

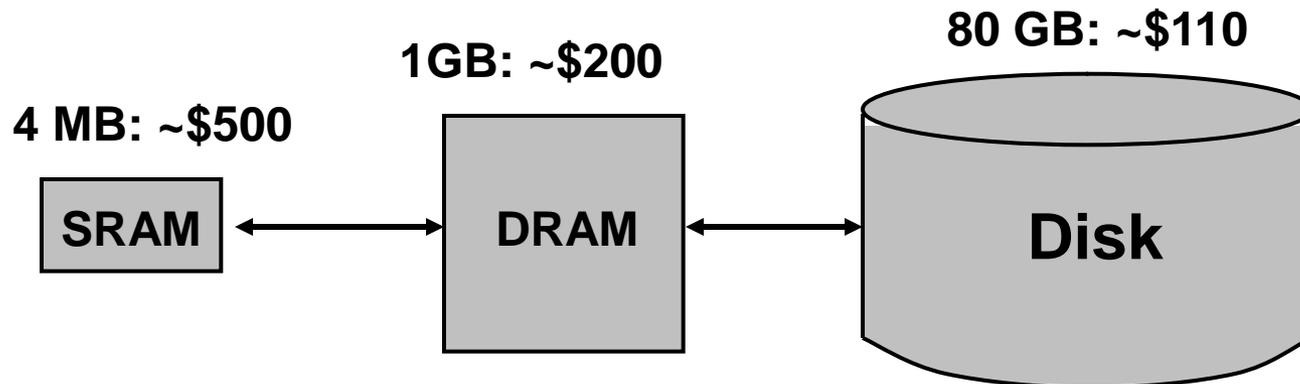
Virtual Memory

Motivations for Virtual Memory

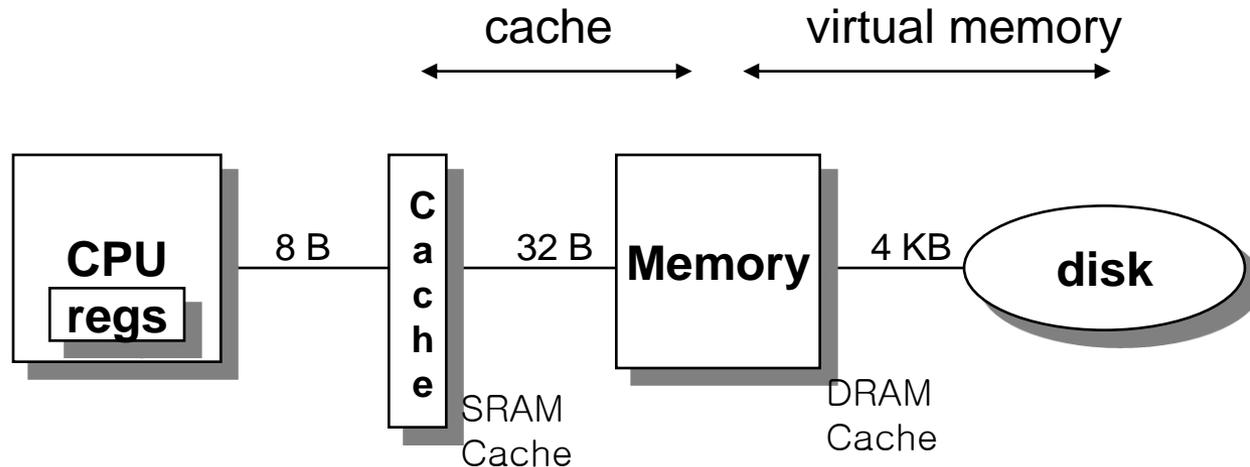
- Use Physical DRAM as a Cache for the Disk
 - Address space of a process can exceed physical memory size
 - Sum of address spaces of multiple processes can exceed physical memory
- Simplify Memory Management
 - Multiple processes resident in main memory.
 - Each process with its own address space
 - Only “active” code and data is actually in memory
 - Allocate more memory to process as needed.
- Provide Protection
 - One process can't interfere with another.
 - because they operate in different address spaces.
 - User process cannot access privileged information
 - different sections of address spaces have different permissions.

Motivation #1: DRAM a “Cache” for Disk

- Full address space is quite large:
 - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
 - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- Disk storage is ~300X cheaper than DRAM storage
 - 80 GB of DRAM: ~ \$33,000
 - 80 GB of disk: ~ \$110
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
\$/Mbyte:		\$125/MB	\$0.20/MB	\$0.001/MB
line size:	8 B	32 B	4 KB	

Annotations:

- A dotted arrow points from Cache to Memory with the label **10X slower**.
- A dotted arrow points from Memory to Disk with the label **•100,000X slower**.
- Below the Memory/Disk comparison, it states: **•First byte is ~100,000X slower than successive bytes on disk**.

DRAM/disk is the extreme of SRAM/DRAM relation

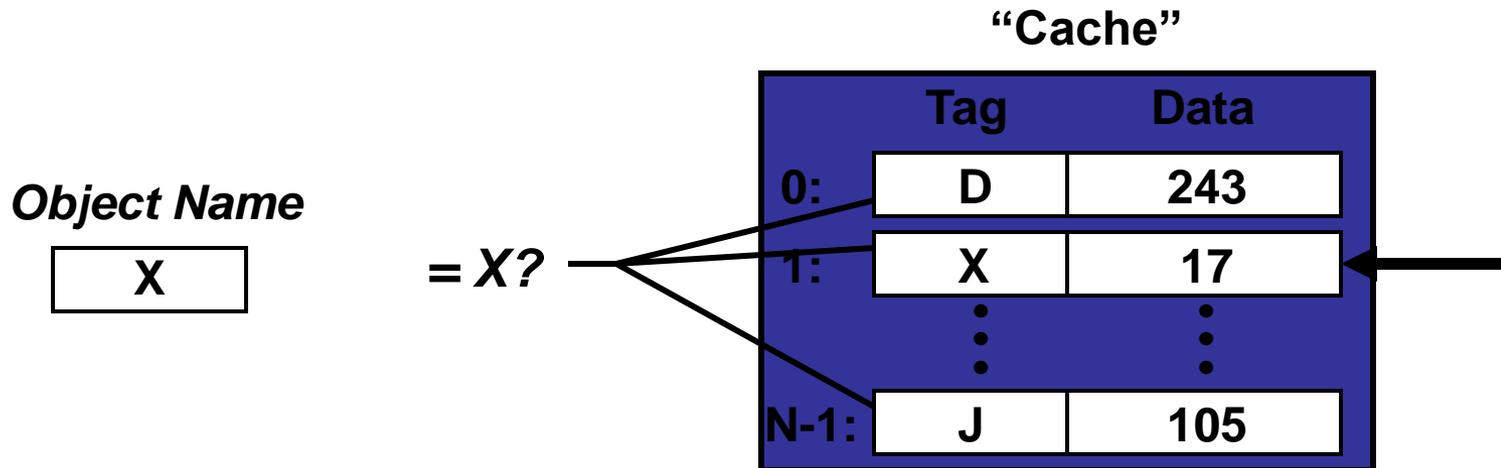
larger, slower, cheaper →

Caching Policy Decision

- DRAM vs. disk is more extreme than SRAM vs. DRAM
- Bottom line:
 - Design decisions made for DRAM caches driven by enormous cost of misses
- DRAM caching policy?
 - Line size?
 - Large, since disk better at transferring large blocks
 - Associativity?
 - High, to minimize miss rate
 - Write through or write back?
 - Write back, since can't afford to perform small writes to disk

Locating an Object in a “Cache”

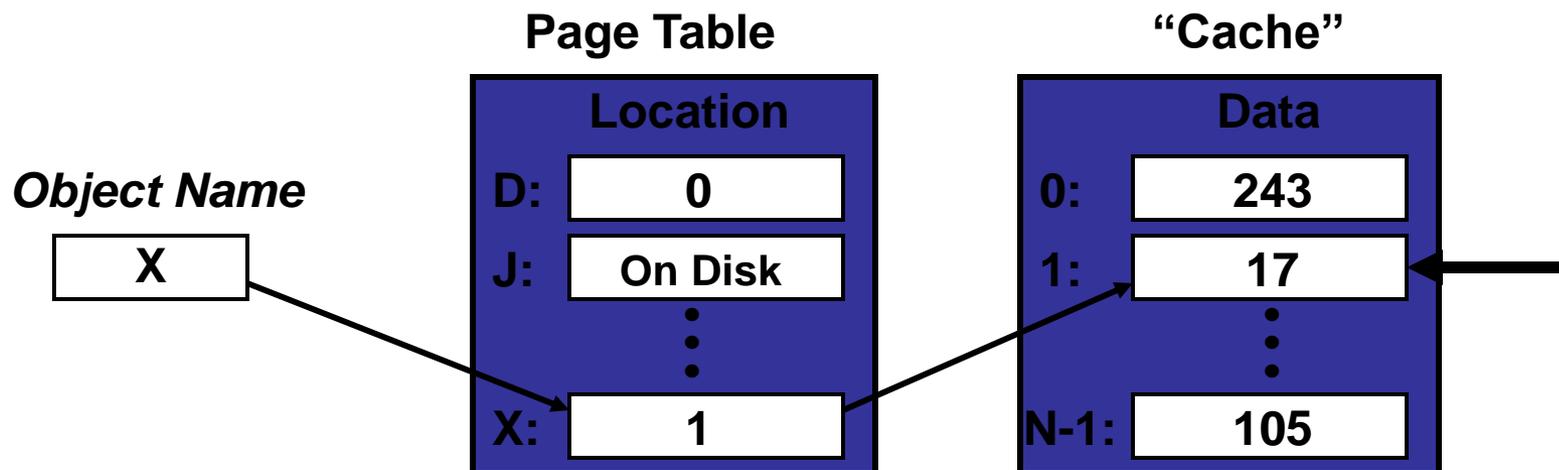
- SRAM Cache
 - Tag stored with cache line
 - No tag for block not in cache
 - Hardware retrieves information
 - can quickly match against multiple tags



Locating an Object in “Cache” (cont.)

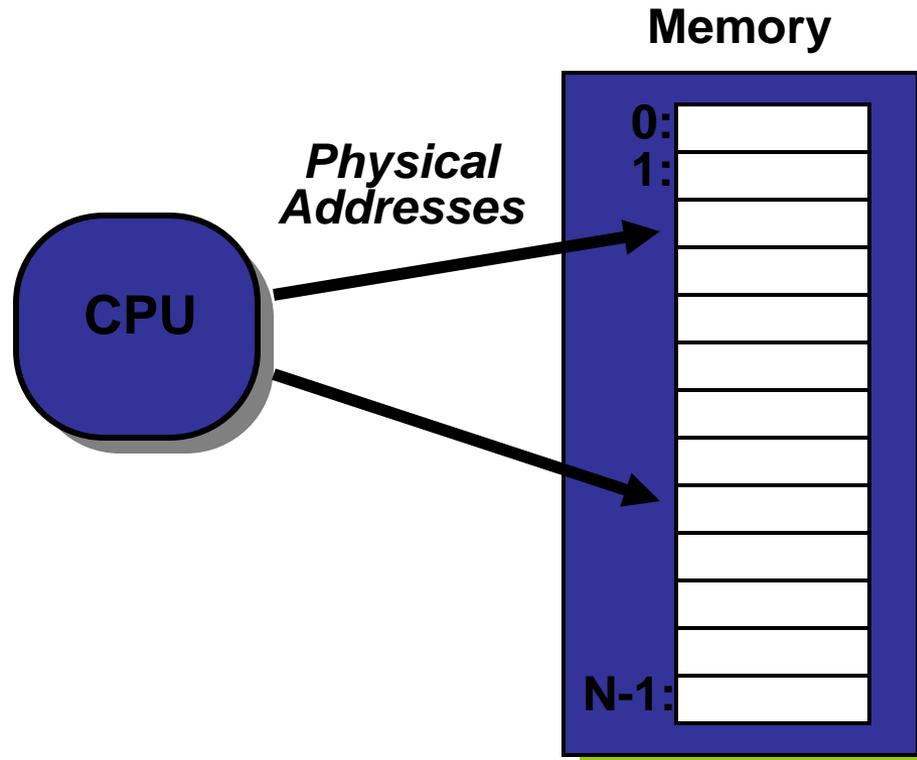
■ DRAM Cache

- Each allocated page of virtual memory has entry in *page table*
- Page table entry even if page not in memory
 - Specifies disk address
 - Only way to indicate where to find page
- HW/OS retrieves information



A System with Physical Memory Only

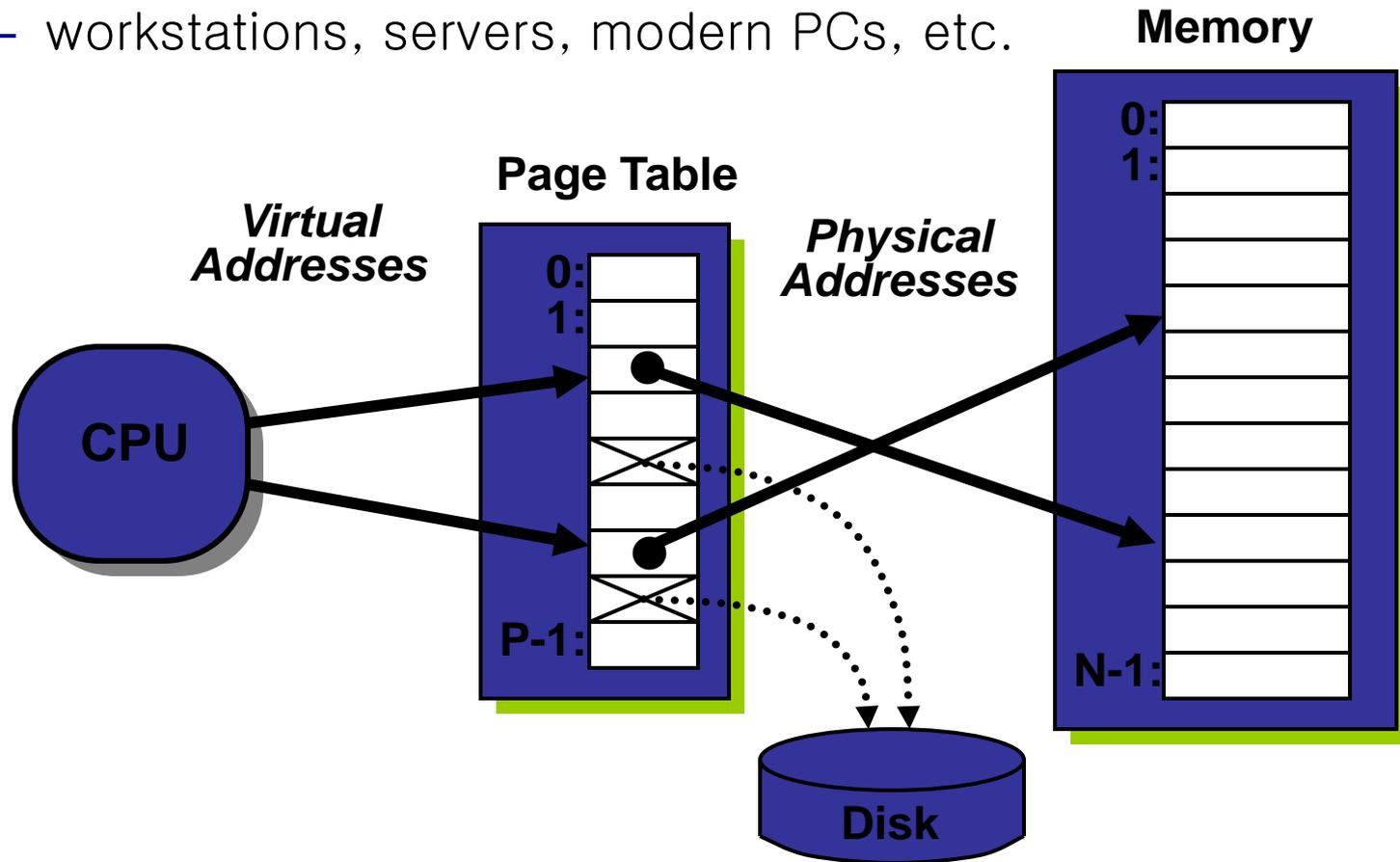
- Examples:
 - most Cray machines, early PCs, nearly all embedded systems, etc.



