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.
- VM treats main memory (MM) as a cache for disk.
  - transfers data back and forth between disk and MM, as needed
- VM simplifies memory management.
  - provides each process with a uniform address space
- VM protects the address space of each process.
  - prevents some process from inadvertently writing to the memory used by another process
- VM 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
- 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
- When the CPU executes a load instruction, it generates a PA *x* and passes *x* to main memory (via memory bus).
- Main memory fetches the word starting at PA *x* and returns it to the CPU.
  - which then stores the data word in a register

Virtual Addressing

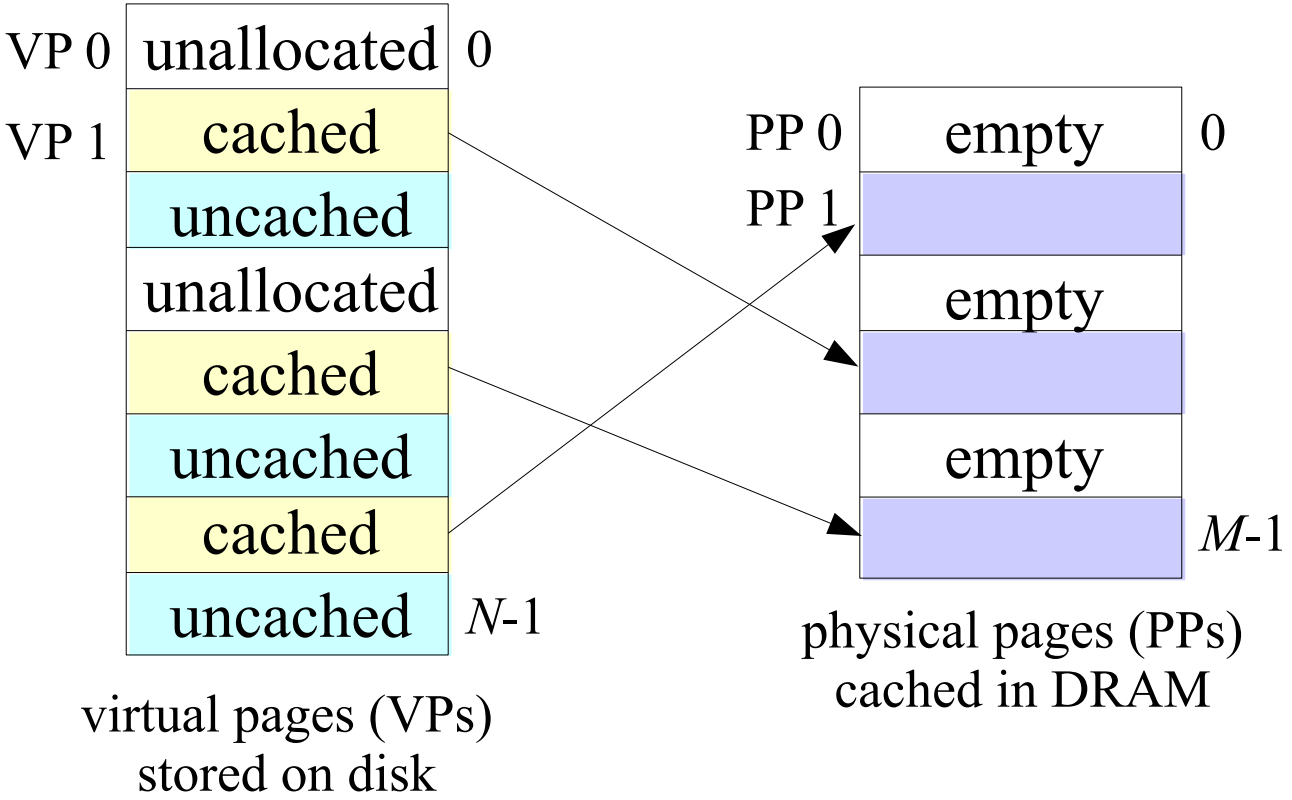
- Another form of addressing uses a *virtual address* (VA).
- *Virtual addressing* is using the VA to access memory.
  - used by modern processors for general-purpose computing
- CPU generates a VA *y*, which is converted to the appropriate PA before being passed to main memory.
- The translation of VA to PA requires close cooperation between the CPU hardware and the OS.
  - memory management unit (MMU)—dedicated CPU hardware for translating VAs, using look-up table stored in memory

Address Space

- *Address space*—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.
- Data objects are distinguished from their addresses.
  - each byte of main memory has a VA and a PA

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? PPs?



