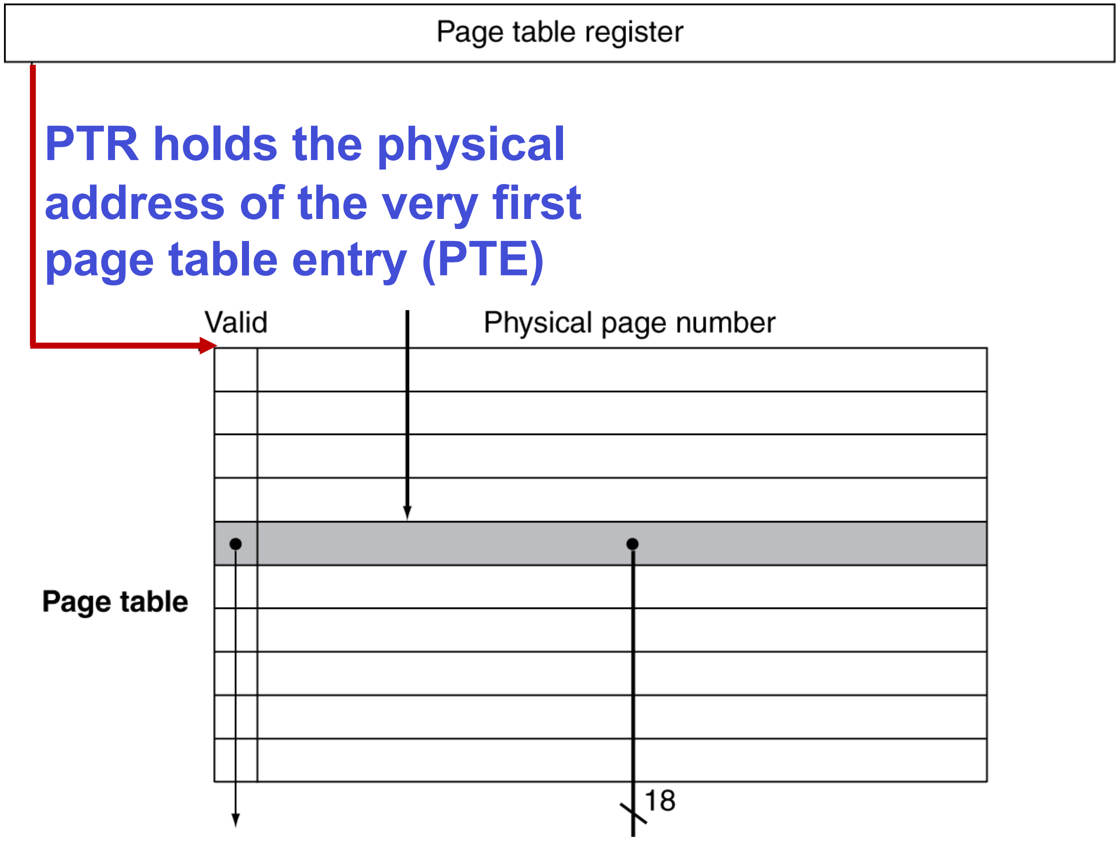
indicates that a page contains 2^{12} = 4 KB

Page table entries (PTEs)