- Addresses generated by the CPU correspond directly to bytes in physical memory

A System with Virtual Memory

- Examples:
 - workstations, servers, modern PCs, etc.

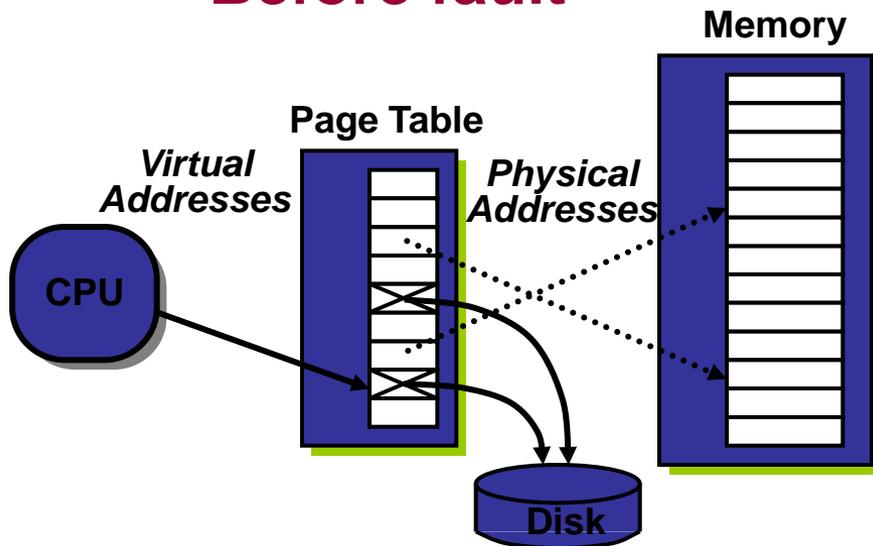


- **Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)**

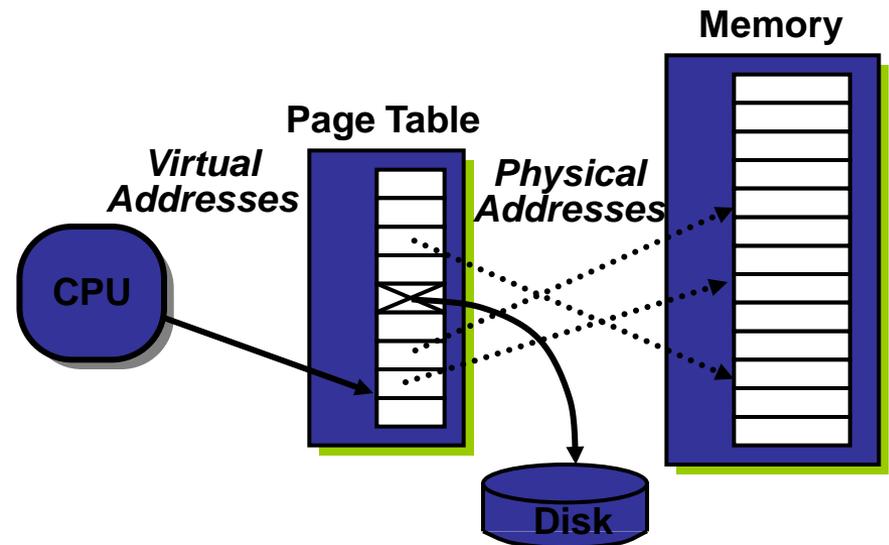
Page Faults (like “Cache Misses”)

- What if an object is on disk rather than in memory?
 - Page table entry indicates virtual address not in memory
 - OS exception handler invoked to move data from disk into memory
 - current process suspends, others can resume
 - OS has full control over placement, etc.

Before fault

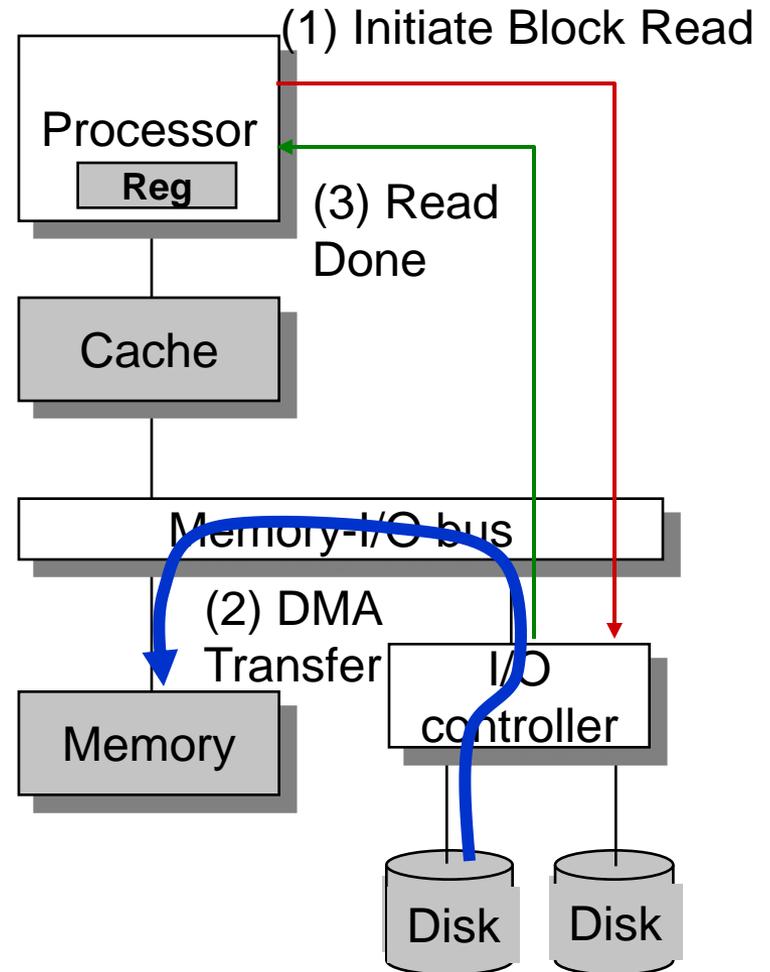


After fault



Servicing a Page Fault

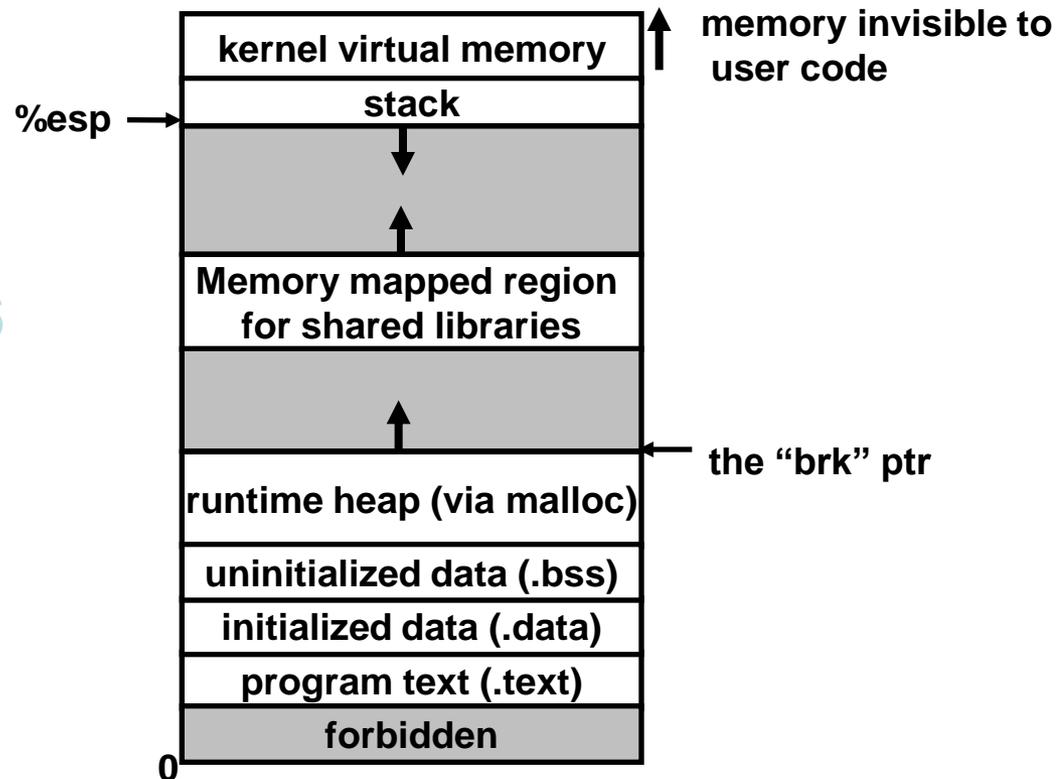
- Processor Signals Controller
 - Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
 - Direct Memory Access (DMA)
 - Under control of I/O controller
- I / O Controller Signals Completion
 - Interrupt processor
 - OS resumes suspended process



Motivation #2: Memory Management

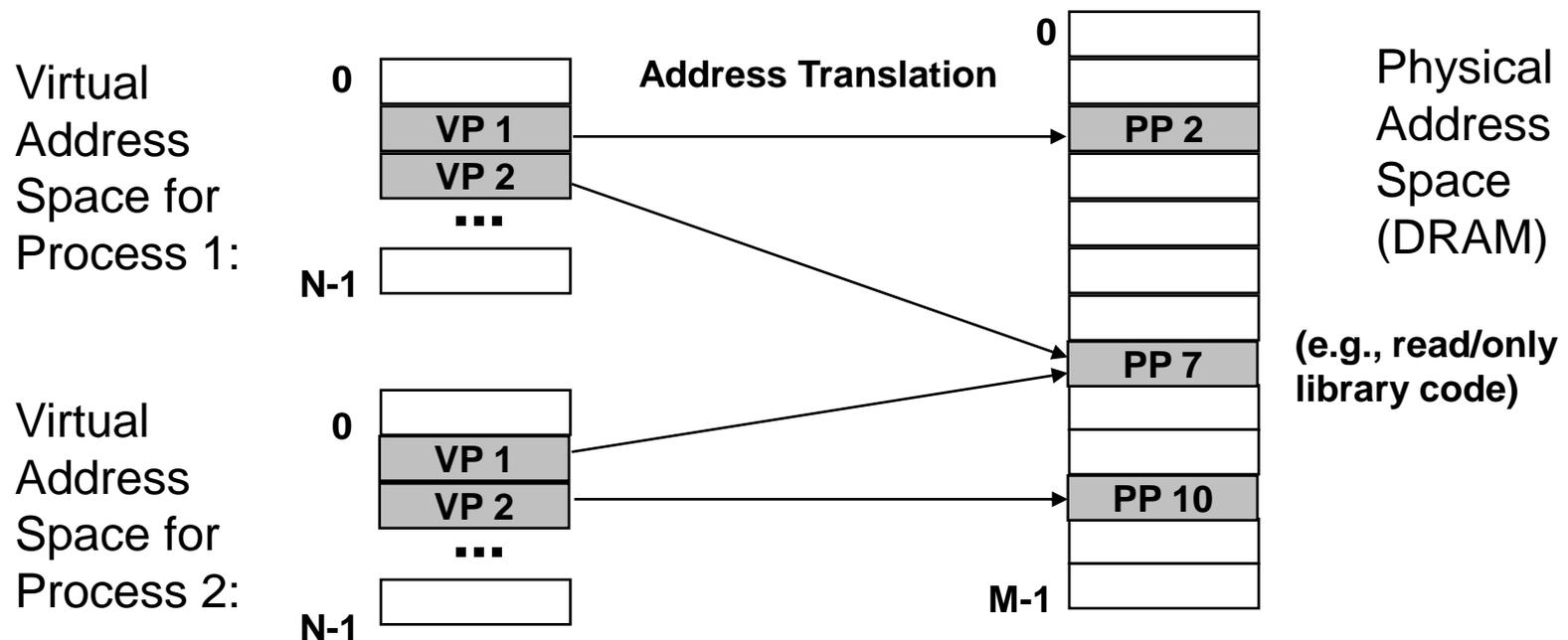
- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
 - what if two processes access something at the same address?