DRAM Cache Organization

- Misses in DRAM caches are much more expensive than those in SRAM caches.
  - DRAM is ~10 times slower than SRAM
  - disk is ~100K times slower than DRAM
  - cost of reading the first byte from a disk sector is ~100K times slower than reading successive bytes in the sector
- The organization of DRAM caches is driven by *the large miss penalty*
  - DRAM caches are fully associative

  and *the expense of accessing the first disk sector byte*
  - virtual pages are large (4-8 KB)

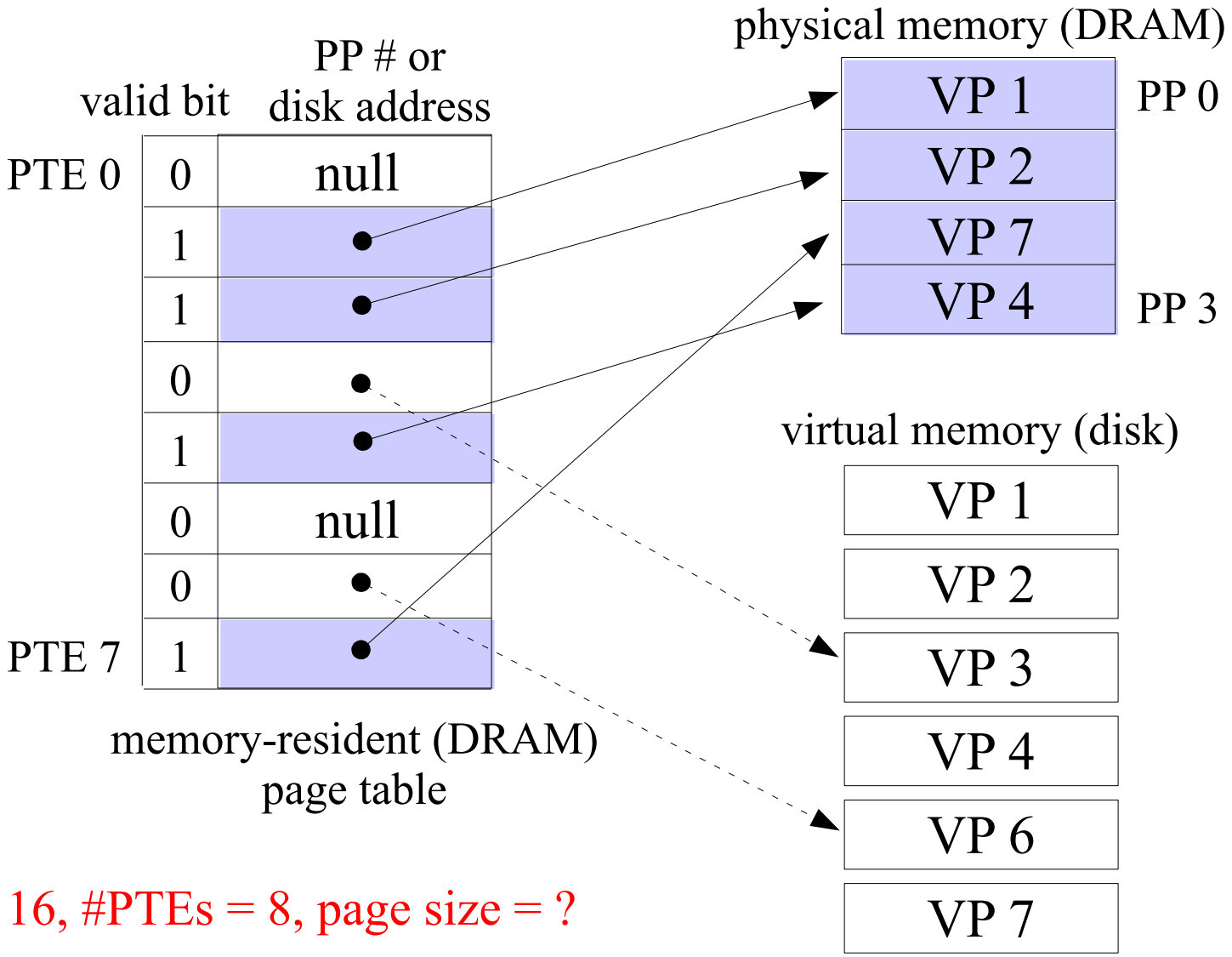
Page Table

- 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?
- The *page table*, a data structure stored in physical memory, 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.
- 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, the address field indicates the start of the corresponding PP in DRAM
  - if valid bit = 0, a null address indicates an unallocated VP and a non-null address points to the start of the VP on disk

Example: Page Table



$n = 16$, #PTEs = 8, page size = ?