Page table stored in physical MM

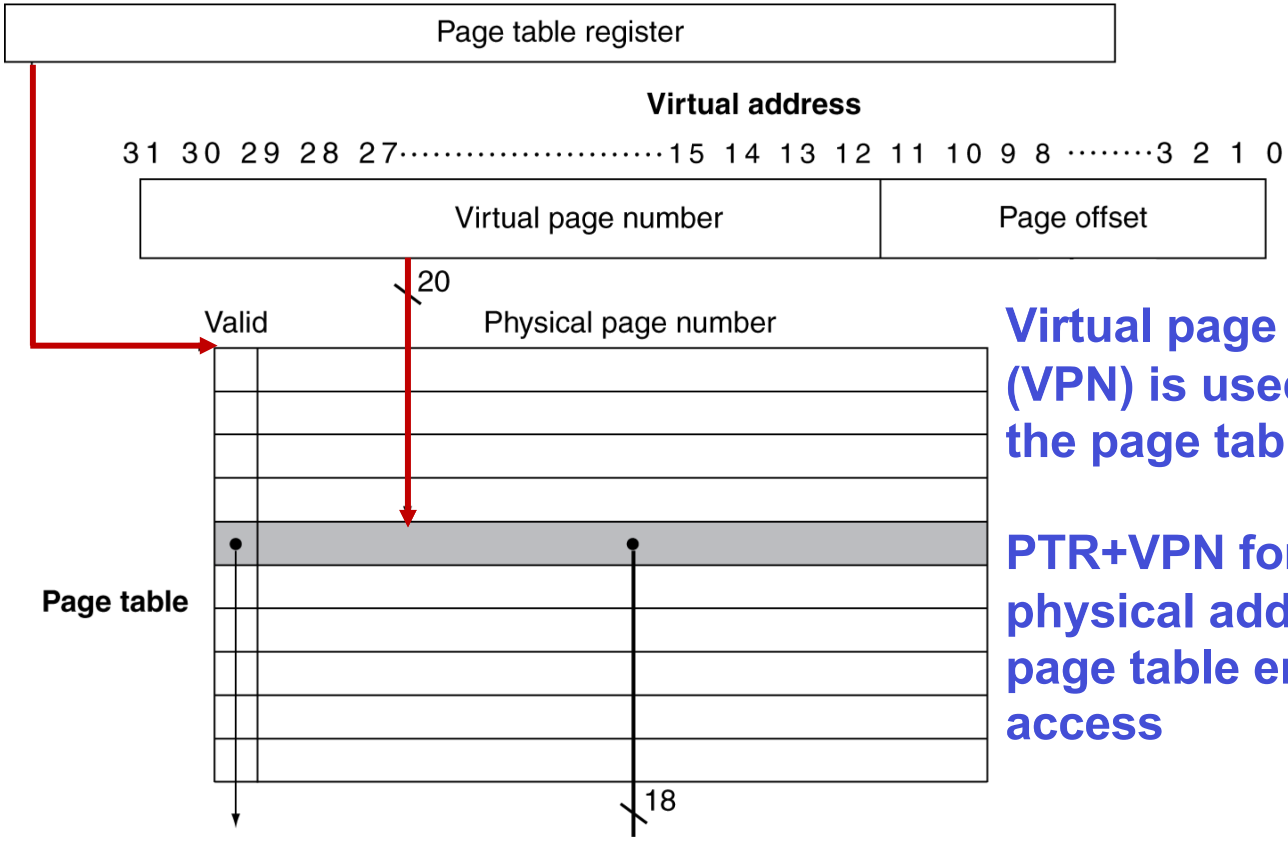
Breaking Down Page Table Operation (1)

The Page Table Register (PTR) is a special CPU register for locating the page table in the physical MM



PTR holds the physical address of the very first page table entry (PTE)