Linux/x86
process
memory
image



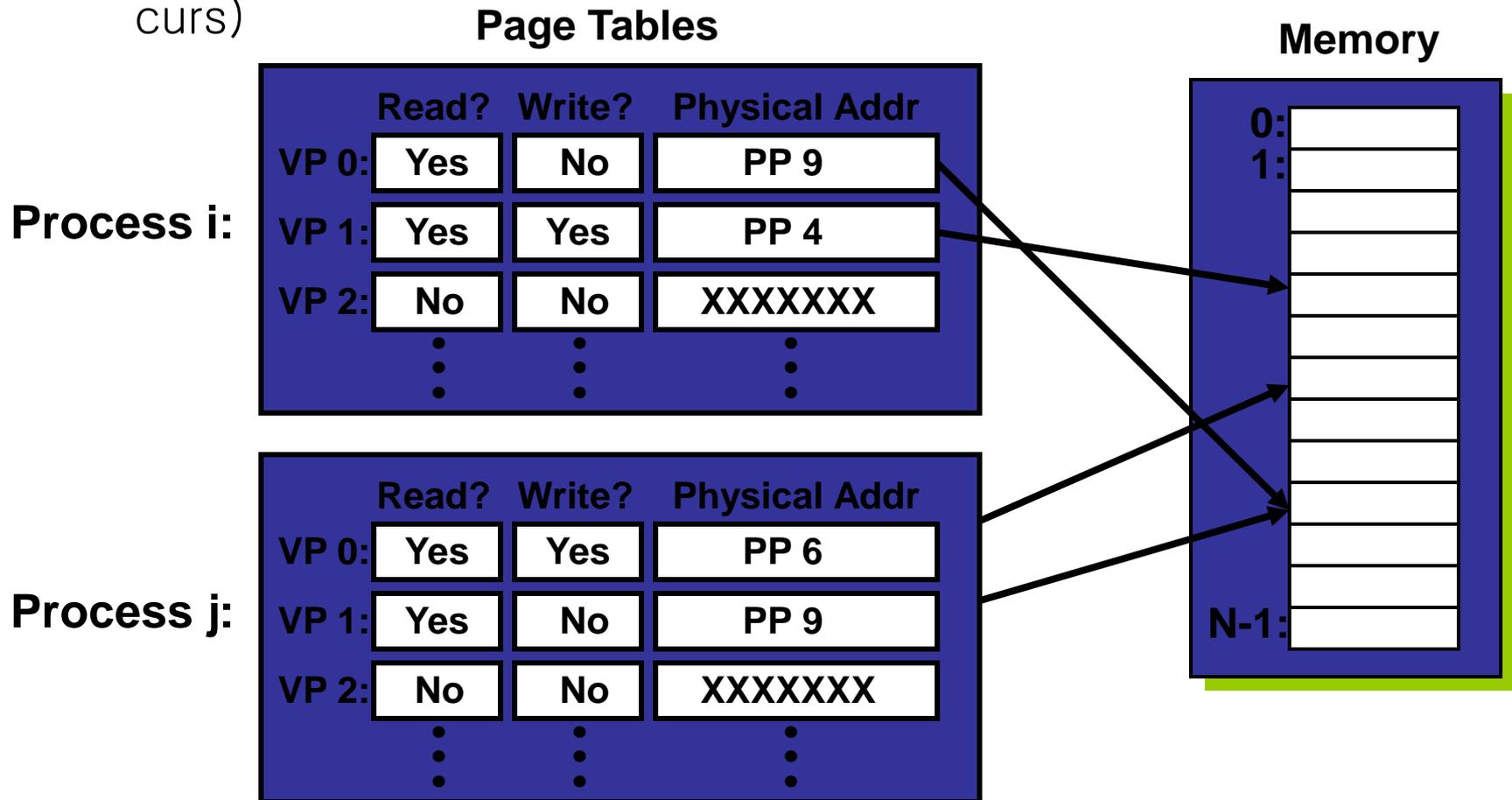
Solution: Separate Virt. Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
 - blocks are called “pages” (both virtual and physical)
- Each process has its own virtual address space
 - operating system controls how virtual pages as assigned to physical memory



Motivation #3: Protection

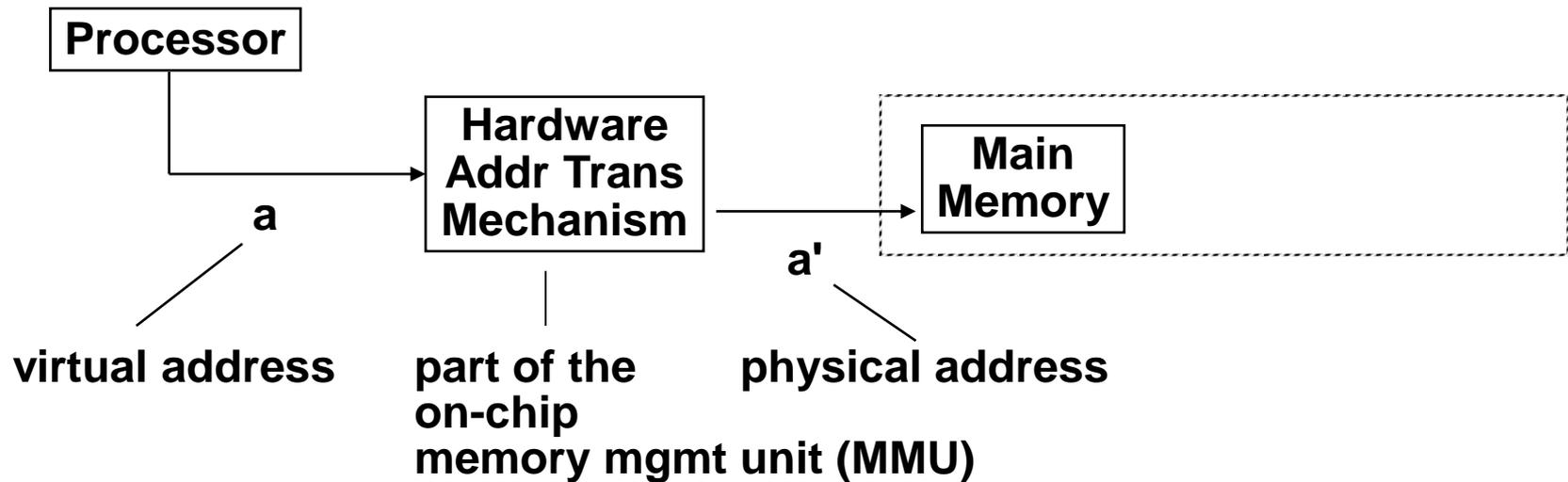
- Page table entry contains access rights information
 - hardware enforces this protection (trap into OS if violation occurs)



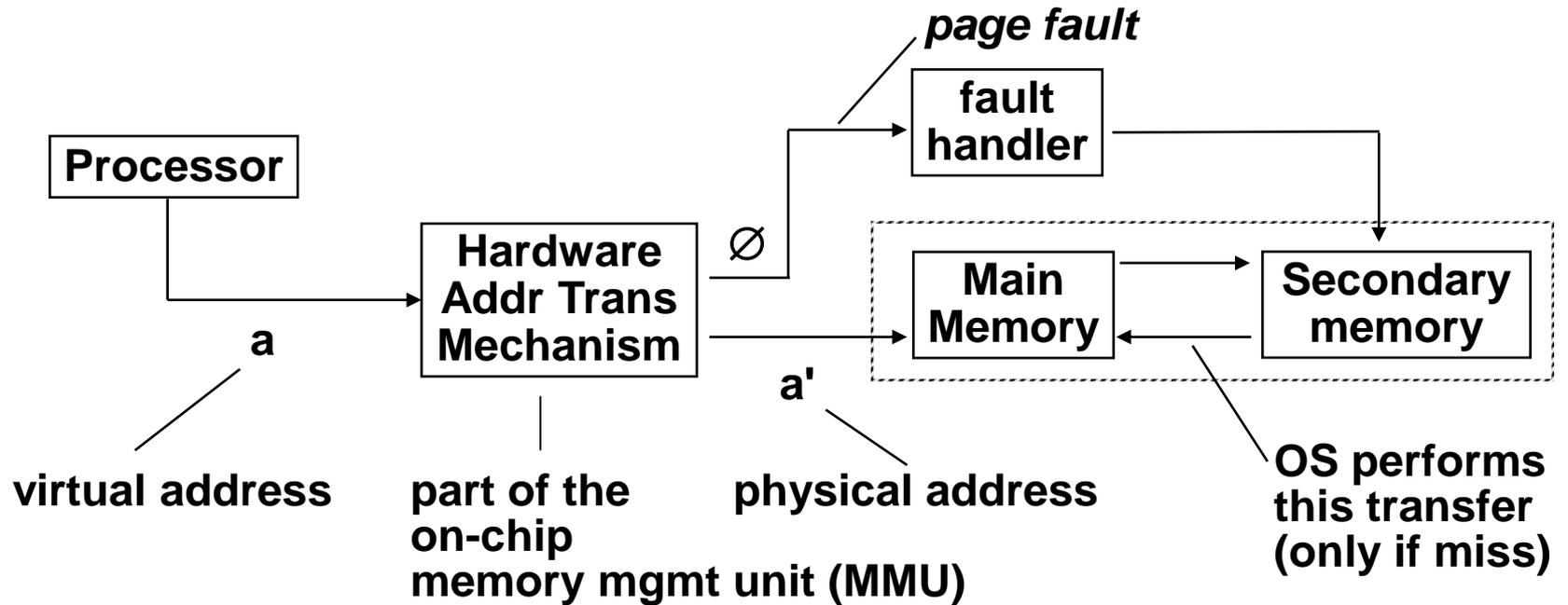
VM Address Translation

- Virtual Address Space
 - $V = \{0, 1, \dots, N-1\}$
- Physical Address Space
 - $P = \{0, 1, \dots, M-1\}$
 - $M < N$
- Address Translation
 - $\text{MAP}: V \rightarrow P \cup \{\emptyset\}$
 - For virtual address a :
 - $\text{MAP}(a) = a'$ if data at virtual address a at physical address a' in P
 - $\text{MAP}(a) = \emptyset$ if data at virtual address a not in physical memory
 - Either invalid or stored on disk

