CS 4400
Computer Systems

LECTURE 18

Virtual memory

Main memory as a cache

Address translation

Virtual Memory

- ***Virtual memory*** (VM) is an abstraction of main memory
- Treats main memory (MM) as a cache for disk
  - Transfers data back and forth between disk and MM, as needed
- Simplifies memory management
  - provides each process with a uniform address space

Virtual Memory

- Protects the address space of each process
  - prevents some process from inadvertently writing to the memory used by another process
- Works silently and automatically
  - *without any intervention* from the application programmer

Why Care About VM

- VM pervades all levels of computer systems
  - hardware exceptions, linkers, loaders, processes, …
- VM gives applications powerful capabilities.
  - create/destroy chunks of memory, map chunks of memory to portions of disk, share memory with other processes

Why Care About VM

- VM, used improperly, can lead to difficult bugs
  - any variable reference, pointer dereference, or `malloc` call uses VM
  - possible behaviors: "Segmentation fault"; long, silent run before crashing; run to completion with incorrect results

Physical Addressing

- MM is an array of *M* contiguous byte-sized cells
  - each byte has a unique ***physical address*** (PA)
- ***Physical addressing*** is using the PA to access memory
  - used by early PCs, embedded microcontrollers, and others

Physical Addressing

- Executes a load instruction
  - Generate a PA ***x***
  - Passes ***x*** to main memory (via memory bus).
- Main memory
  - Fetches the word starting at PA ***x***
  - Returns it to the CPU
  - Stores in a register

Virtual Addressing

- Another form of addressing uses a ***virtual address*** (VA).
- ***Virtual addressing*** is using the VA to access memory.
  - used by modern processors for general-purpose computing

Virtual Addressing

- On load:
  - CPU generates a VA ***y***
  - ***y*** is converted to the appropriate PA ***x***
  - Passed to main memory
- The translation of VA to PA requires close cooperation between the CPU and OS
  - memory management unit (MMU)—dedicated hardware for translating VAs, using look-up table stored in memory

Address Space

- ***Address space***—an ordered set of integer addresses (> 0)
  - if addresses are consecutive, the address space is *linear*
- ***Virtual address space***—$N = 2^n$ virtual addresses.
  - *n* is the number of bits needed to represent the largest address
  - typically, modern virtual address spaces are 32-bit or 64-bit
- ***Physical address space***—corresponds to the $M = 2^m$ bytes of physical memory in the system.

Main Memory as a Cache

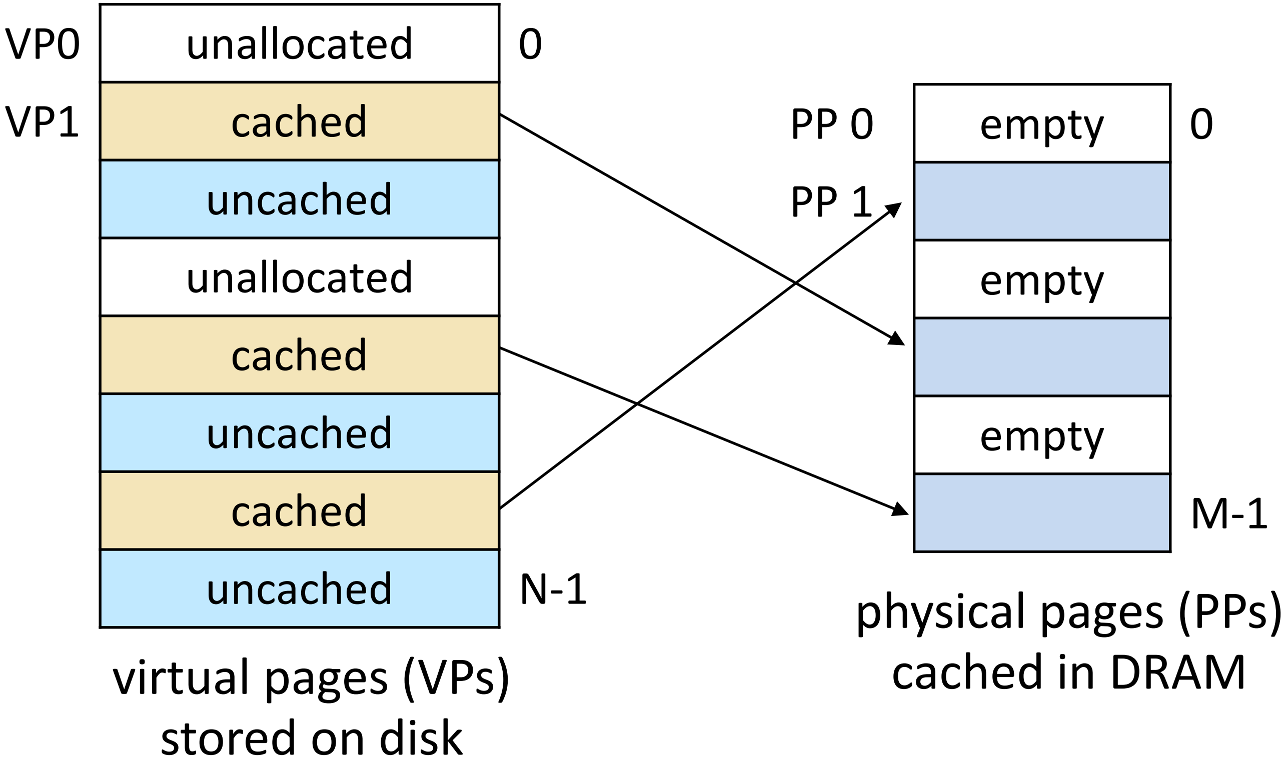
- Think of virtual memory as an array of *N* contiguous byte-sized cells stored on disk.
  - each byte has a unique VA that is an index into the array

- Data on disk is partitioned into blocks (called pages) that serve as the transfer units.