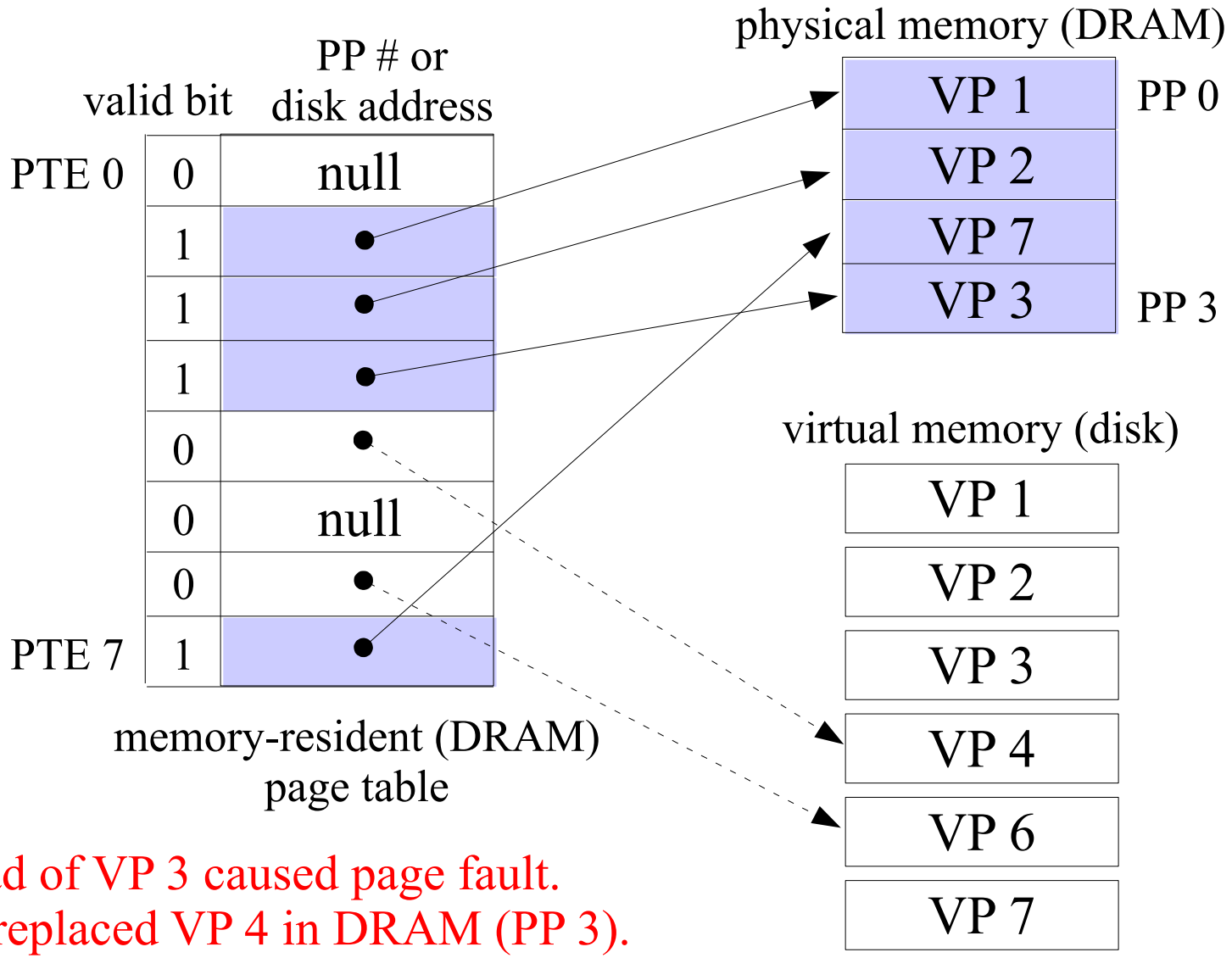
Page Hit

- When the CPU reads a word of virtual memory contained in VP 2 (cached in DRAM), a *page hit* occurs.
- The memory management unit (MMU) is dedicated hardware on the CPU chip that uses the virtual address to locate PTE 2 and read it from memory.
- Because the valid bit of PTE 2 is set, the MMU knows that VP 2 is cached.
- Then the MMU gets the starting address of the cached page in PP 0 from the address field of PTE 2.

Page Fault