VM Address Translation: Hit



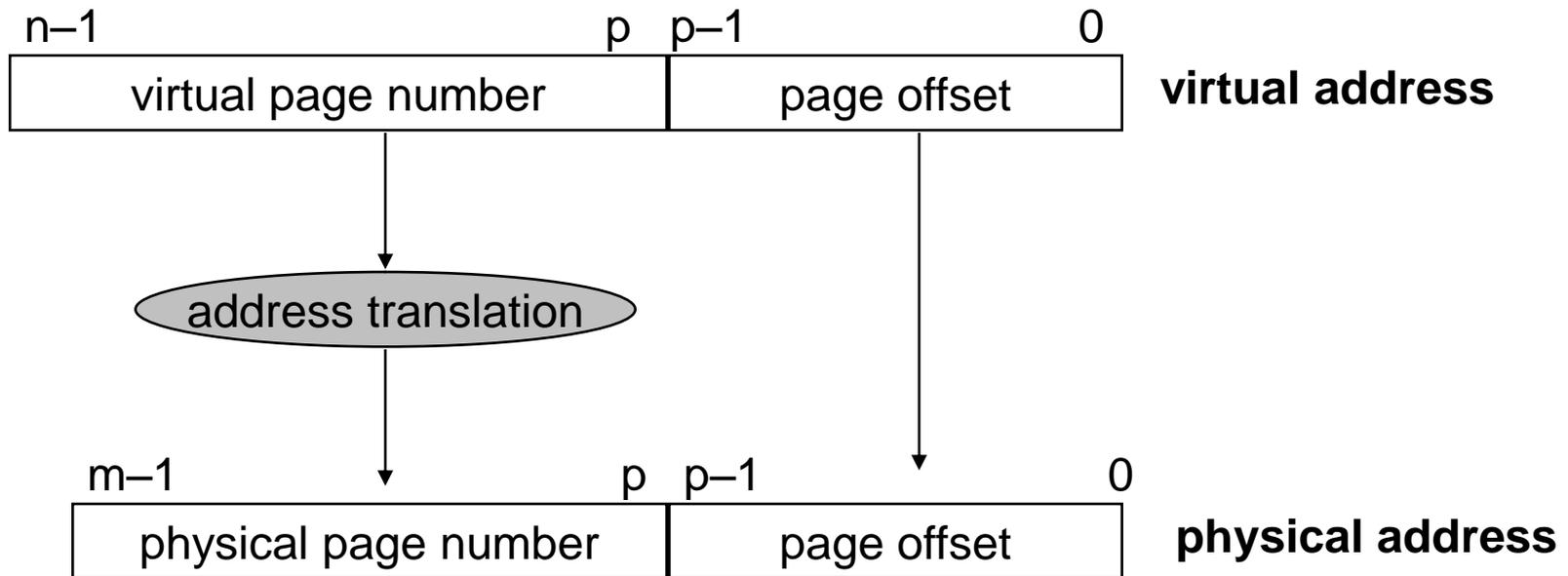
VM Address Translation: Miss



VM Address Translation

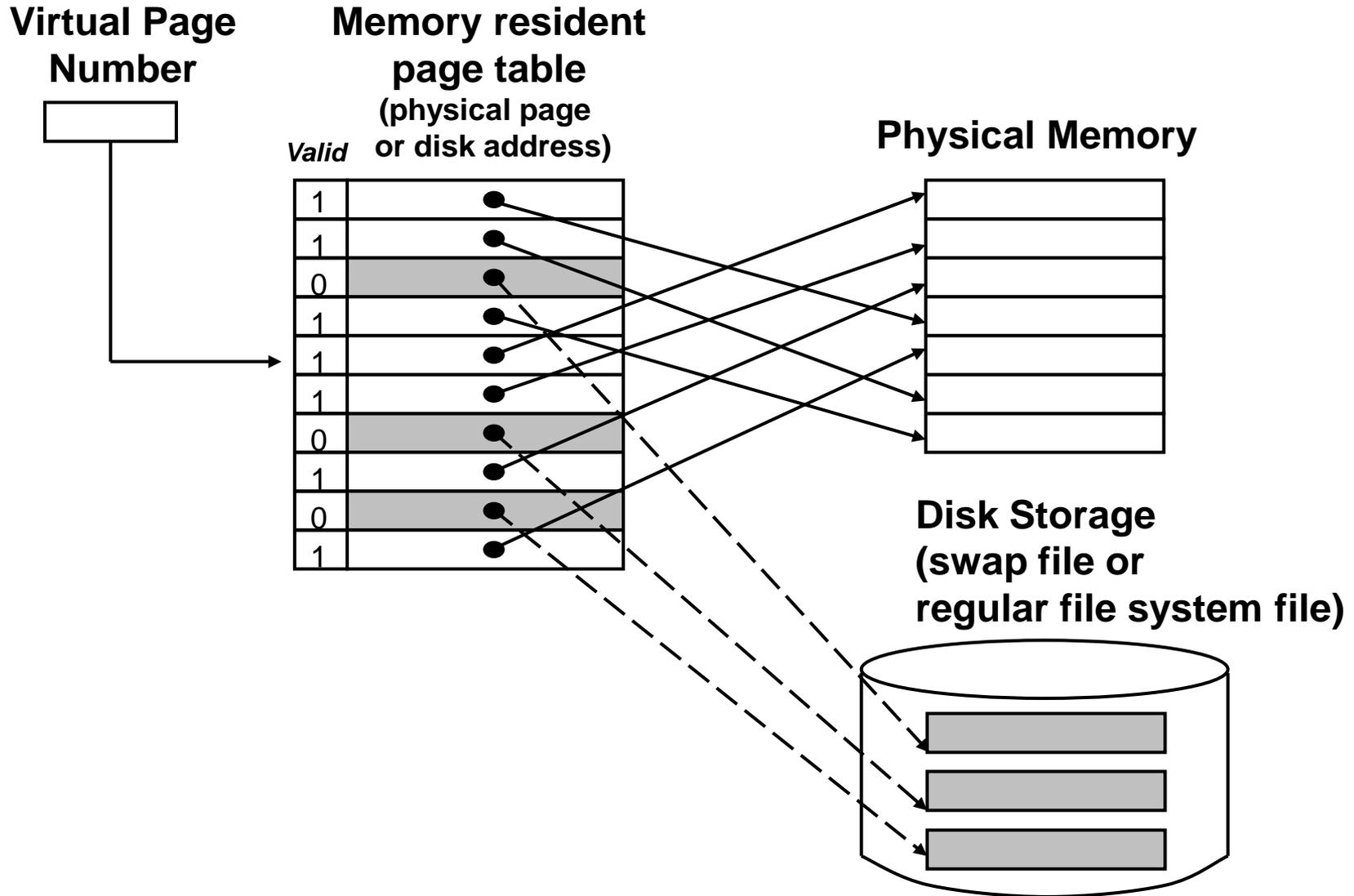
■ Parameters

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

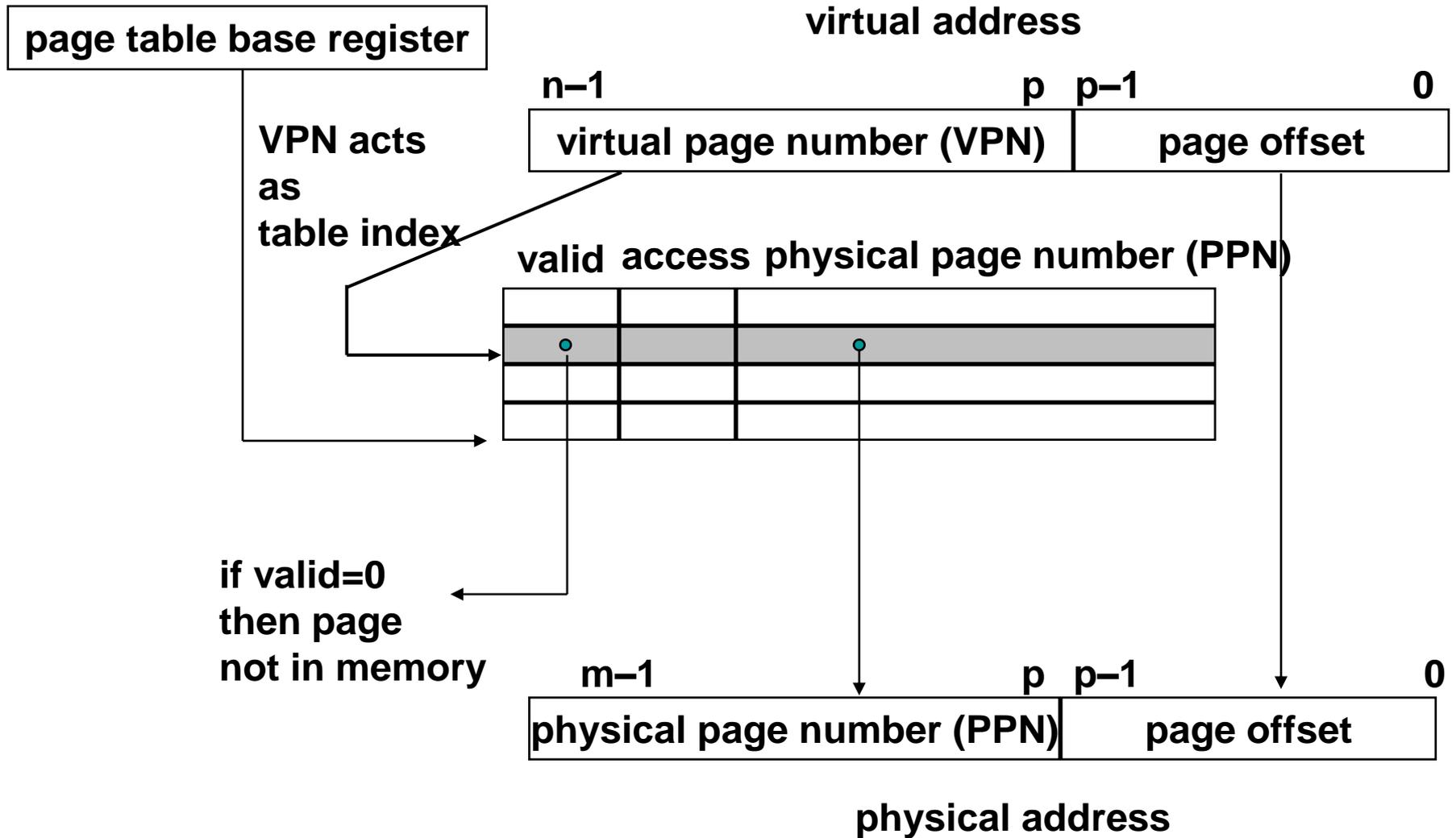


Page offset bits don't change as a result of translation

Page Tables



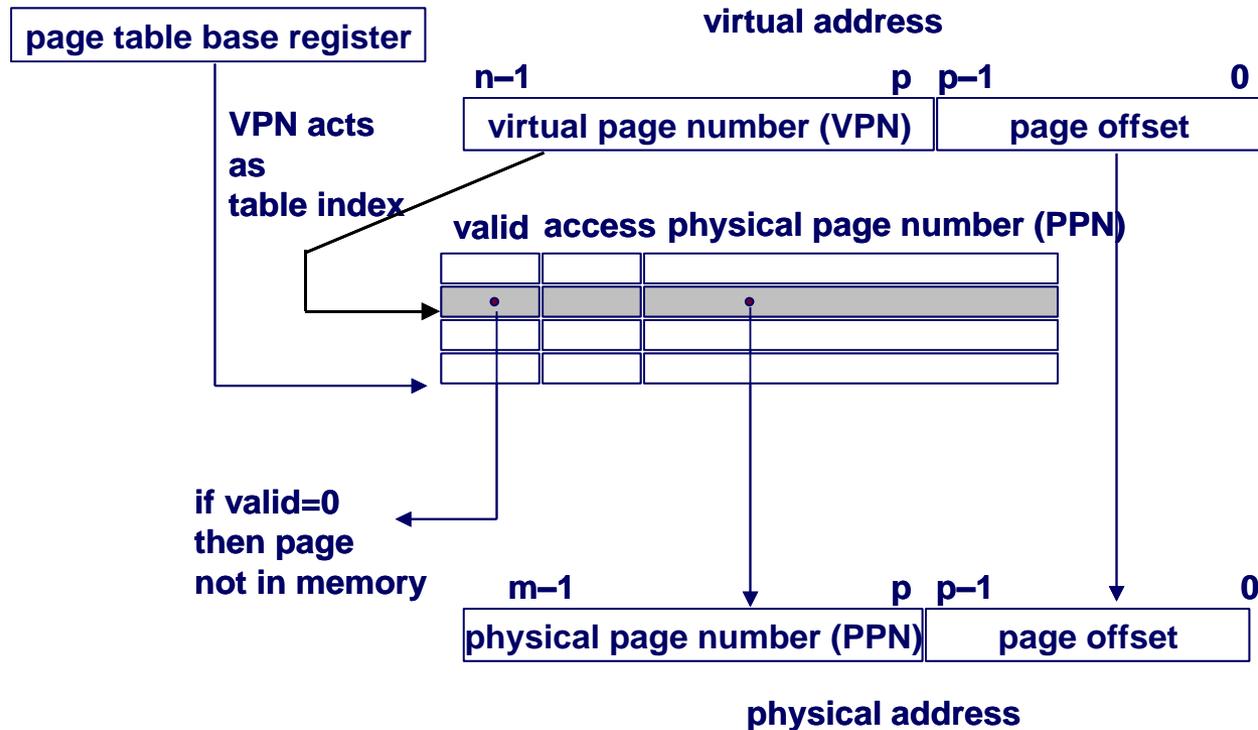
Address Translation via Page Table



Page Table Operation

■ Translation

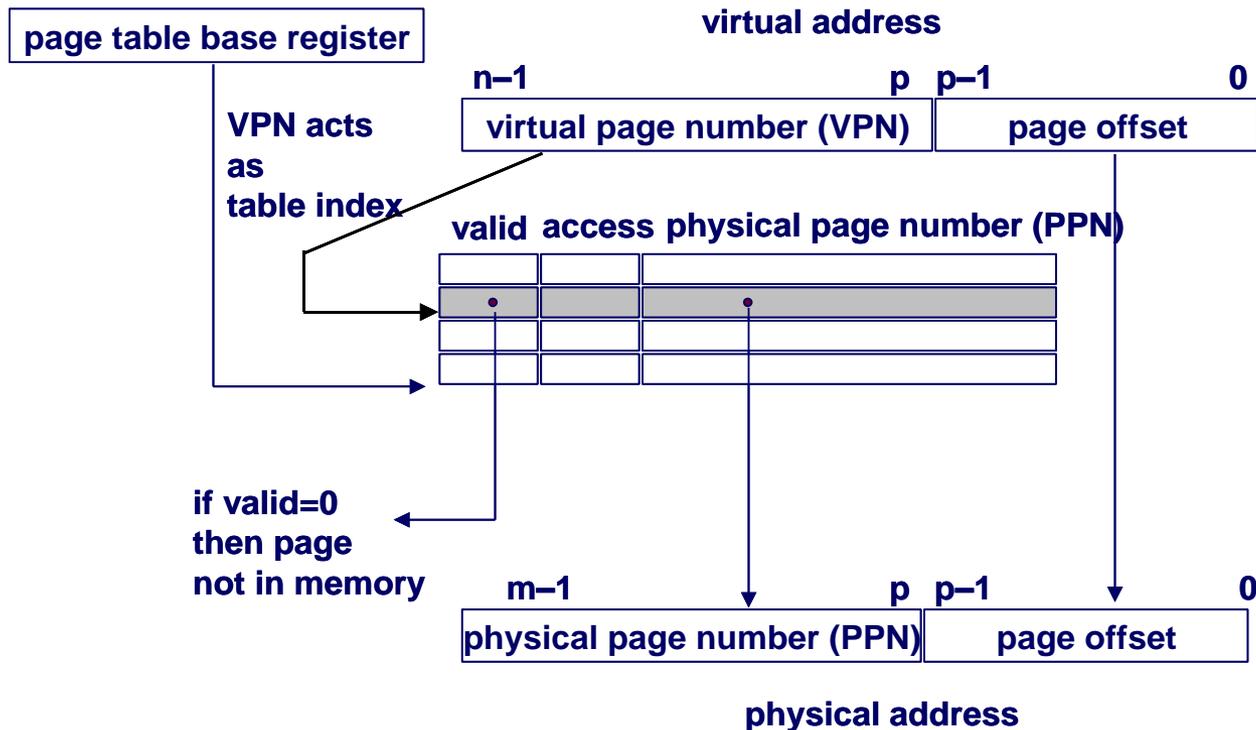
- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)



Page Table Operation

■ Computing Physical Address

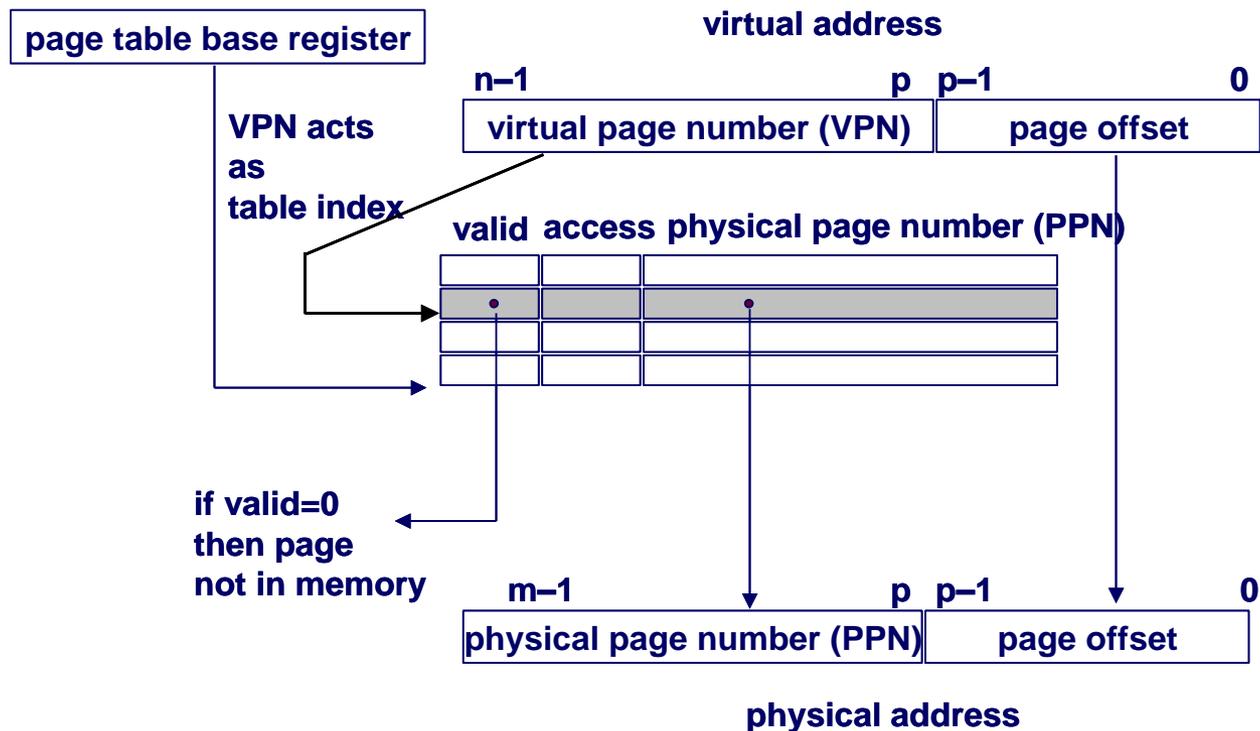
- Page Table Entry (PTE) provides information about page
 - if (valid bit = 1) then the page is in memory.
 - Use physical page number (PPN) to construct address
 - if (valid bit = 0) then the page is on disk
 - Page fault