- page size $P=2^p$

- How many VPs?

- How many PPs?



virtual pages (VPs) stored on disk

physical pages (PPs) cached in DRAM

DRAM Cache Organization

- Misses in DRAM caches are much more expensive than those in SRAM caches.
  - DRAM is ~10 times slower than SRAM
    - DRAM: ~10 ns / SRAM: ~1 ns
  - Spinning disk is ~1M times slower than DRAM
    - Disk: ~10 ms / DRAM: ~10 ns
    - Cost of reading the first byte from a (spinning) disk sector is ~100K times slower than reading successive bytes in the sector
  - Read cost basically uniform on SSD, still gap of ~10k
    - SSD: ~100 microseconds / DRAM: ~10 ns

DRAM Cache Organization

- The organization of DRAM caches is driven by ***the large miss penalty***
  - DRAM caches are fully associative
  - virtual pages are large (4-8 KB)

Page Table

- If a VP is cached in DRAM, which physical page is it cached in?
- If there is a miss:
  - Where is the virtual page stored on disk?
  - Which page in physical memory is the victim?

Page Table

- ***Page table***
  - Data structure stored in physical memory that maps virtual addresses to physical addresses
- Address translation hardware reads the page table
- The OS maintains the contents of the page table and transfers pages between disk and DRAM.

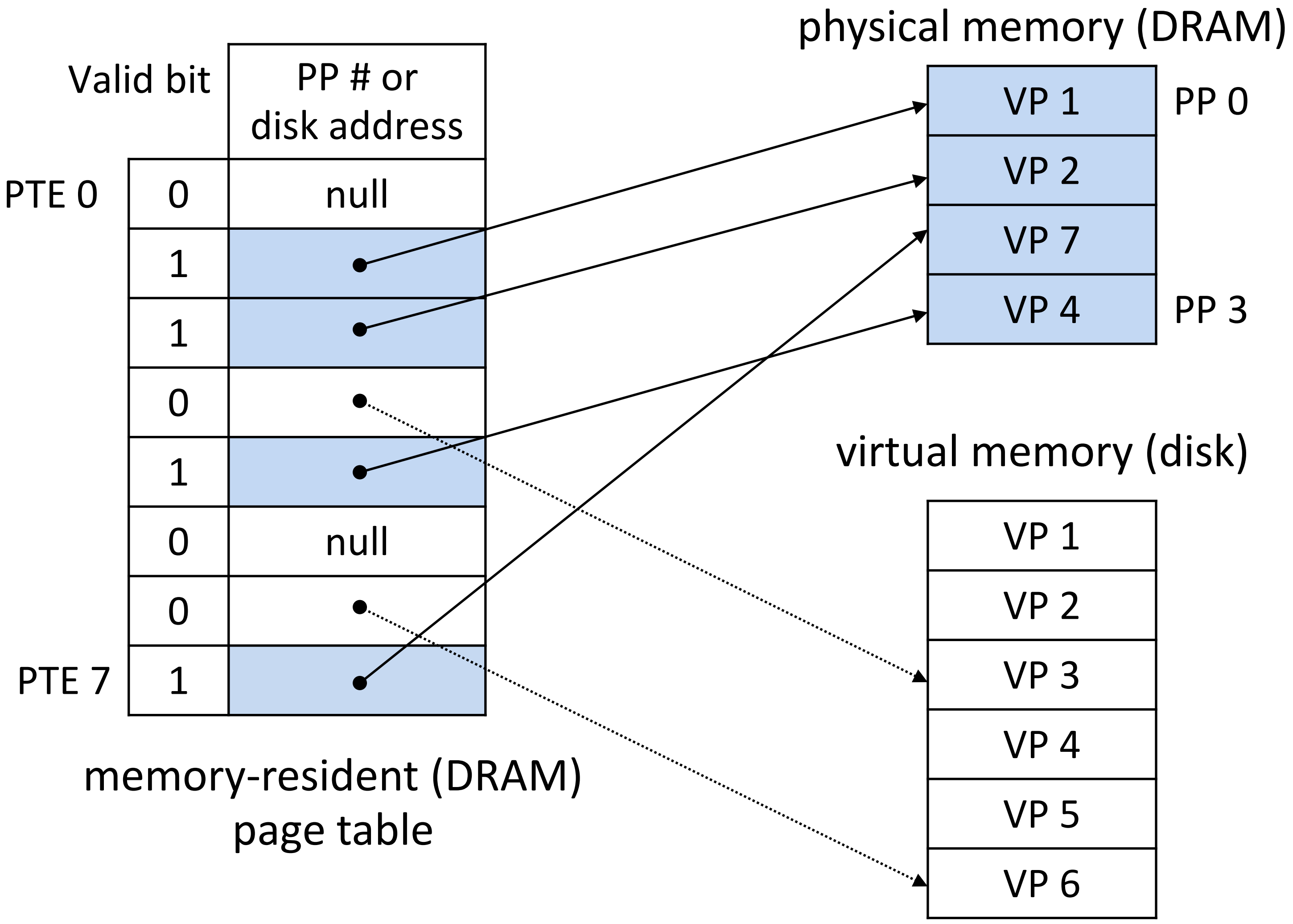
Page Table Entries

- A page table is array of ***page table entries*** (PTEs).
- Each VP in the virtual address space has a PTE at a fixed offset in the page table.