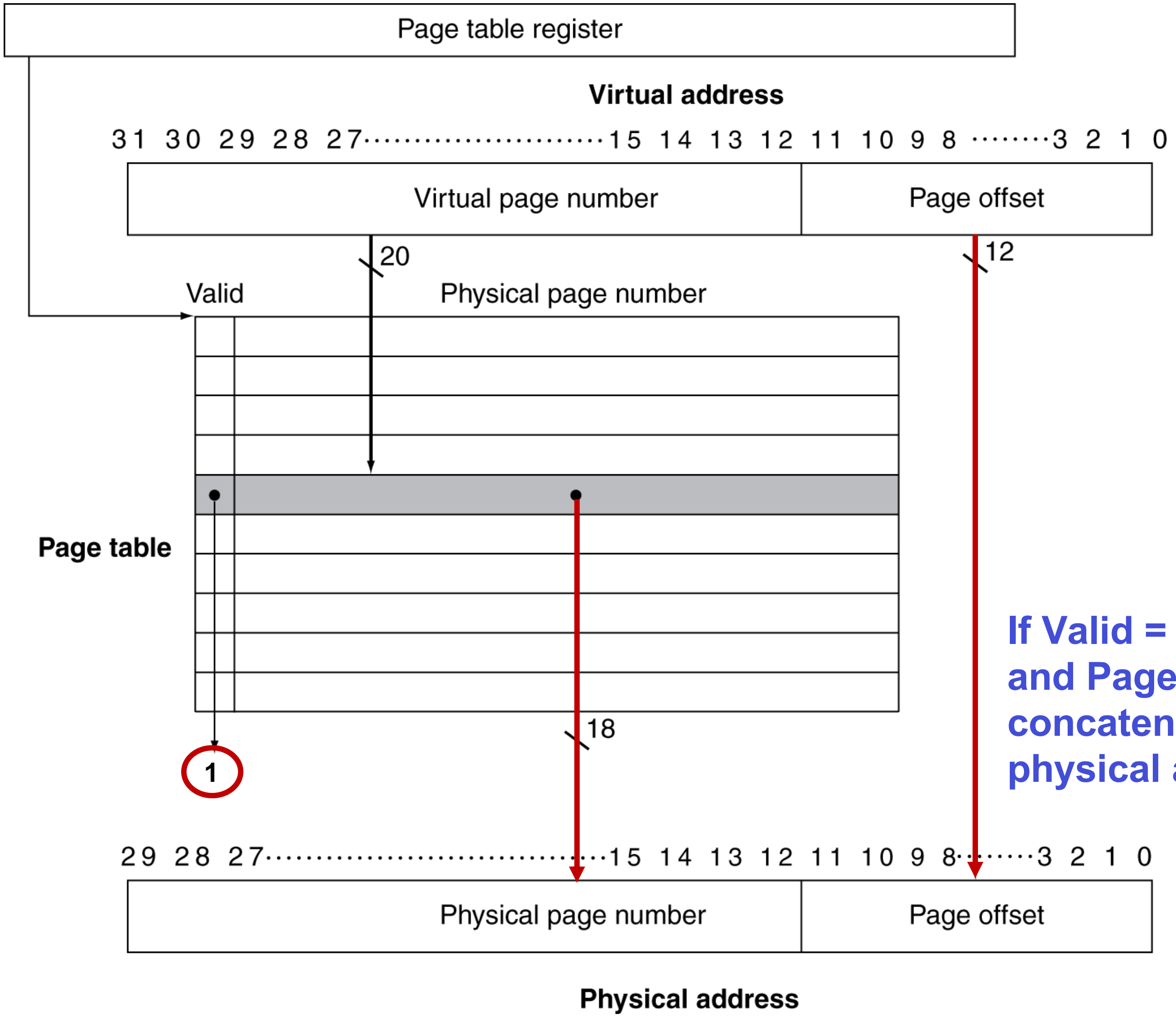
Breaking Down Page Table Operation (2)



Virtual page number (VPN) is used to index the page table;