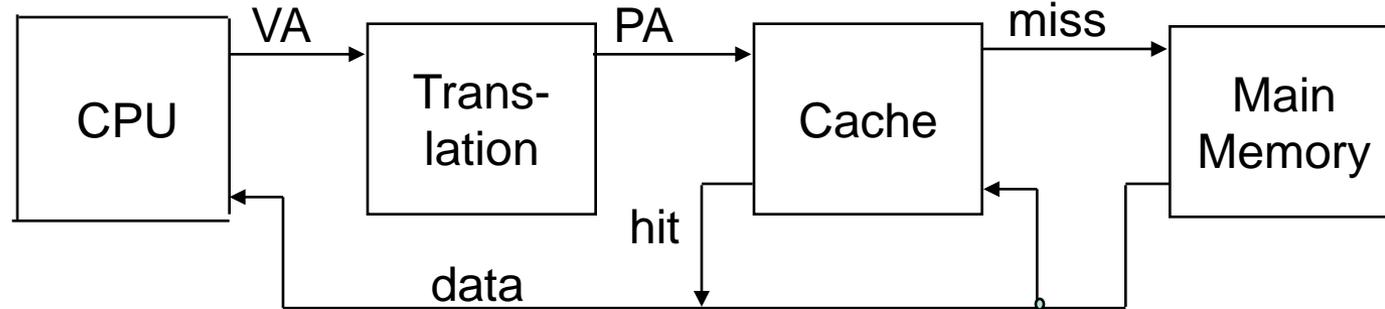
Page Table Operation

■ Checking Protection

- Access rights field indicate allowable access
 - e.g., read-only, read-write, execute-only
 - typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if user doesn't have necessary permission



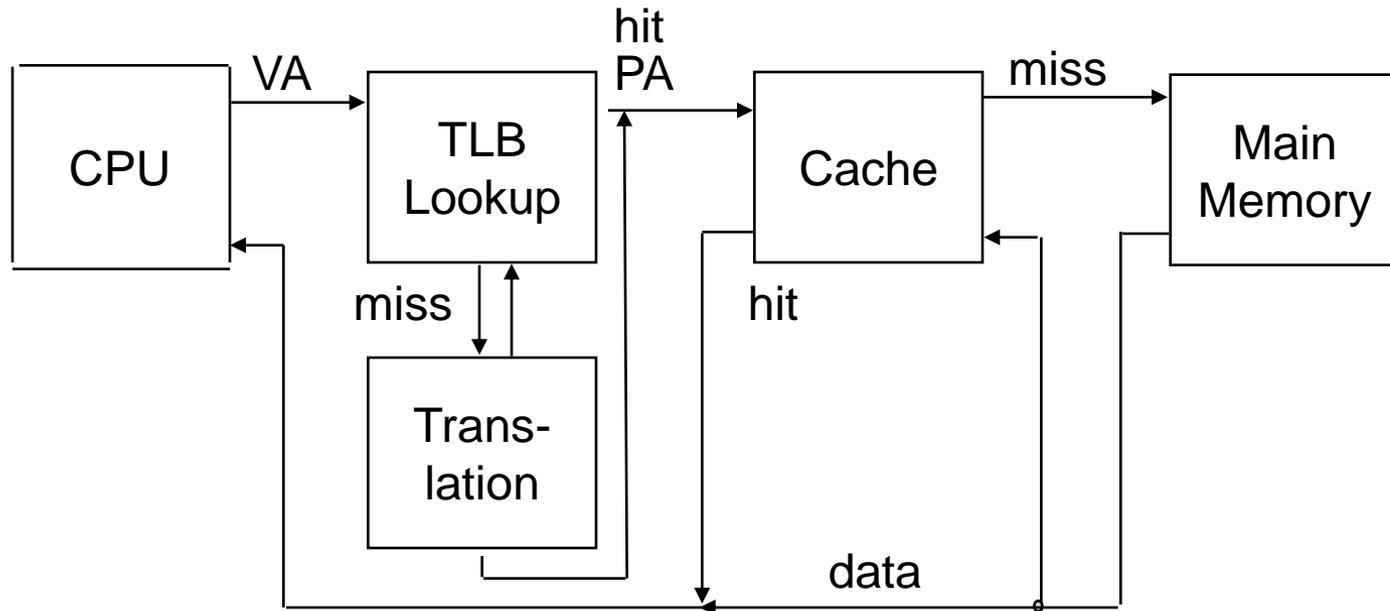
Integrating VM and Cache



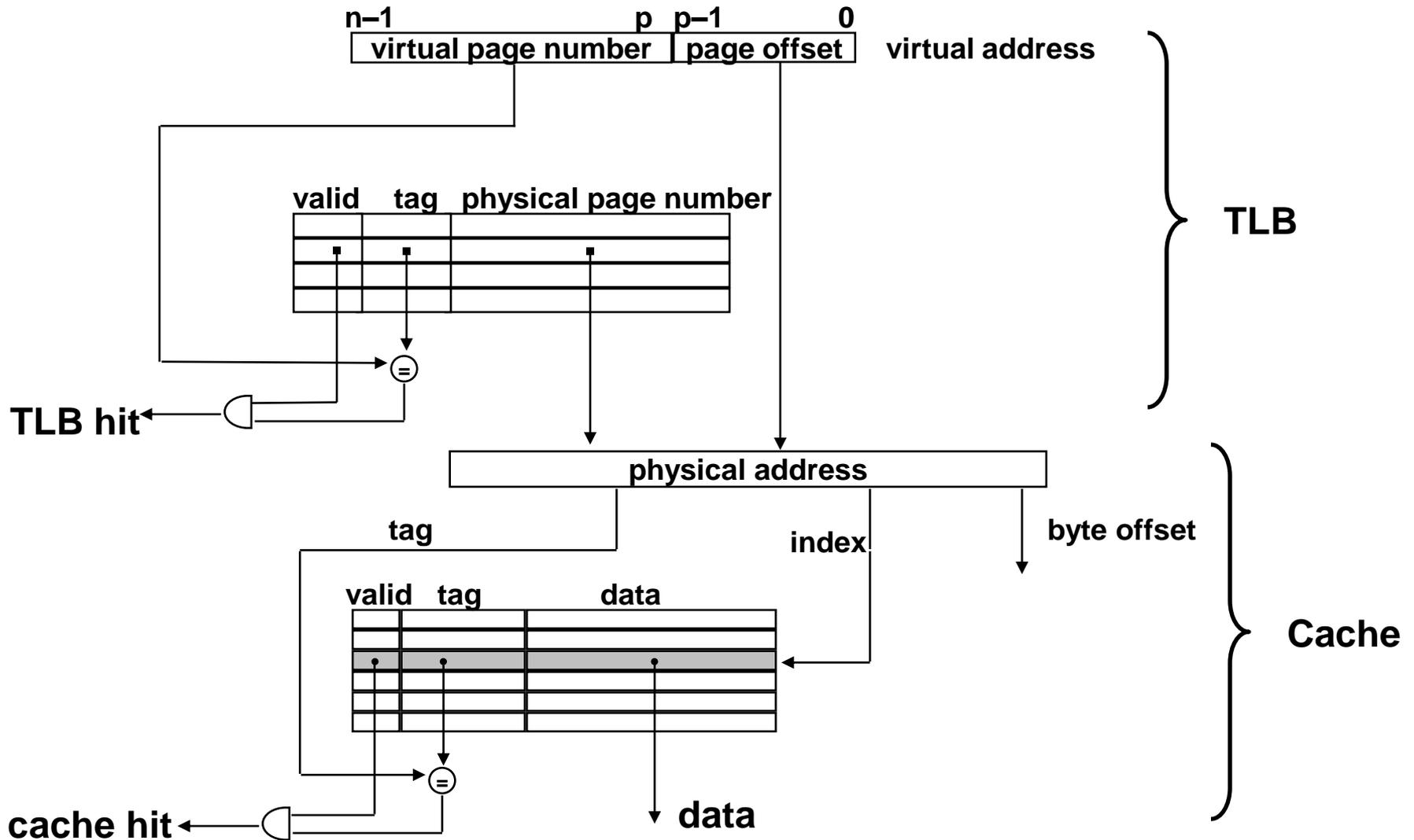
- Most Caches “Physically Addressed”
 - Accessed by physical addresses
 - Allows multiple processes to have blocks in cache at same time
 - Allows multiple processes to share pages
 - Cache doesn’t need to be concerned with protection issues
 - Access rights checked as part of address translation
- Perform Address Translation Before Cache Lookup
 - But this could involve a memory access itself (of the PTE)
 - Of course, page table entries can also become cached

Speeding up Translation with a TLB

- “Translation Lookaside Buffer” (TLB)
 - Small hardware cache in MMU
 - Maps virtual page numbers to physical page numbers
 - Contains complete page table entries for small number of pages