Page Table Entries

- Assume that each PTE consists of a *valid bit* and a *k*-bit *address field*.
- Valid bit indicates if the VP is currently cached in DRAM
  - if *valid bit = 1*
    - Address field → PP in DRAM
  - if *valid bit = 0*
    - Null address → unallocated VP
    - Non-null address → start of VP on disk

Example: Page Table



	Valid bit	PP # or disk address
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

memory-resident (DRAM)
page table

physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

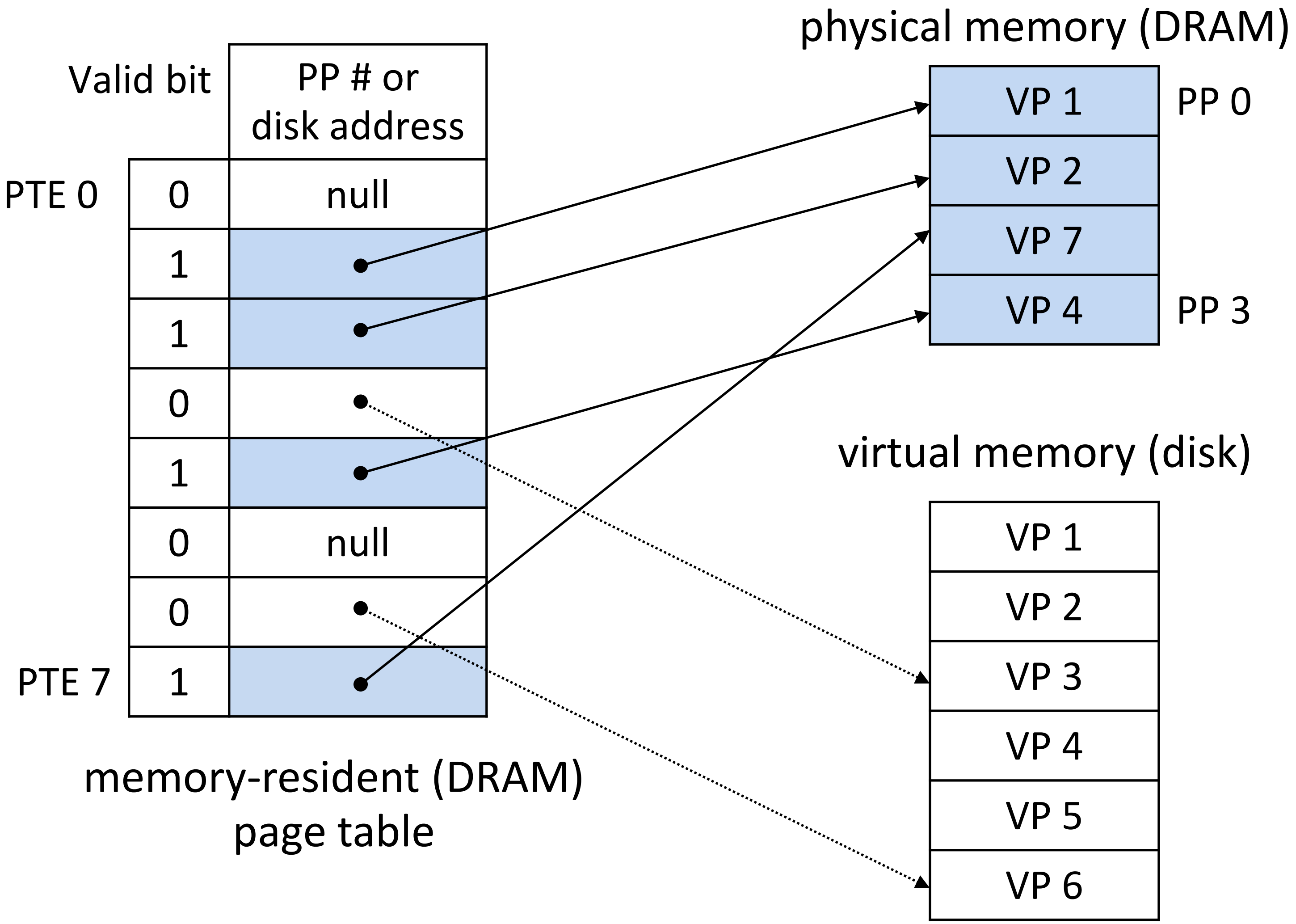
virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6

Page Hit

- The ***memory management unit*** (MMU) is dedicated hardware on the CPU that uses the virtual address to locate and read data from memory