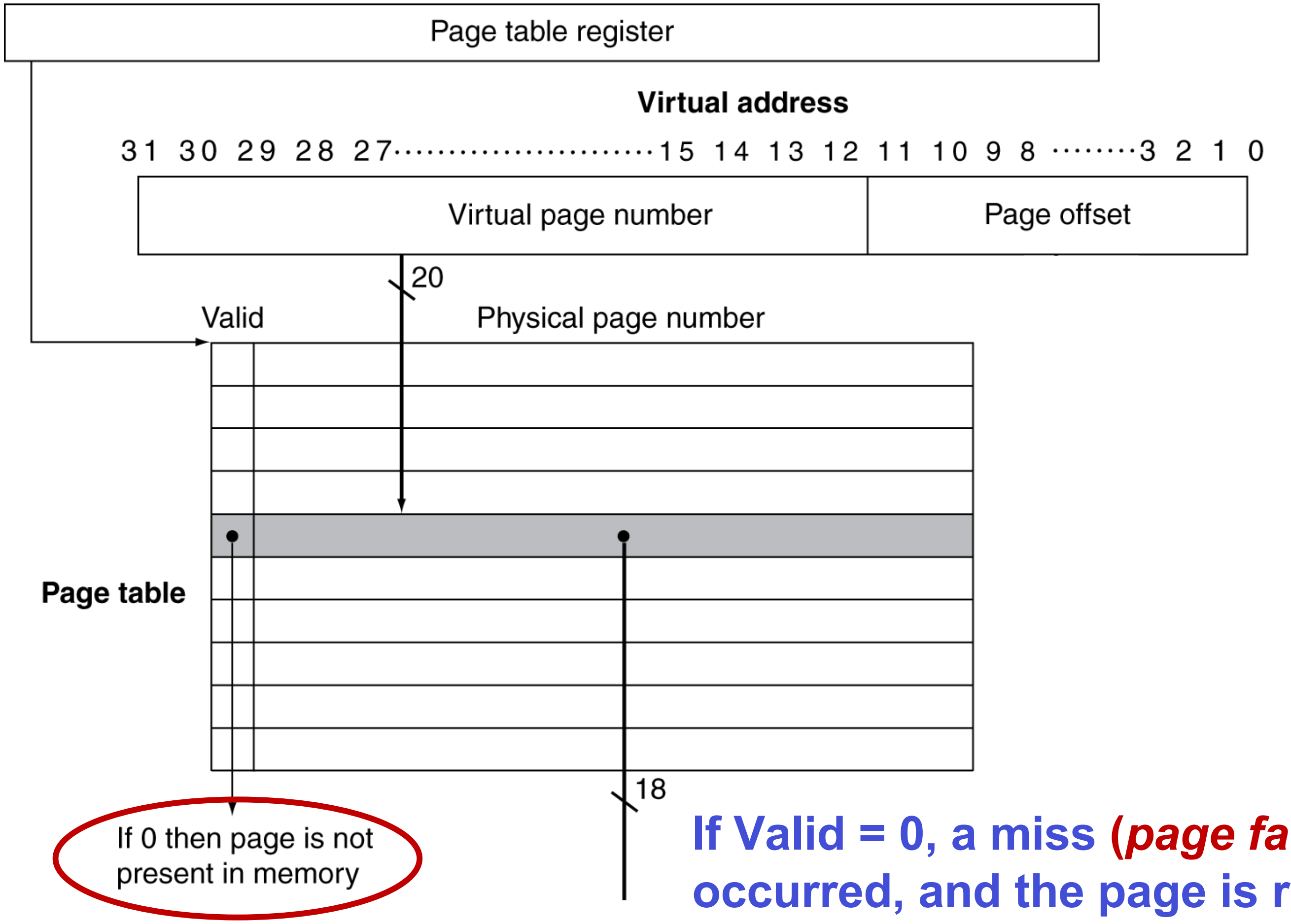
PTR+VPN form the physical address of the page table entry (PTE) to access

Breaking Down Page Table Operation (3)



If Valid = 1, then the PPN and Page Offset are concatenated to form the physical address

Breaking Down Page Table Operation (4)



If Valid = 0, a miss (***page fault***) has occurred, and the page is read from disk into MM (replacing another page if the MM is full)

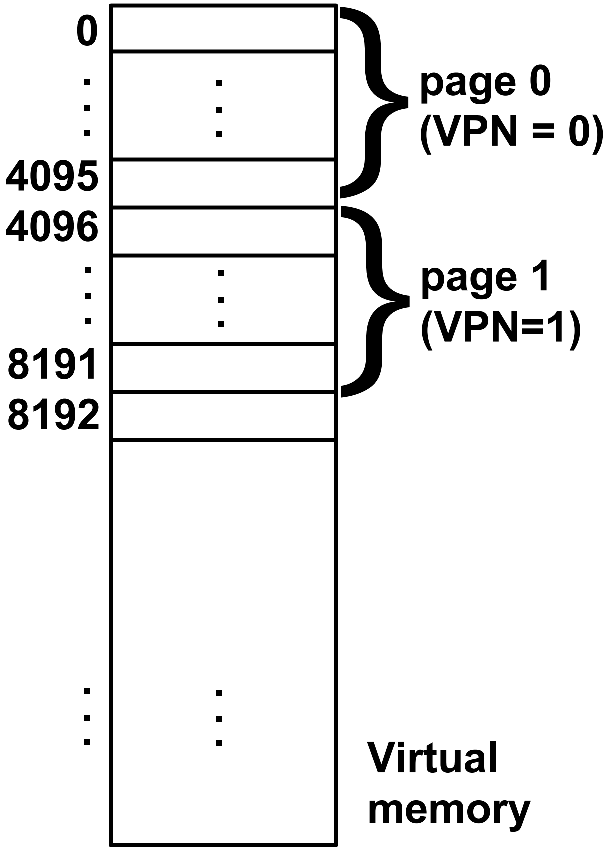
Example: Page Table Access

- Given the following page table and virtual address stream (in decimal), identify the potential page faults