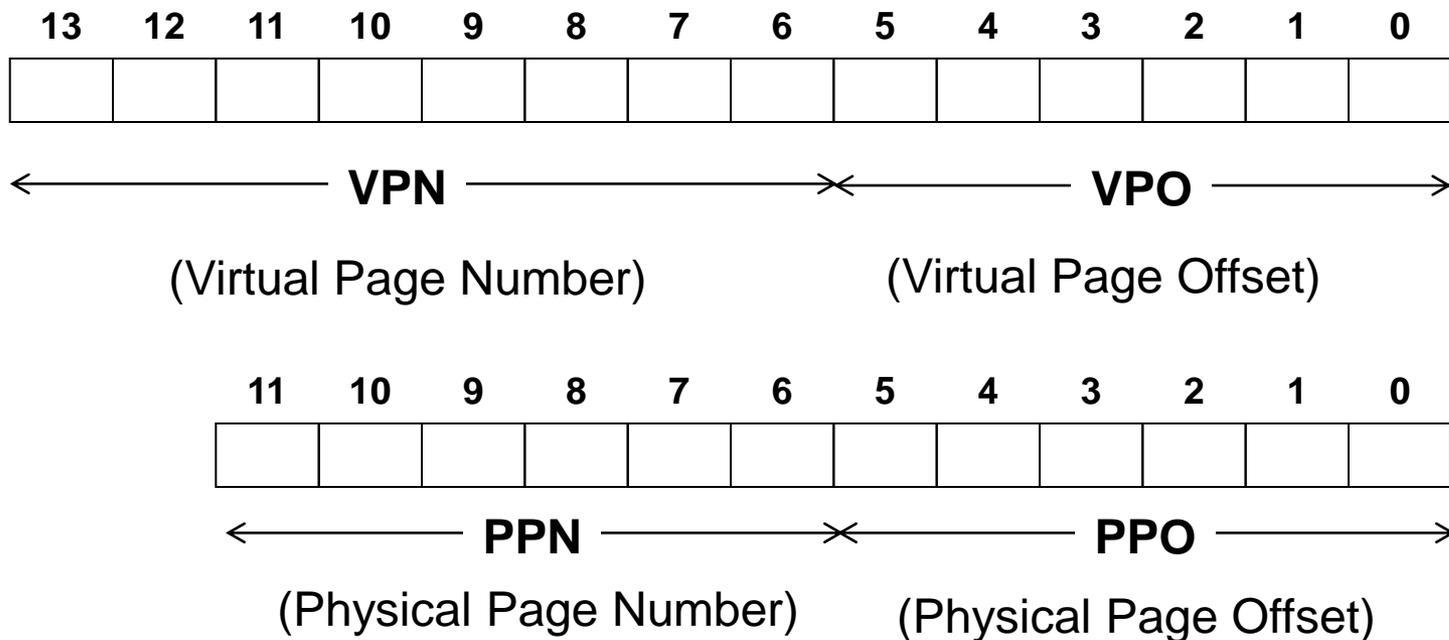


Address Translation with a TLB



Simple Memory System Example

- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 bytes



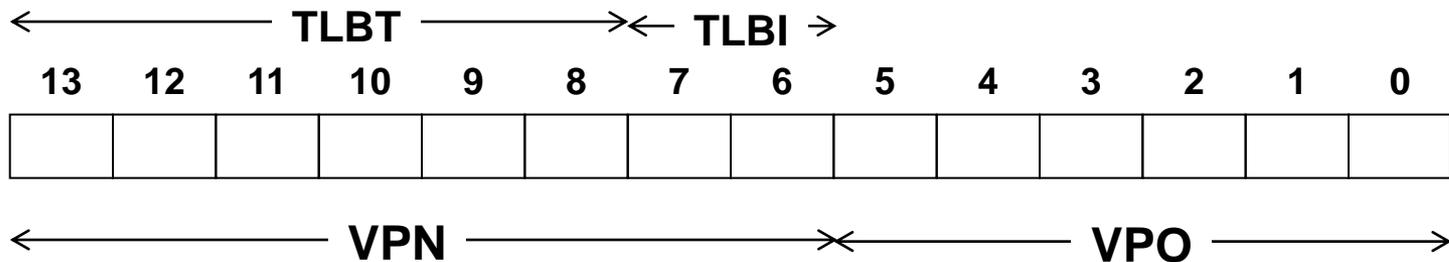
Simple Memory System Page Table

- Only show 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

Simple Memory System TLB

- TLB
 - 16 entries
 - 4-way associative

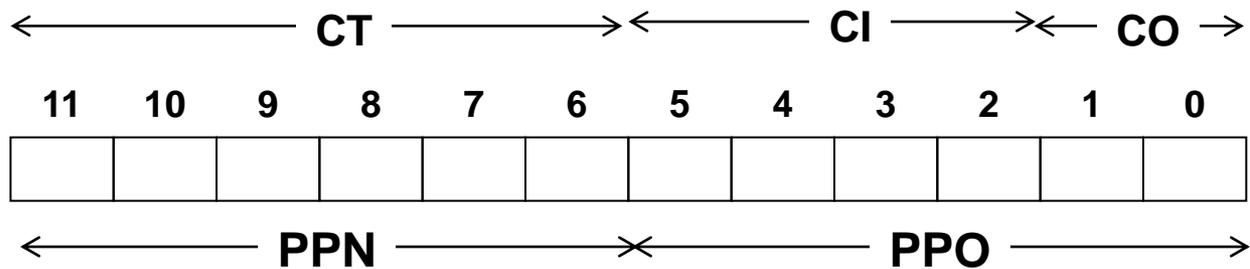


Set	Tag	PPN	Valid									
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

Simple Memory System Cache

Cache

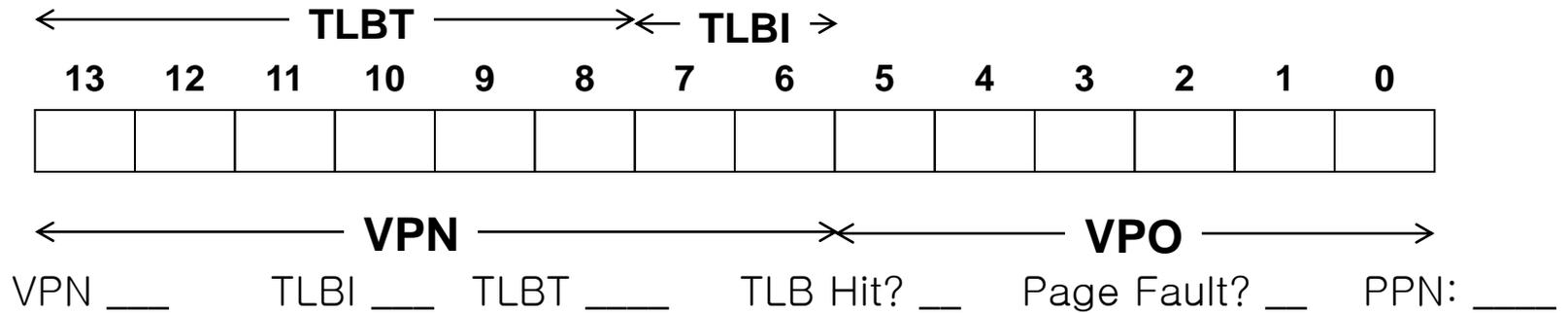
- 16 lines
- 4-byte line size
- Direct mapped



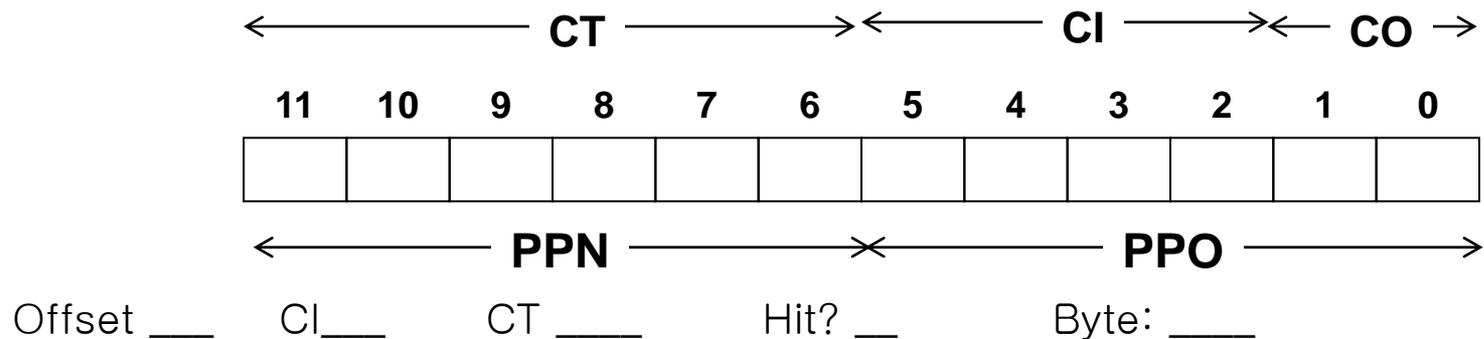
Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