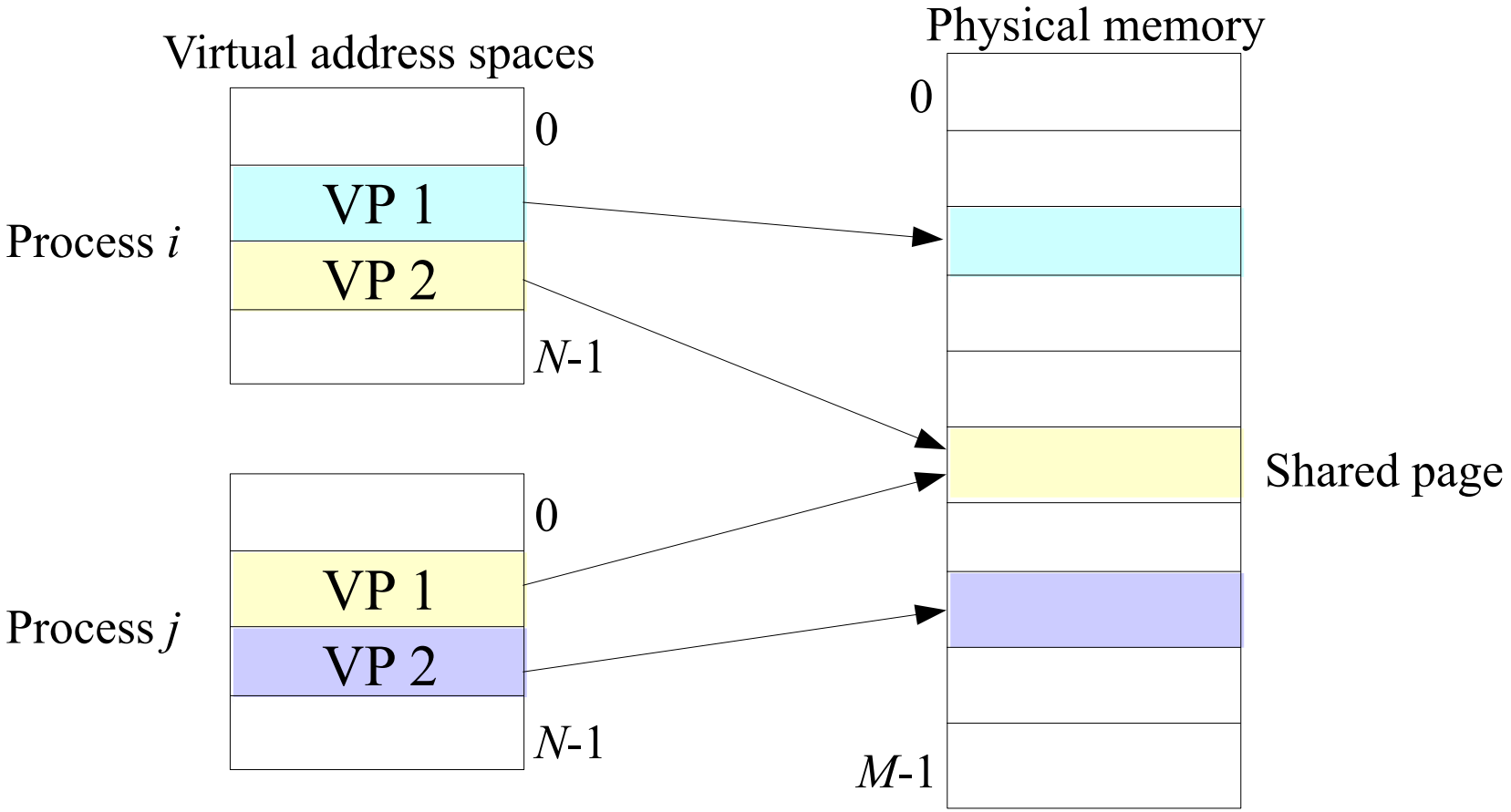
- *Page fault*—a DRAM cache miss.
- When the CPU reads a word of virtual memory contained in VP 3 (uncached), a page fault occurs.
- MMU reads PTE 3 from memory (valid bit indicates that VP 3 is uncached) and triggers a page fault exception.
- The page fault exception handler (in the kernel) selects a victim page (say, VP 4 in PP 3).
- The kernel modifies PTE 4 and PTE 3. How?
- At handler return, faulting instruction restarts—page hit.