128, 2048, 4096, 8192

	Valid	Physical Page#
PTE0 →	0	Disk
PTE1 →	1	16
PTE2 →	0	Disk
	1	4
	…	…

Page Table



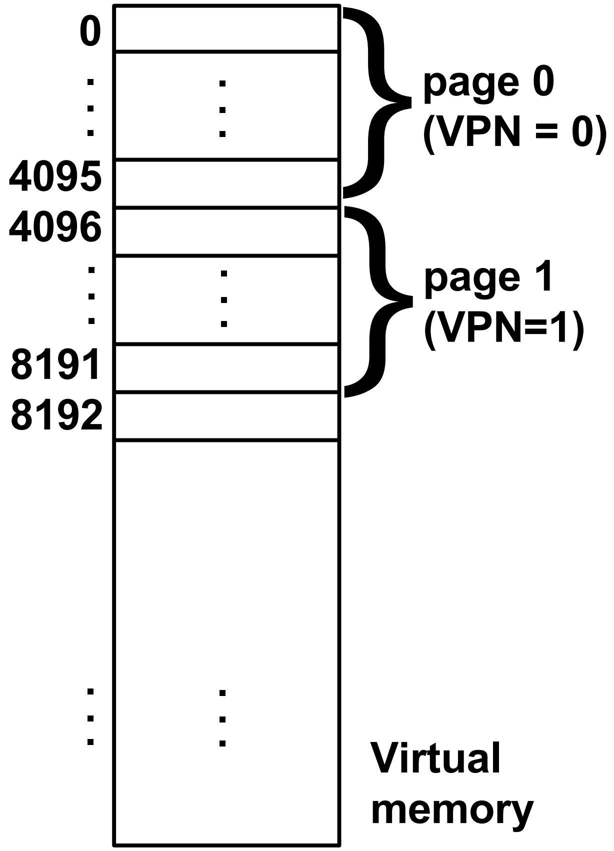
Example: Page Table Access

- Given the following page table and virtual address stream (in decimal), identify the potential page faults

128 (VPN=0), 2048 (VPN=0), 4096 (VPN=1), 8192 (VPN=2)

	Valid	Physical Page#
PTE0 →	0	Disk
PTE1 →	1	16
PTE2 →	0	Disk
	1	4
	…	…

Page Table



Page Faults and Page Replacement

- Miss penalty on a page fault is significant
  - Up to ~100M cycles

- Low miss (page fault) rates are essential
  - Fully associative page placement (put anywhere in MM)
  - LRU replacement of a page when MM is full

- The Operating System (OS) handles page placement

Page Replacement and Write Policy

- Too expensive to do true LRU (100K-1M pages); Use LRU approximation
  - Each PTE has a Reference bit (ref)
  - Reference bit is set when a page is accessed
  - OS periodically clears all Reference bits
  - OS chooses a page with a Reference bit of 0

- Write back policy is used (instead of write through)
  - Dirty bit in PTE is set on a write to main memory
  - Page with set Dirty bit is written to disk if replaced