Address Translation Example #1

- Virtual Address 0x03D4

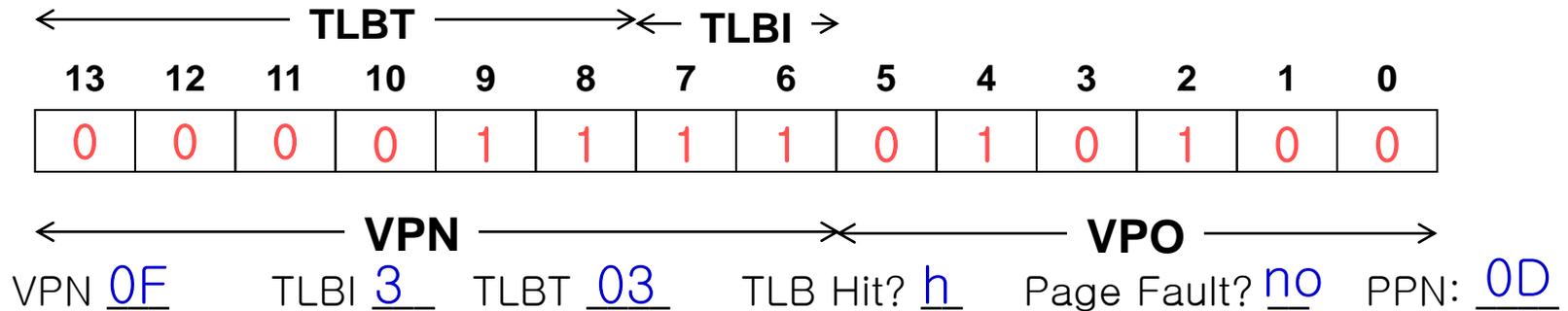


- Physical Address

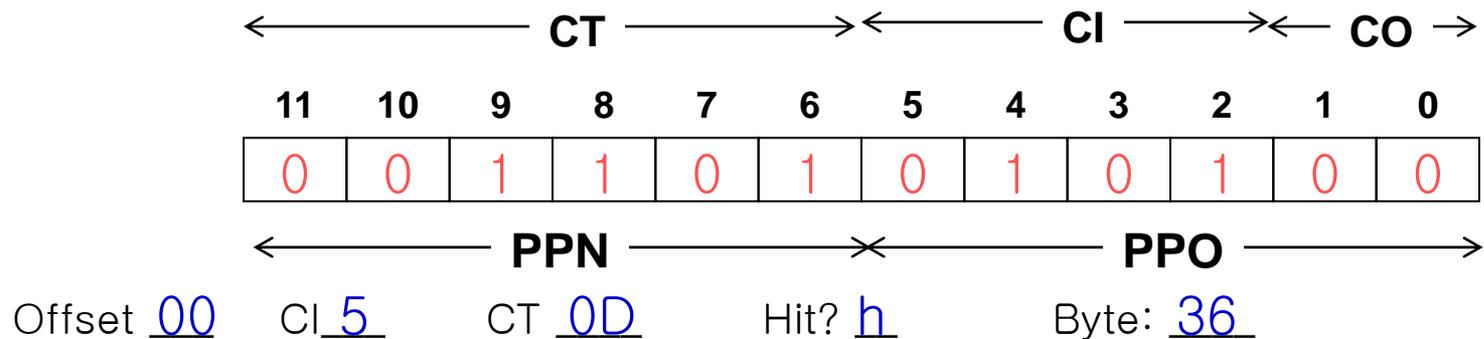


Address Translation Example #1

- Virtual Address 0x03D4

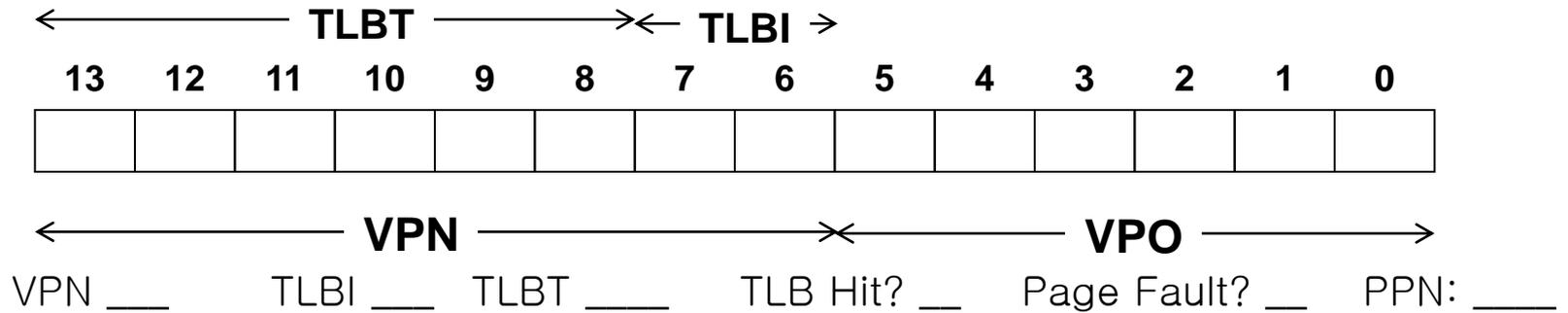


- Physical Address

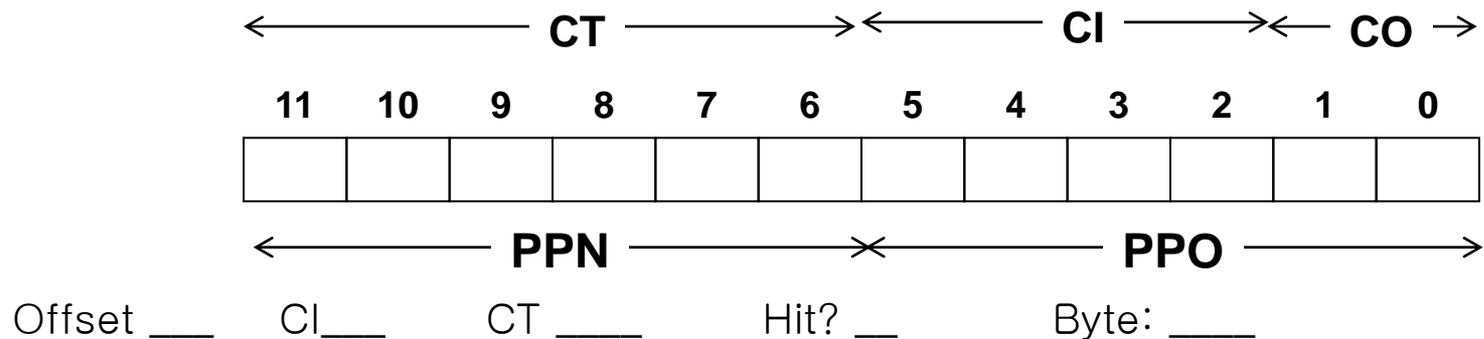


Address Translation Example #2

- Virtual Address 0x038F

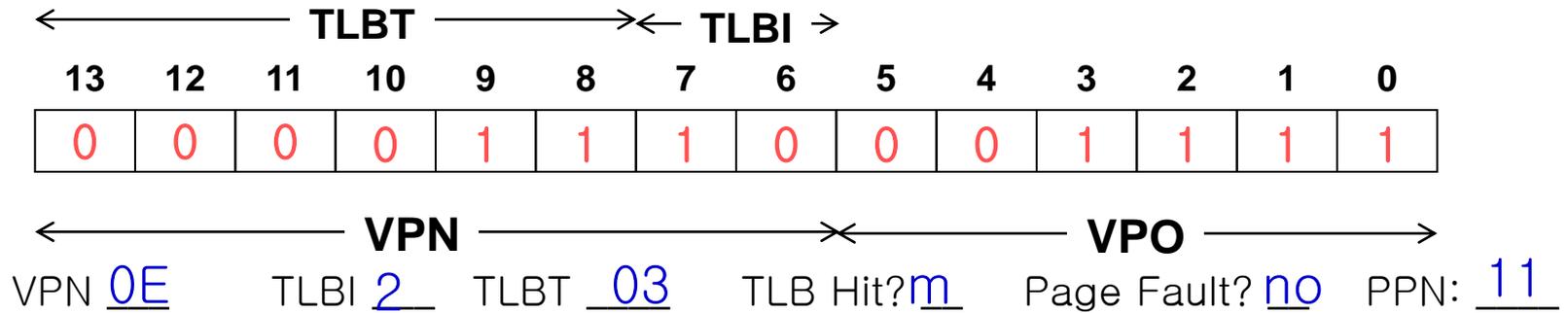


- Physical Address

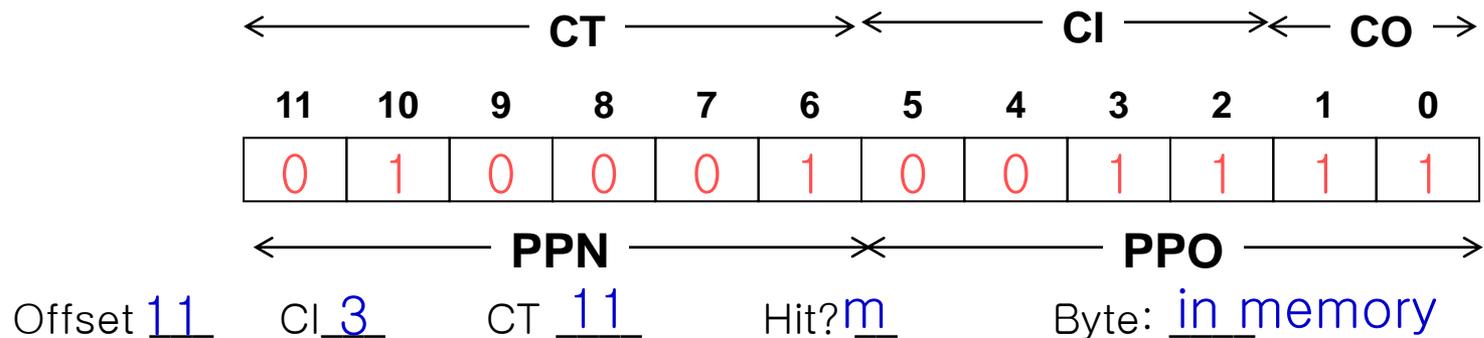


Address Translation Example #2

- Virtual Address 0x038F

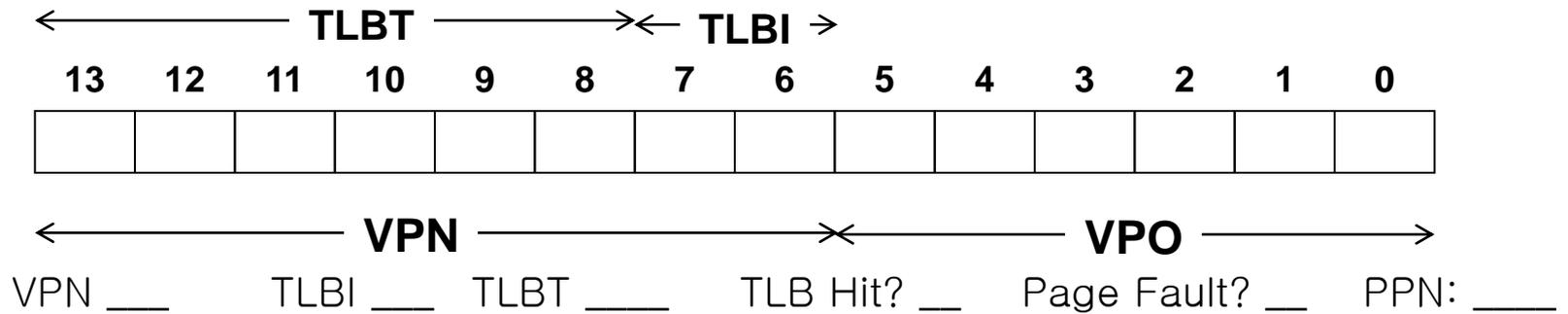


- Physical Address

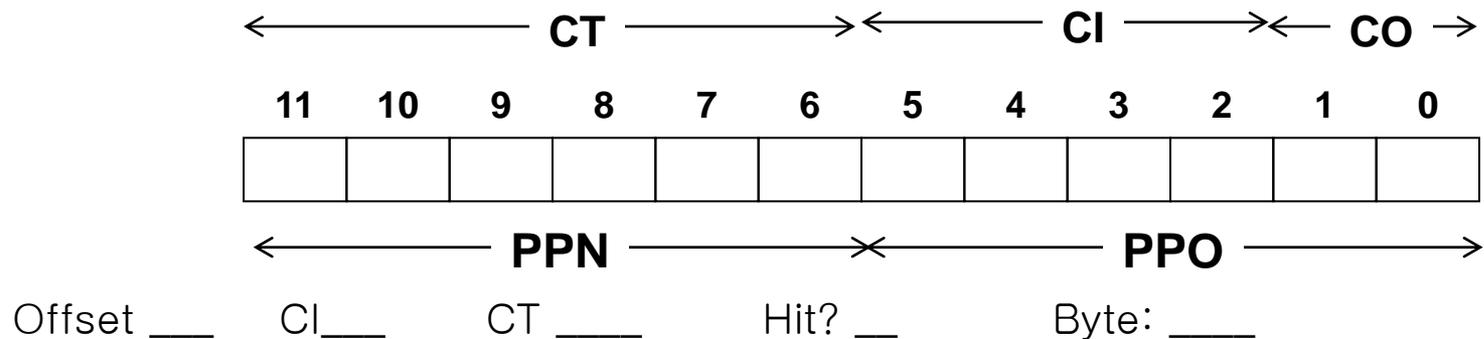


Address Translation Example #3

- Virtual Address 0x0040

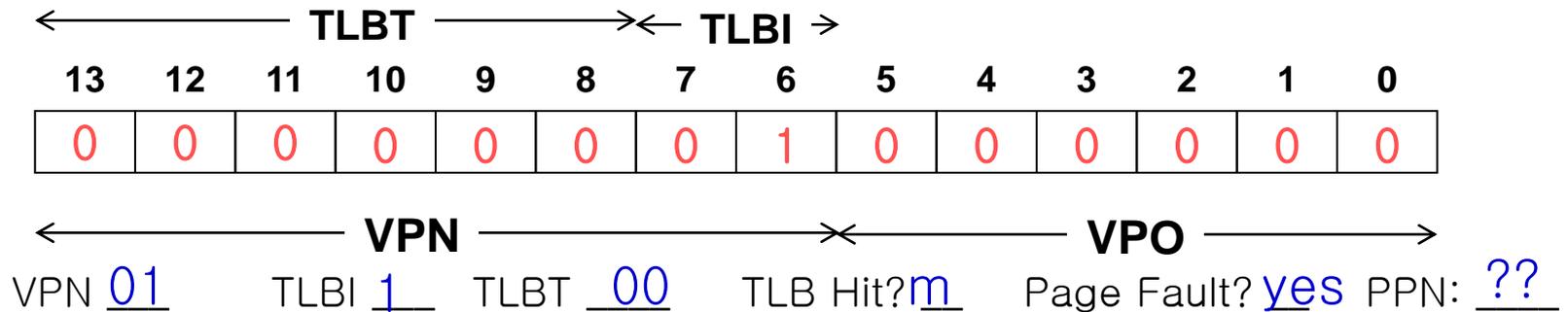


- Physical Address

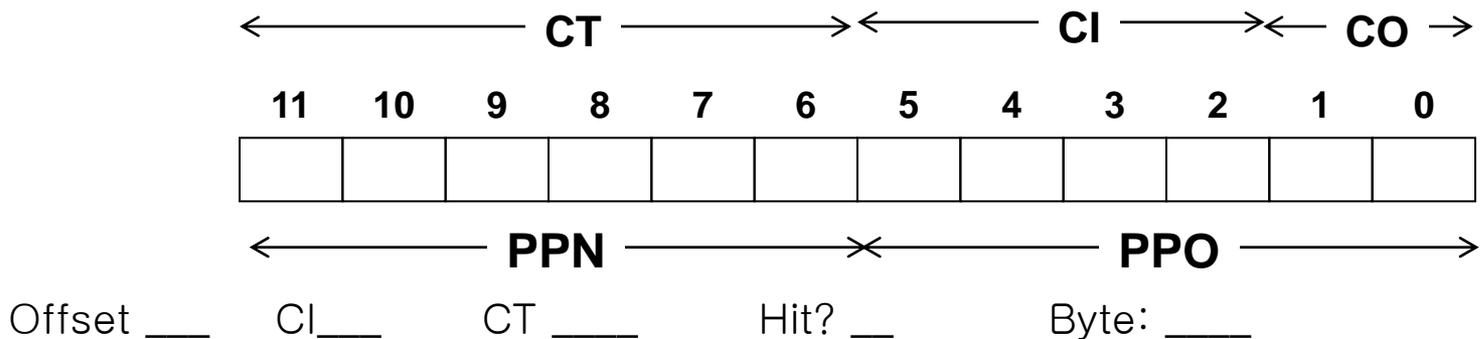


Address Translation Example #3

- Virtual Address 0x0040

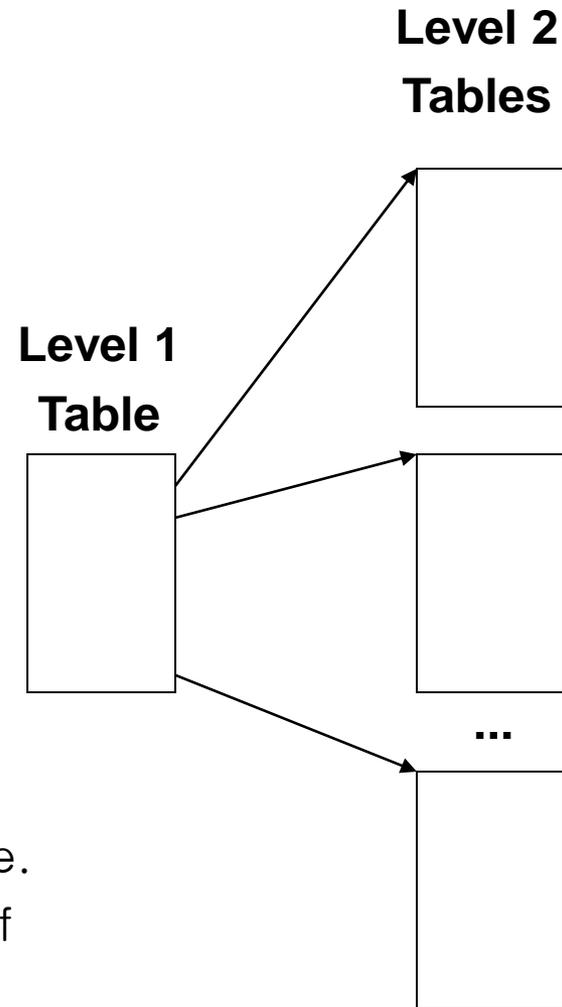


- Physical Address



Multi-Level Page Tables

- Given:
 - 4KB (2^{12}) page size
 - 32-bit address space
 - 4-byte PTE
- Problem:
 - Would need a 4 MB page table!
 - $2^{20} * 4$ bytes
- Common solution
 - multi-level page tables
 - e.g., 2-level table (P6)
 - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
 - Level 2 table: 1024 entries, each of which points to a page



Summary: Main Themes

■ Programmer's View

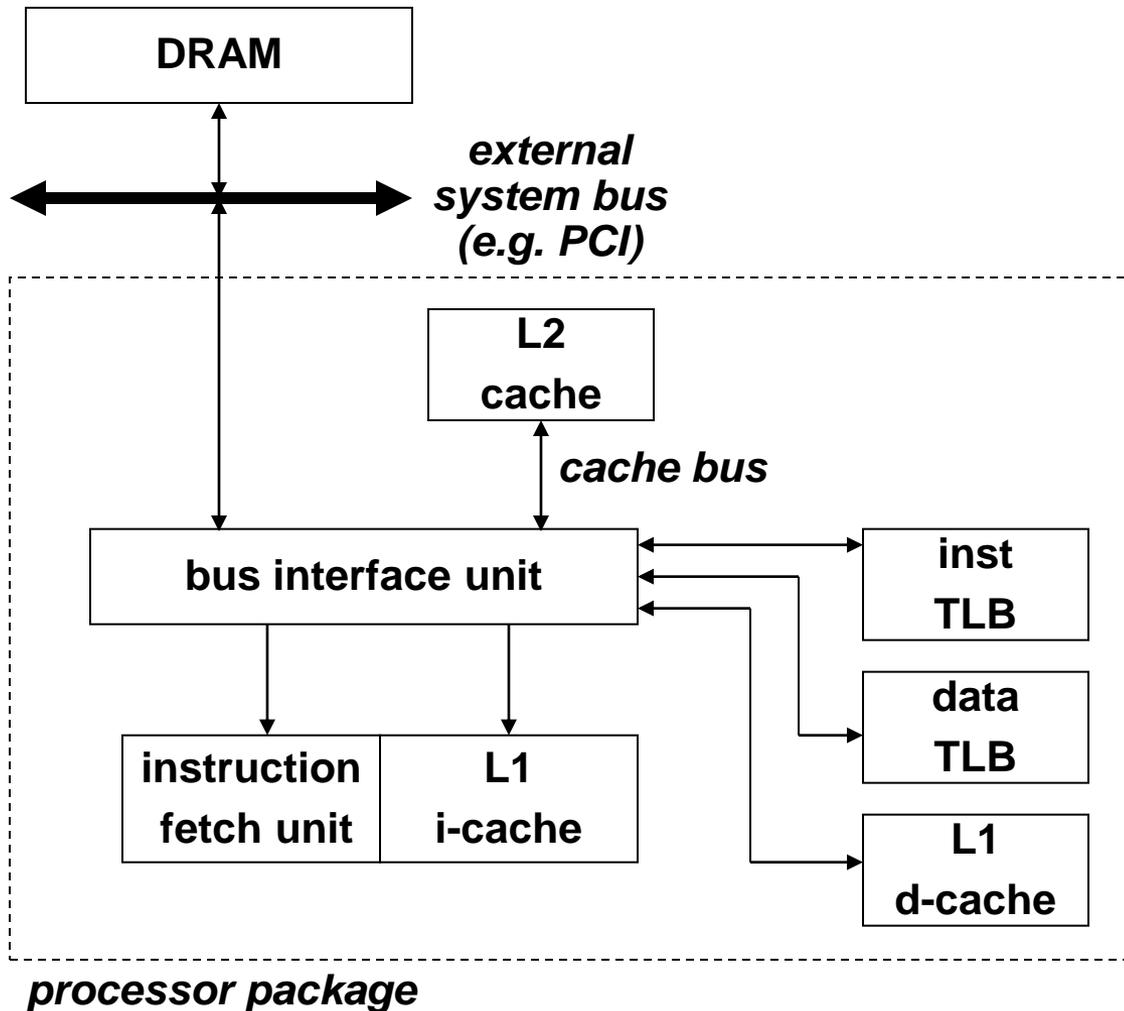
- Large “flat” address space
 - Can allocate large blocks of contiguous addresses
- Processor “owns” machine
 - Has private address space
 - Unaffected by behavior of other processes

■ System View

- User virtual address space created by mapping to set of pages
 - Need not be contiguous
 - Allocated dynamically
 - Enforce protection during address translation
- OS manages many processes simultaneously
 - Continually switching among processes
 - Especially when one must wait for resource
 - E.g., disk I/O to handle page fault