Example: Page Fault



Read of VP 3 caused page fault.
VP 3 replaced VP 4 in DRAM (PP 3).

Memory Management



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

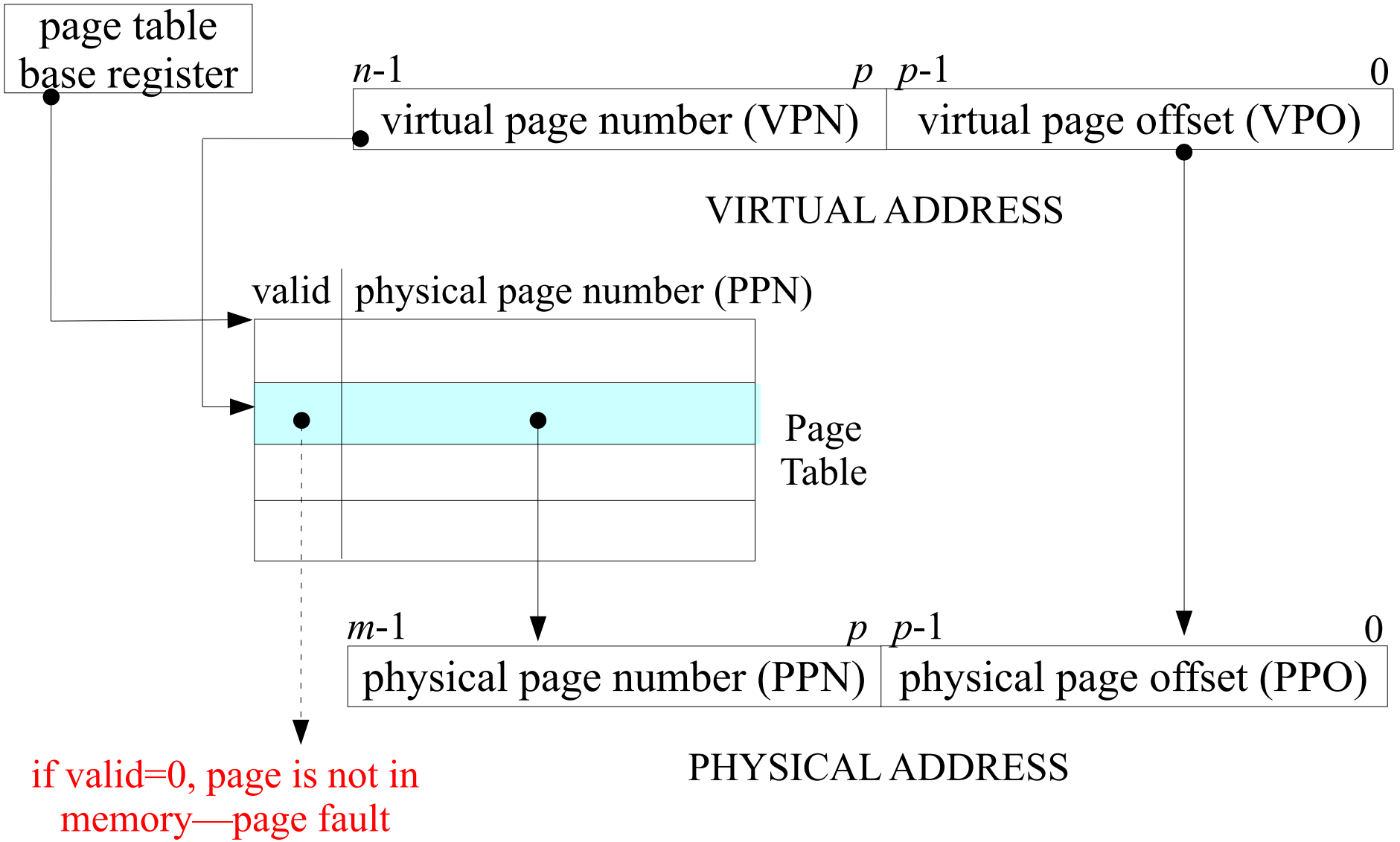
VM Simplifies . . .