Example: Page Table



	Valid bit	PP # or disk address
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

memory-resident (DRAM)
page table

physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6

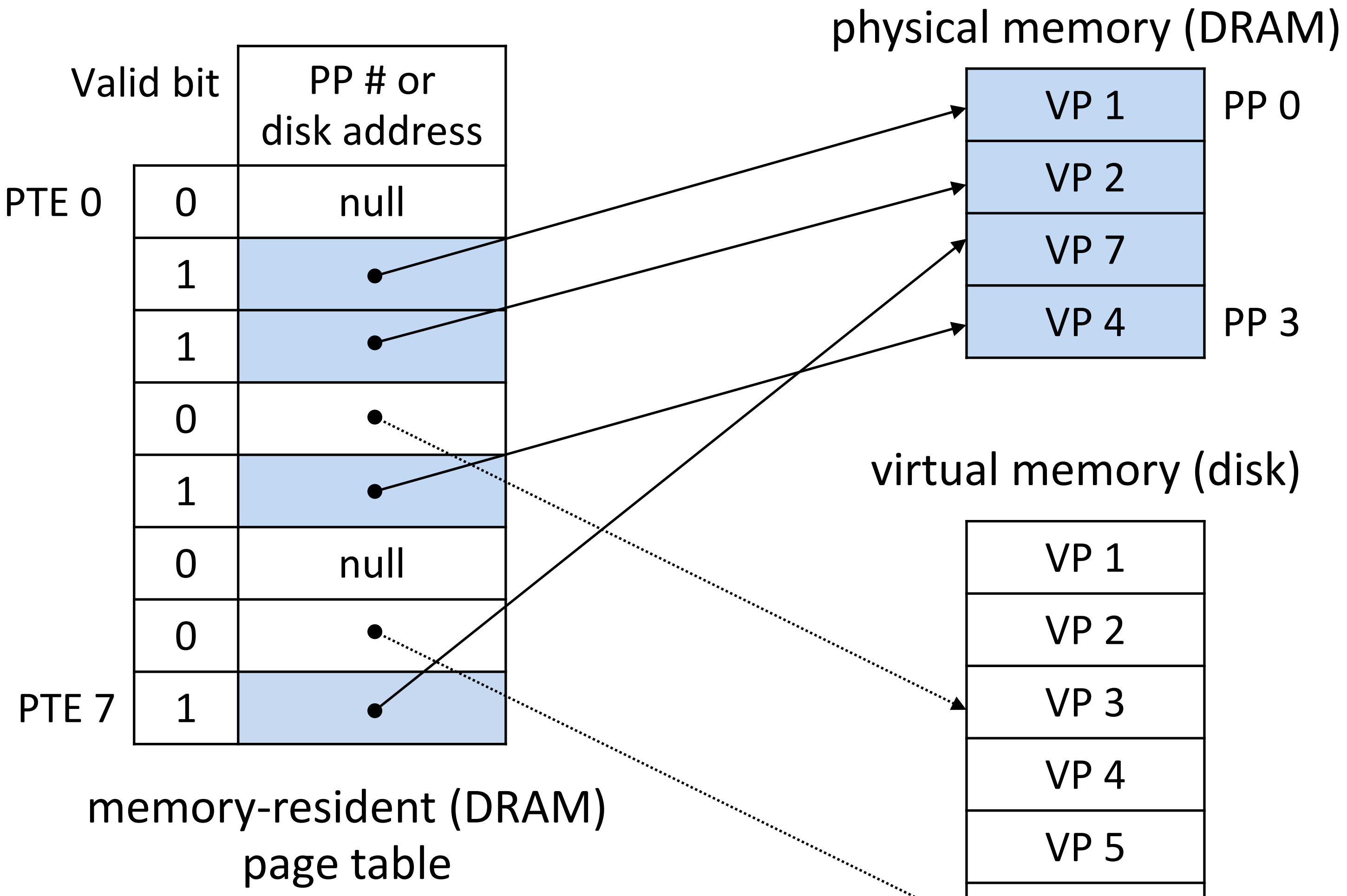
What happens on a read to VP 2?

Page Hit

CPU reads a word of virtual memory contained in VP 2 (cached in DRAM):

- Valid bit of PTE 2 is set
  - MMU knows VP 2 is cached, a ***Page Hit*** occurs
- MMU gets address of the cached page (PP 1) from the address field of PTE 2