Faster Address Translation

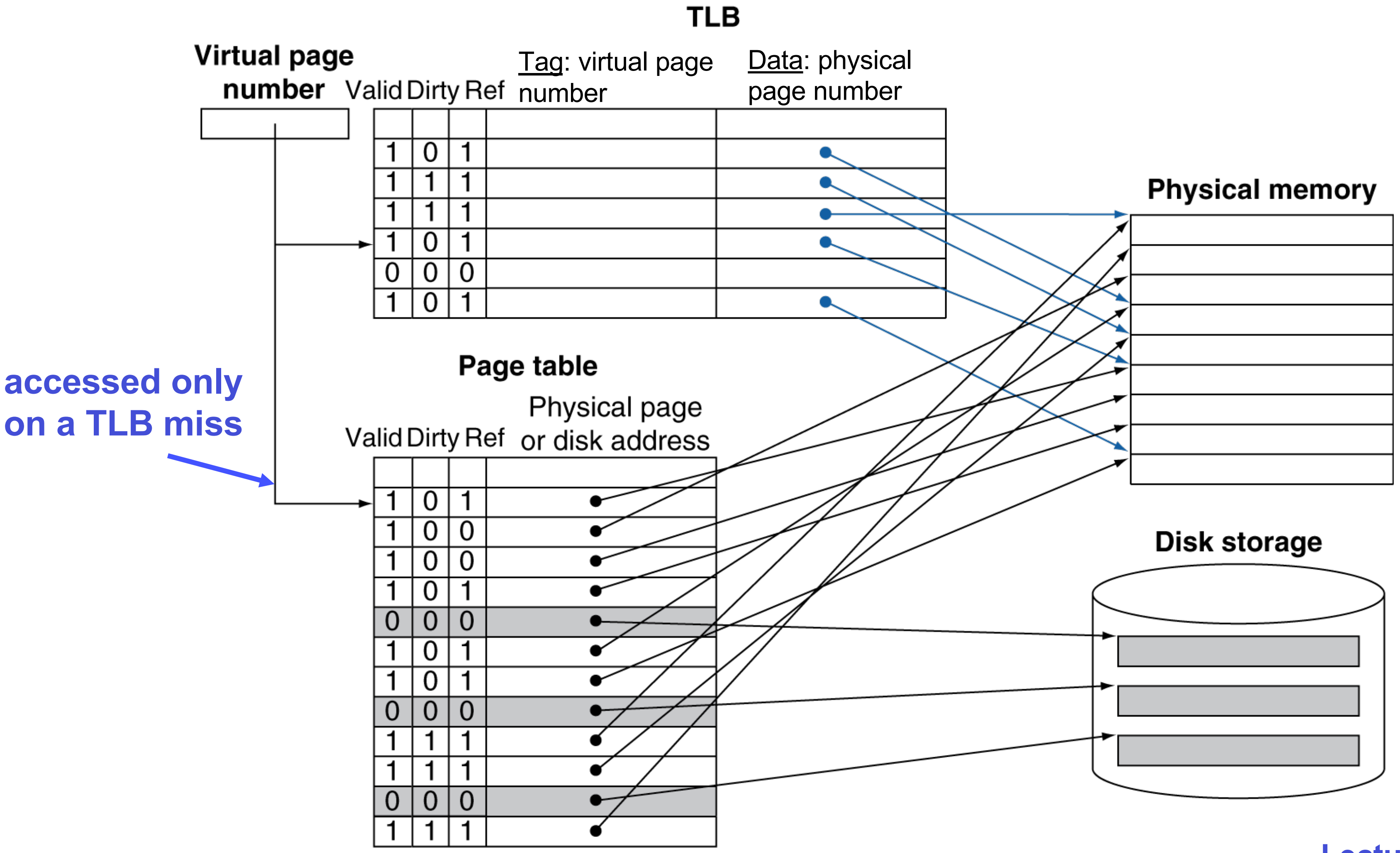
- **Must access the page table before an instruction can be fetched and before data cache/memory can be accessed**

- **Page table accesses have good locality**

⇒ Cache the most recent PTEs within the CPU

Translation Lookaside Buffer (TLB)

- Small cache of recently accessed PTE (typically 16-512 entries, fully associative)



accessed only on a TLB miss

TLB

Valid	Dirty	Ref	Tag: virtual page number	Data: physical page number
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

TLB Miss Scenarios

- TLB miss, page table hit
  - Bring in the PTE information from page table to TLB
  - Retry the access
  - Usually handled by hardware (the memory management unit, or MMU)
- TLB miss, page fault
  - Bring in the page from disk (orchestrated by OS)
  - Load the page table and TLB (orchestrated by OS)
  - Retry the access
    - Cache miss will definitely occur!

Next Class

Exceptions

Inputs/Outputs

(H&H 6.6.2, 9.2, 9.3.8)