- 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`
- Sharing—provides a consistent mechanism for processes to share code and/or data.
  - mapping appropriate VPs in different processes to the same PP
- 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 must control 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)
- This can be accomplished by adding permission bits to the PTE the 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 Hit Actions

1. The processor generates a virtual address and sends it to the memory management unit (MMU).
2. The MMU generates the page table entry (PTE) address and requests it from the cache/main memory.
3. The cache/main memory returns the PTE to the MMU.
4. The MMU constructs the physical address and sends it to the cache/main memory.
5. The cache/main memory returns the requested data word to the processor.

Page Fault Actions

1-3. Same as for a page hit. (PTE returned to MMU)

4. (PTE valid=0) 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.

Exercise: Page Sizes

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

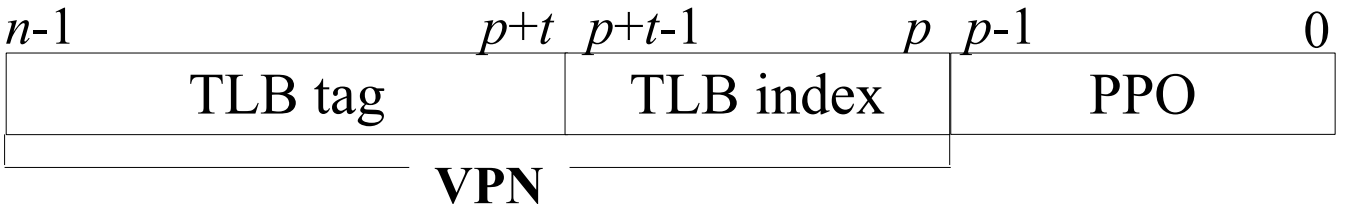
- page size $P = 1$ KB
  - $2^{32} / 2^{10} = 2^{22}$, 22 virtual page number (VPN) bits
  - 10 virtual page offset (VPO) bits
  - $2^{24} / 2^{10} = 2^{14}$, 14 physical page number (PPN) bits
  - 10 virtual page offset (PPO) bits
- What page size P will give 20 VPN bits and 12 PPN bits?

  CLICK:

  A. 2 KB
  C. 8 KB
  B. 4 KB
  D. none of the above

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).
- 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, $T=2^t$ sets



TLB Hit Actions

1. CPU generates virtual address.
2. The MMU looks up the appropriate PTE in the TLB, using the VPN bits.
3. (TLB hit) The MMU fetches the PTE from the TLB.
4. The MMU constructs the physical address and sends it to the cache/main memory.
5. The cache/main memory returns the requested data word to the processor.