Example: Page Table



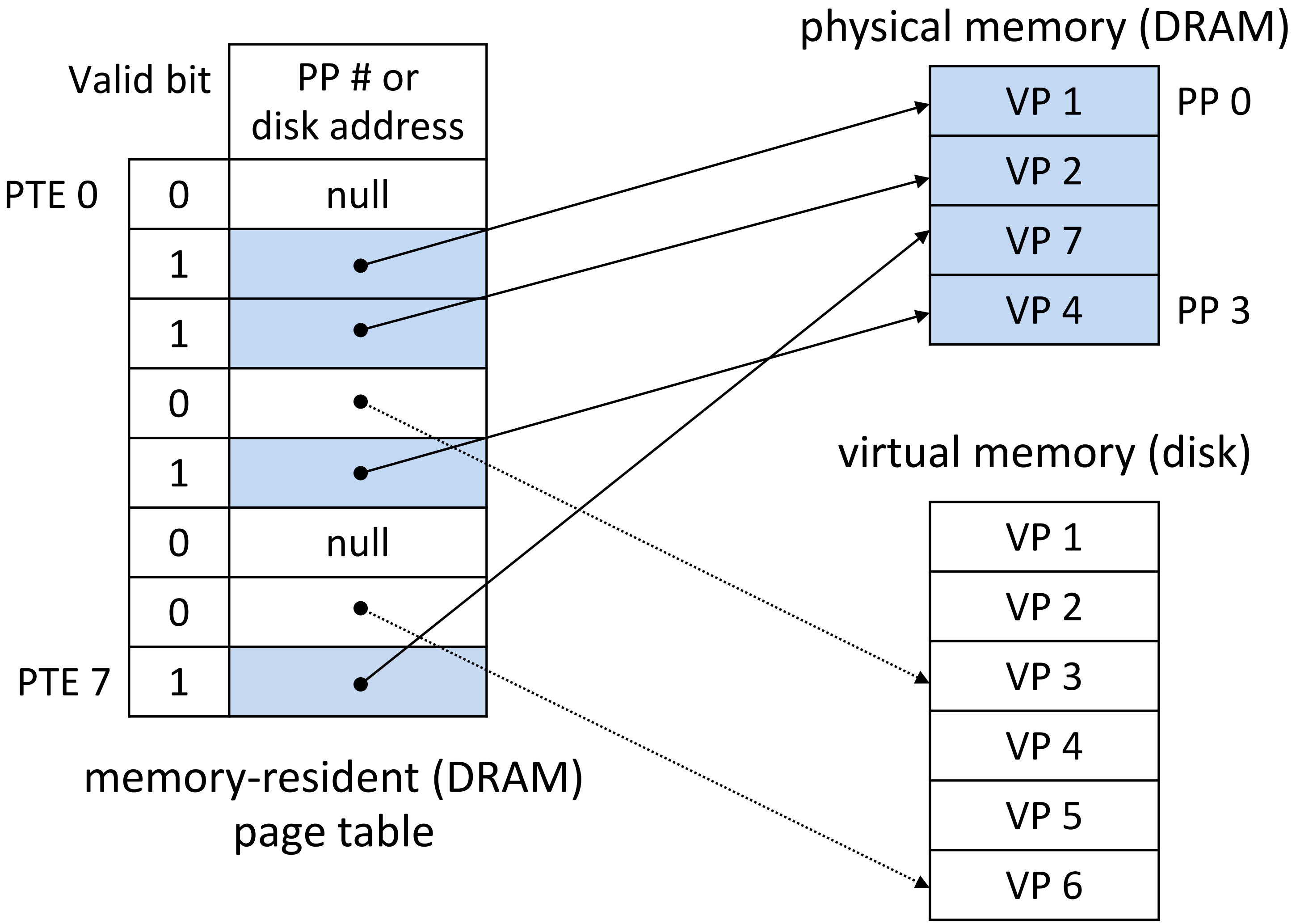
What happens on a read to VP 3?

Page Fault

When the CPU reads a word of VM contained in VP 3:

- The valid bit in PTE 3 is not set
  - Indicates to MMU that VP 3 is uncached, a ***page fault*** occurs
  - Triggers a ***page fault exception*** in kernel
- The page fault exception handler selects a victim page (say, VP 4) and swaps.
- The kernel modifies PTE 4 and PTE 3
- At handler return, faulting instruction restarts, resulting in a page hit

Example: Page Table



physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

	Valid bit	PP # or disk address
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