TLB Miss Actions

1-2. Same as for a TLB hit. (TLB look-up)

3. (TLB miss) MMU generates the PTE address and requests it from the cache/main memory.
4. The TLB is updated with this PTE, evicting another PTE from the TLB.
5. The MMU constructs the physical address and sends it to the cache/main memory.
6. The cache/main memory returns the requested data word to the processor.

Exercise: Address Translation

Assume: memory accesses to are to 1-byte words
n = 14 (virtual), m = 12 (physical), P = 64 (page size)
TLB is 4-way set associative with 16 total entries

- Number of VPs?

  PPs?

  PTEs?

- Which virtual address bits 13-0 are the VPN?

  the VPO?

- Which physical address bits 11-0 are the PPN?

  the PPO?

- Which bits of the VPN are the TLB set index?

  the TLB tag?

Exercise: Address Translation

TLB: 4-way, 16 entries, 4 sets

set	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*
0	03	--	0	09	0D	1	00	--	0	07	02	1
1	03	2D	1	02	--	0	04	--	0	0A	--	0
2	02	--	0	08	--	0	06	--	0	03	--	0
3	07	--	0	03	0D	1	0A	34	1	02	--	0

- What happens when a load reads the byte at VA `0x03d4`?
- What is the VPN? the VPO?
- Does the TLB have a cached copy of the PTE?
- What is the PPN? the PPO?

Page Table (first 16 entries)

VPN	*PPN*	*valid*	*VPN*	*PPN*	*valid*
00	28	1	08	13	1
01	--	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	--	0
04	--	0	0C	--	0
05	16	1	0D	2D	1
06	--	0	0E	11	1
07	--	0	0F	0D	1

Exercise: Address Translation

- PPN `0x0D` and PPO `0x14` form 12-bit physical address `0x354`.
- Which bits of the PA are the
  cache block offset?
  the cache set index?
  the cache tag?
- L1 cache hit or miss?

L1 cache: DM, 4-byte block, 16 sets

set	*tag*	*valid*	*blk0*	*blk1*	*blk2*	*blk3*
0	19	1	99	11	23	11
1	15	0	--	--	--	--
2	1B	1	00	02	04	08
3	36	0	--	--	--	--
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	--	--	--	--
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	--	--	--	--
10	2D	1	93	15	DA	3B
11	0B	0	--	--	--	--
12	12	0	--	--	--	--
13	16	1	04	96	34	15
14	13	1	83	77	1B	D3
15	14	0	--	--	--	--

TLB: 4-way, 16 entries, 4 sets

set	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*	*tag*	*PPN*	*valid*
0	03	--	0	09	0D	1	00	--	0	07	02	1
1	03	2D	1	02	--	0	04	--	0	0A	--	0
2	02	--	0	08	--	0	06	--	0	03	--	0
3	07	--	0	03	0D	1	0A	34	1	02	--	0

Page Table (first 16 entries)

VPN	*PPN*	*valid*	*VPN*	*PPN*	*valid*
00	28	1	08	13	1
01	--	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	--	0
04	--	0	0C	--	0
05	16	1	0D	2D	1
06	--	0	0E	11	1
07	--	0	0F	0D	1

L1 cache: DM, 4-byte block, 16 sets

set	*tag*	*valid*	*blk0*	*blk1*	*blk2*	*blk3*
0	19	1	99	11	23	11
1	15	0	--	--	--	--
2	1B	1	00	02	04	08
3	36	0	--	--	--	--
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	--	--	--	--
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	--	--	--	--
10	2D	1	93	15	DA	3B
11	0B	0	--	--	--	--
12	12	0	--	--	--	--
13	16	1	04	96	34	15
14	13	1	83	77	1B	D3
15	14	0	--	--	--	--

What happens when a load reads 14-bit virtual address `0x026A`?

CLICK (for TLB, page, cache):

A. miss B. hit