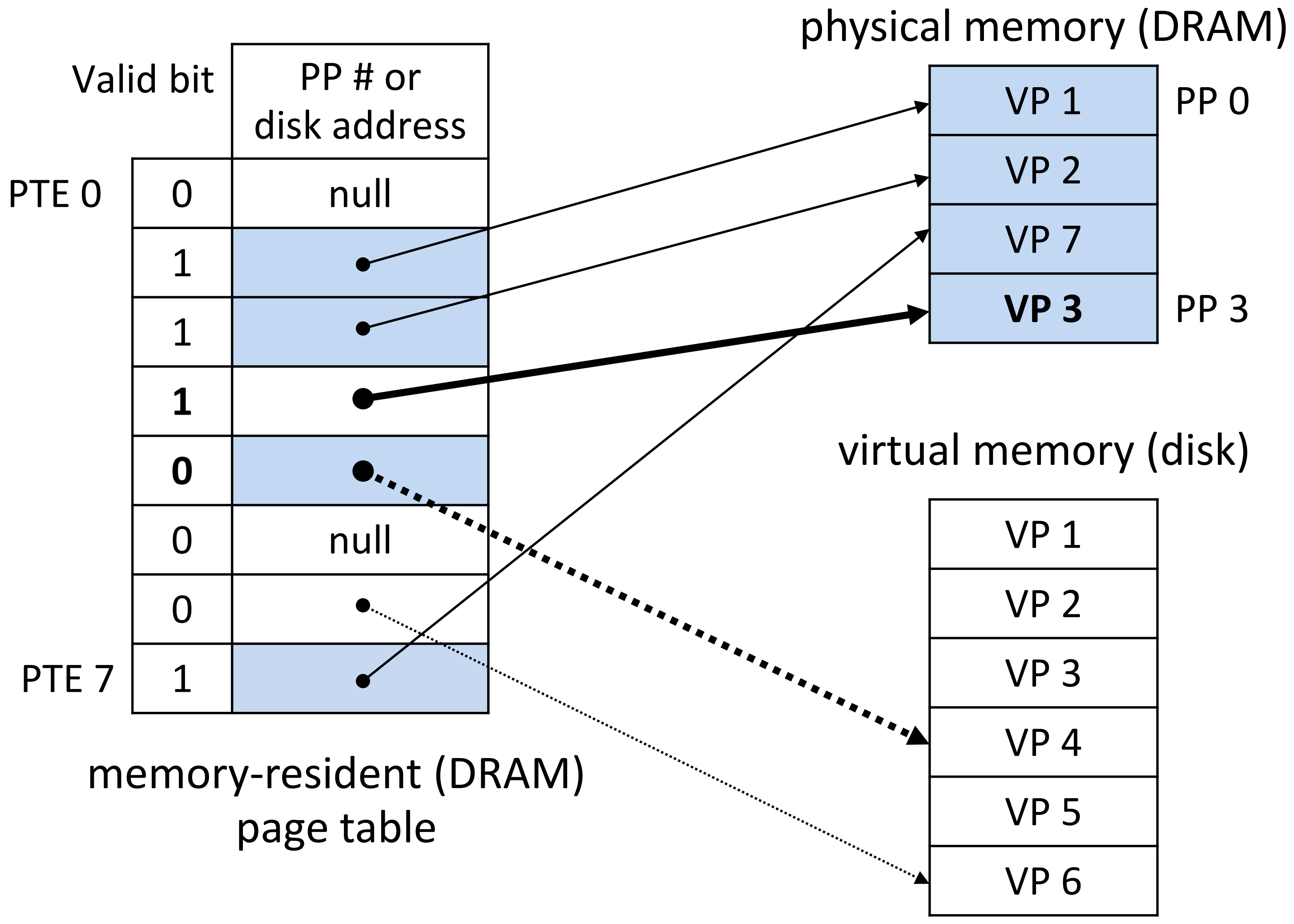
memory-resident (DRAM)
page table

virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6

Read of VP 3 caused page fault

Example: Page Table



	Valid bit	PP # or disk address
PTE 0	0	null
	1	●
	1	●
	1	●
	0	●
	0	null
	0	●
PTE 7	1	●

memory-resident (DRAM) page table

physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 3	PP 3

virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6

VP 3 replaced VP 4 in DRAM (PP 3)

Memory Request Actions

1. Virtual address sent to the MMU
2. The MMU gets page table entry (PTE) address
   - requests PTE from the cache/main memory
3. The cache/main memory returns the PTE to the MMU.

Page Hit Actions

4. The MMU constructs the physical address
    - Sends it to the cache/main memory

5. Cache/main memory returns the requested data

Page Fault Actions

When (PTE valid=0):

4. The MMU triggers a page fault exception, transferring control to fault handler in OS
5. The fault handler identifies a victim page in physical memory (pages out to disk if needs write-back)
6. The fault handler pages in the new page and updates the PTE in memory
7. The fault handler returns to original process, restarting the faulting instruction—CPU resends VA, page hit

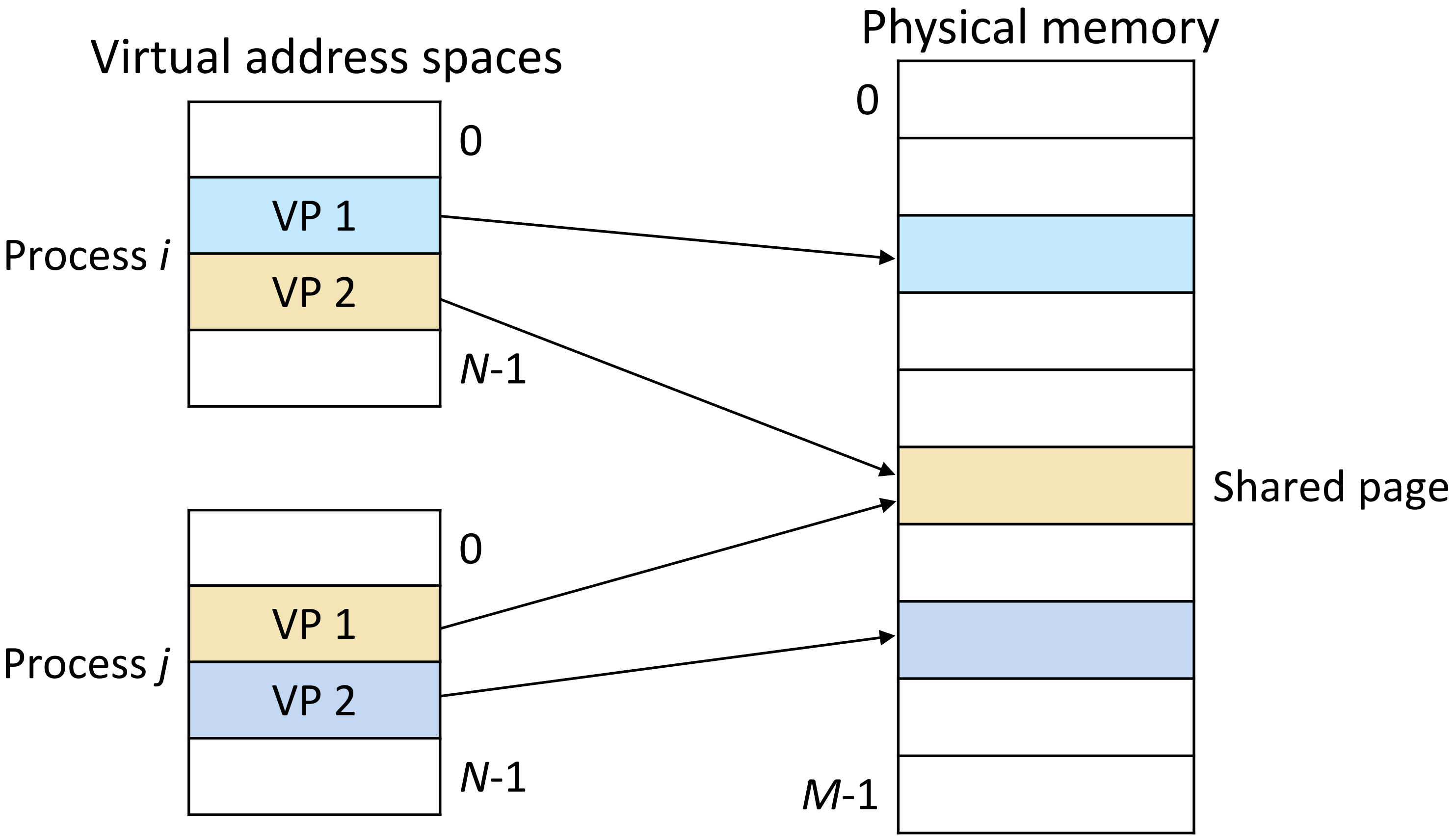
VM Simplifies . . .

- *Linking*—each process uses the same basic format for its memory image, regardless of where code and data actually reside in physical memory (allows uniformity)
  - `.text` always starts at virtual address `0x08048000`

VM Simplifies . . .

- *Sharing*—provides a consistent mechanism for processes to share code and/or data between processes
  - mapping appropriate VPs in different processes to the same PP

Memory Management



The OS provides a separate page table, and thus a separate virtual address space, for each process.

VM Simplifies . . .

- *Memory allocation*—simple mechanism for allocating additional memory to user processes (heap space).
  - allocate *k* contiguous VPs, no need for PPs to be contiguous

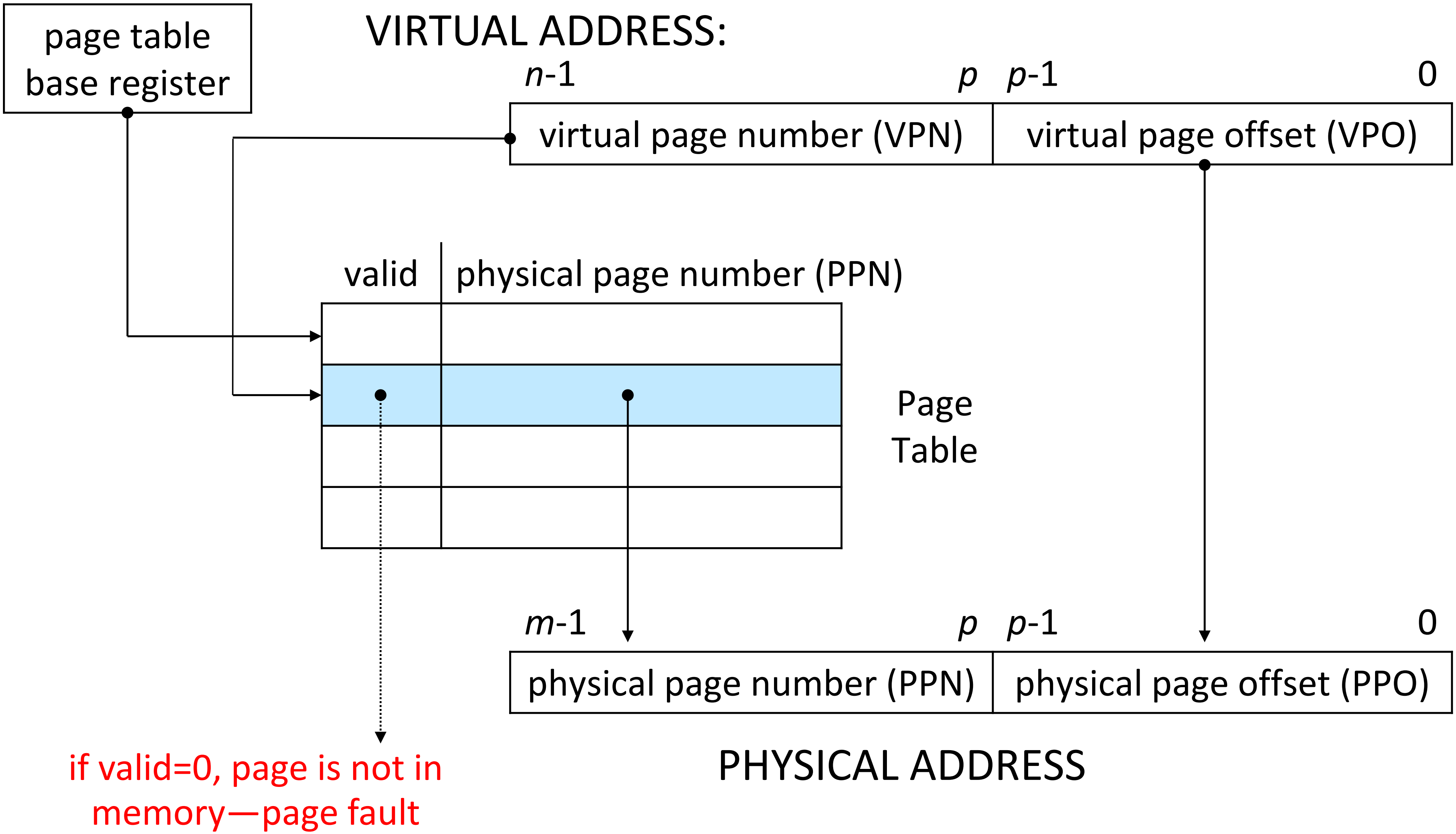
Memory Protection

- The OS controls access to the memory system
- A user process should be prevented from:
  - modifying its read-only `.text` section
  - reading or modifying any code/data in the kernel
  - reading or modifying the private memory of other processes
  - modifying any VPs shared with other processes (unless all parties explicitly allow it)

Memory Protection

- The OS controls access to the memory system
- Accomplished by adding permission bits to PTE
  - Indicate a process's read/modify access
  - if an instruction violates these permissions, CPU triggers a general protection fault (typically "segmentation fault")

Address Translation



page table base register

VIRTUAL ADDRESS:

n-1 *p* *p*-1 0

virtual page number (VPN)

virtual page offset (VPO)

valid

physical page number (PPN)

Page Table

m-1 *p* *p*-1 0

physical page number (PPN)

physical page offset (PPO)

PHYSICAL ADDRESS

if valid=0, page is not in memory—page fault

Exercise: Page Sizes

- Suppose 32-bit virtual addresses, 24-bit physical addresses.
- Page size P = 1 KB
  - – $\log_2 1024 = 10$, 10 virtual page offset (VPO) bits
  - – $2^{32} / 2^{10} = 2^{22}$, 22 virtual page number (VPN) bits
  - – $\log_2 1024 = 10$, 10 virtual page offset (PPO) bits
  - – $2^{24} / 2^{10} = 2^{14}$, 14 physical page number (PPN) bits

Exercise: Page Sizes

- Suppose 32-bit virtual addresses, 24-bit physical addresses.

- Page size *P* = 4 KB
  - – VPO bits?
  - – VPN bits?
  - – PPO bits?
  - – PPN bits?

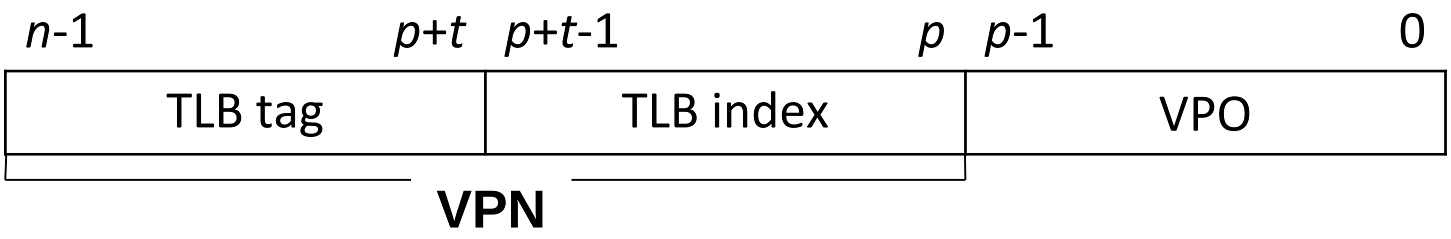
Translation Lookaside Buffer

- For every virtual address generated, the MMU must refer to a PTE to get the corresponding physical address
  - If the PTE is cached in L1, this costs a few cycles
  - If not, it costs tens to hundreds of cycles (L2 or main memory)

Translation Lookaside Buffer

- A ***translation lookaside buffer*** (TLB), small cache of PTEs in the MMU, can eliminate this cost.

- TLB is virtually addressed and each block holds a PTE
  - – typically has a high degree of associativity, T=2t sets



n-1 *p*+*t* *p*+*t*-1 *p* *p*-1 0

TLB tag | TLB index | VPO

VPN

TLB Hit Actions

1. CPU generates virtual address
2. MMU looks up PTE in the TLB
3. TLB hit:
    4. Fetch the PTE from the TLB
5. Construct physical address and send to cache/main memory
6. Cache/main memory returns the requested data

TLB Miss Actions

1. CPU generates virtual address

2. MMU looks up PTE in the TLB

3. TLB miss:

    4. PTE requested from the cache/main memory

    5. TLB is updated with new PTE (evicting another PTE from the TLB)

6. Construct physical address and send to cache/main memory

7. Cache/main memory returns the requested data