Case Study: The Pentium/Linux Memory system

P6 Memory System



32 bit address space

4 KB page size

L1, L2, and TLBs

- 4-way set associative

inst TLB

- 32 entries
- 8 sets

data TLB

- 64 entries
- 16 sets

L1 i-cache and d-cache

- 16 KB
- 32 Byte line size
- 128 sets

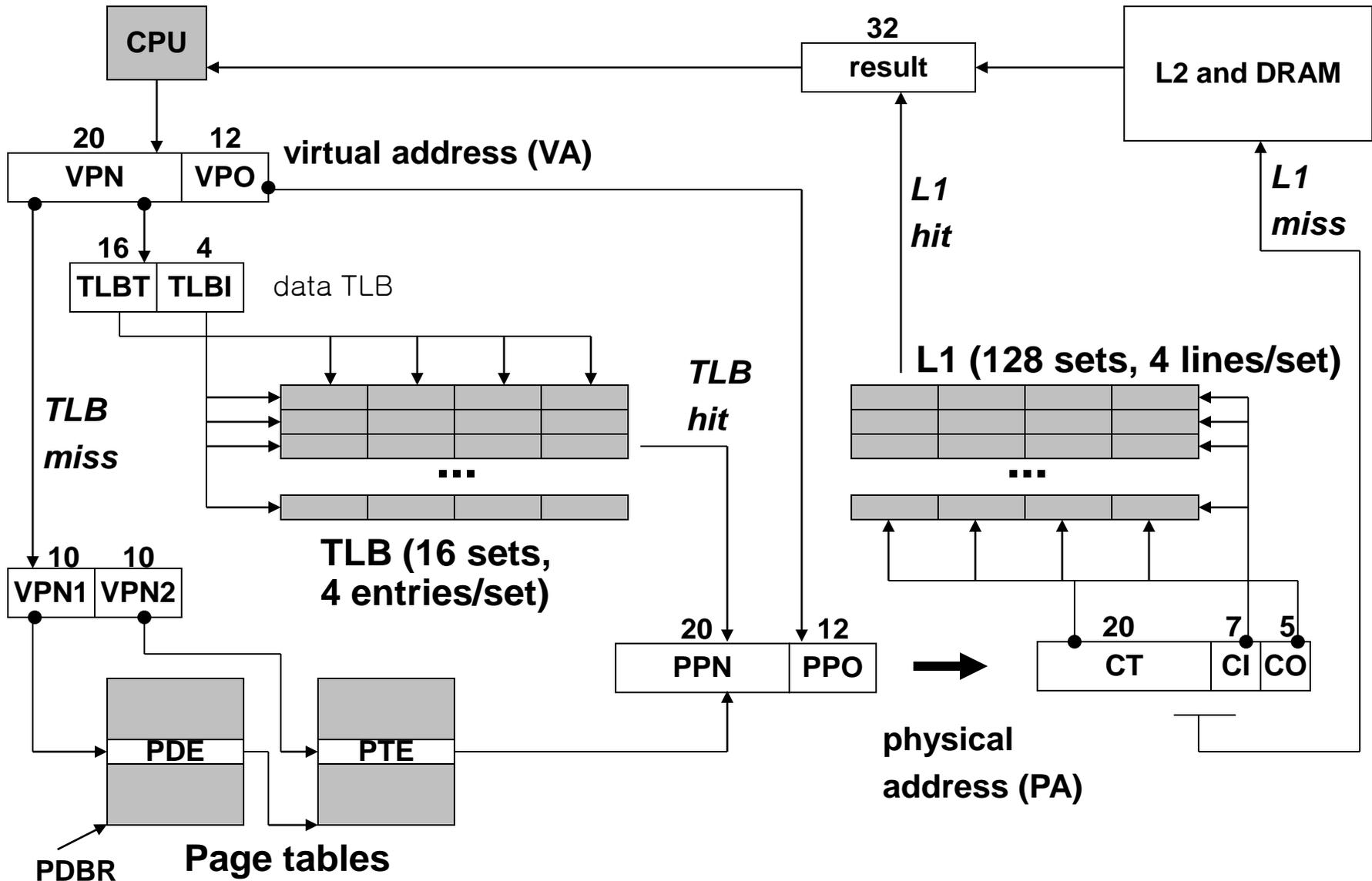
L2 cache

- unified
- 128 KB -- 2 MB

Review of Abbreviations

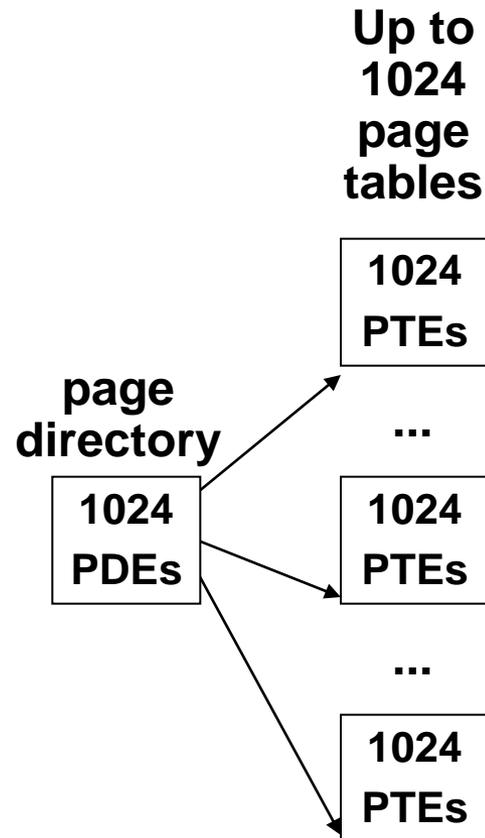
- Symbols:
 - Components of the virtual address (VA)
 - TLBI: TLB index
 - TLBT: TLB tag
 - VPO: virtual page offset
 - VPN: virtual page number
 - Components of the physical address (PA)
 - PPO: physical page offset (same as VPO)
 - PPN: physical page number
 - CO: byte offset within cache line
 - CI: cache index
 - CT: cache tag

Overview of P6 Address Translation



P6 2-level Page Table Structure

- Page directory
 - 1024 4-byte page directory entries (PDEs) that point to page tables
 - one page directory per process.
 - page directory must be in memory when its process is running
 - always pointed to by PD BR
- Page tables:
 - 1024 4-byte page table entries (PTEs) that point to pages.
 - page tables can be paged in and out.



P6 Page Directory Entry (PDE)

31	12 11	9	8	7	6	5	4	3	2	1	0
Page table physical base addr		Avail	G	PS		A	CD	WT	U/S	R/W	P=1

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Avail: These bits available for system programmers

G: global page (don't evict from TLB on task switch)

PS: page size 4K (0) or 4M (1)

A: accessed (set by MMU on reads and writes, cleared by software)

CD: cache disabled (1) or enabled (0)

WT: write-through or write-back cache policy for this page table

U/S: user or supervisor mode access

R/W: read-only or read-write access

P: page table is present in memory (1) or not (0)

31	1	0
Available for OS (page table location in secondary storage)		P=0

P6 Page Table Entry (PTE)

31	12 11	9	8	7	6	5	4	3	2	1	0
Page physical base address		Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don't evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

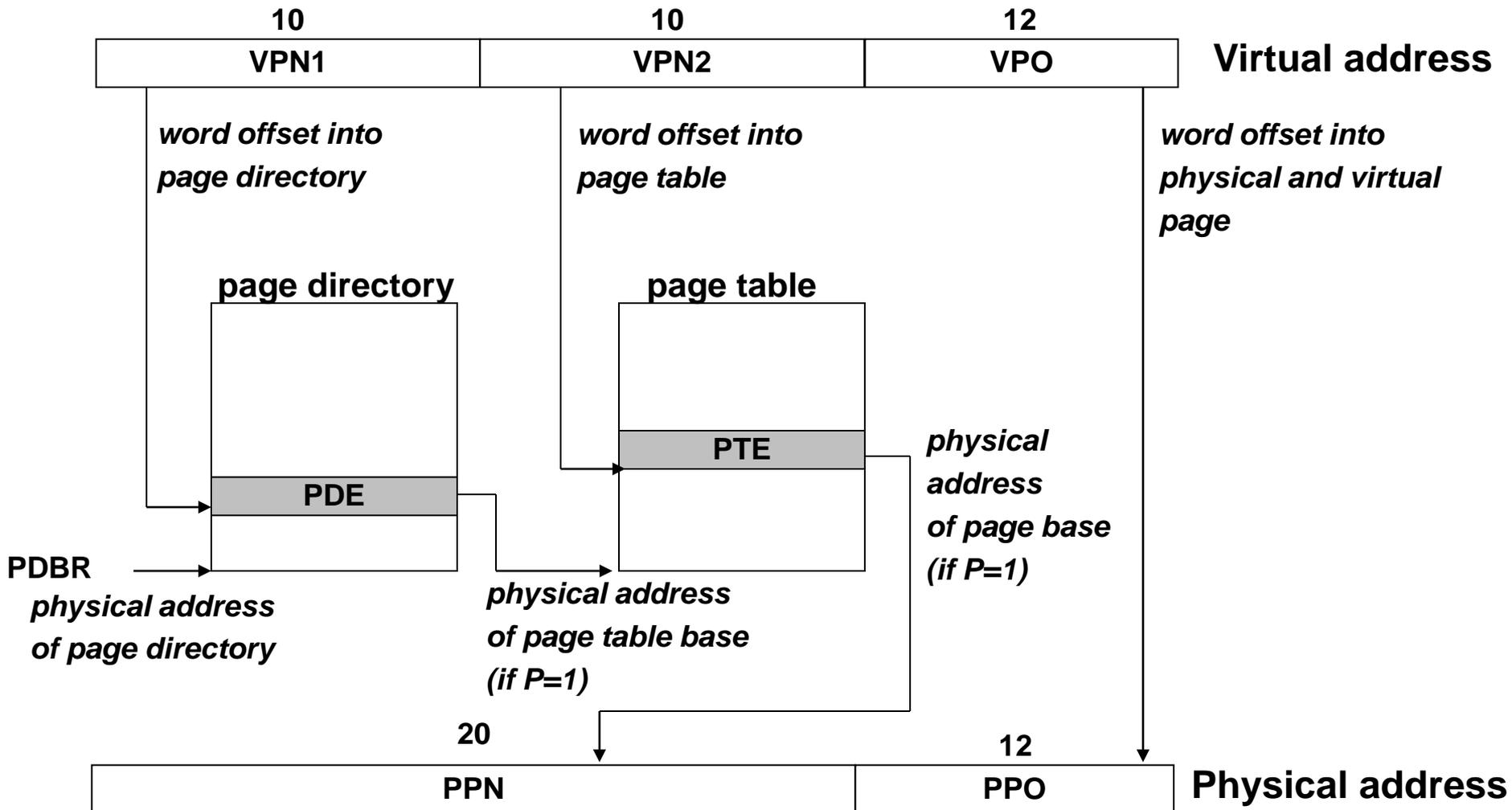
U/S: user/supervisor

R/W: read/write

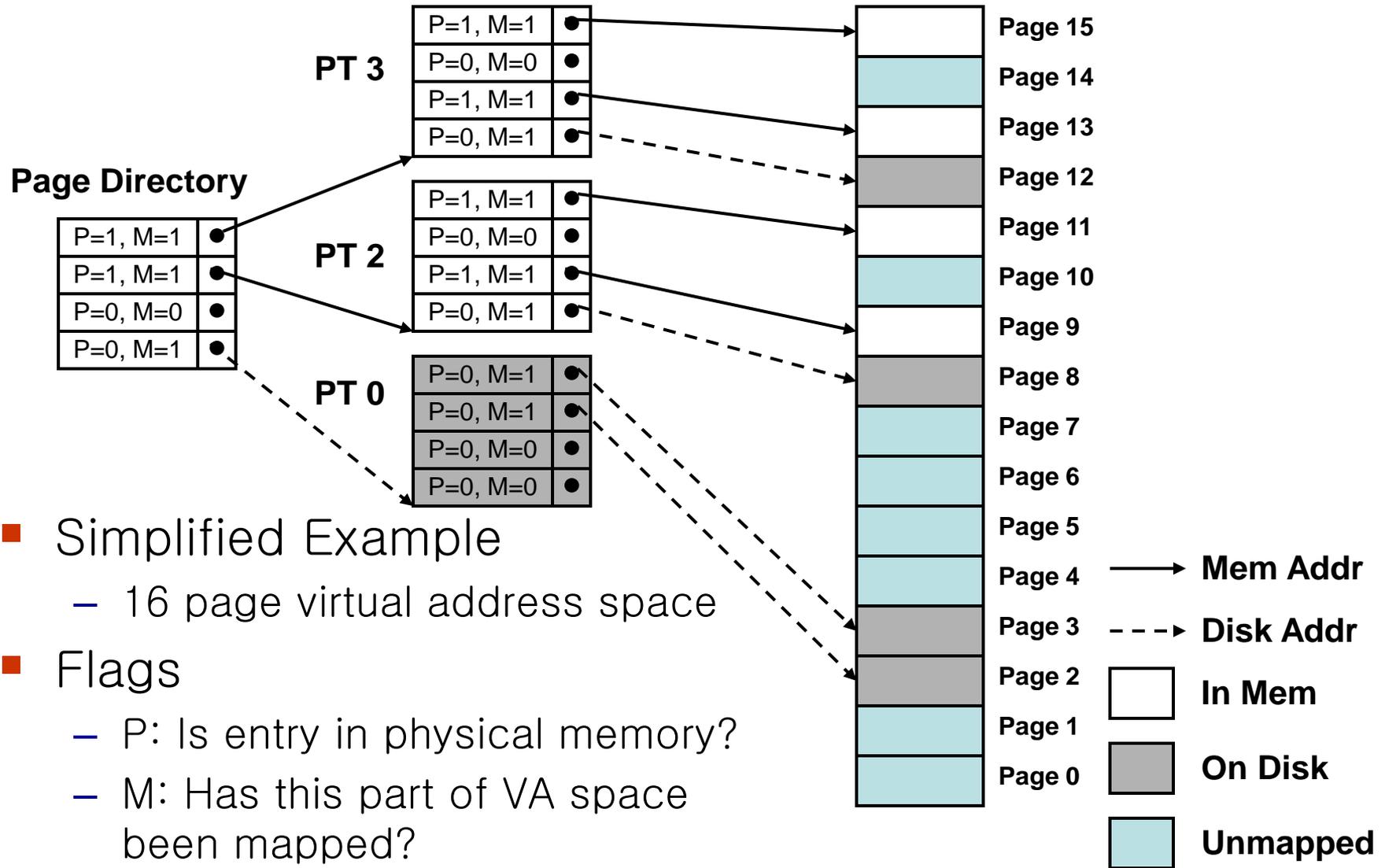
P: page is present in physical memory (1) or not (0)

31	1	0
Available for OS (page location in secondary storage)		P=0

How P6 Page Tables Map Virtual Addresses to Physical Ones

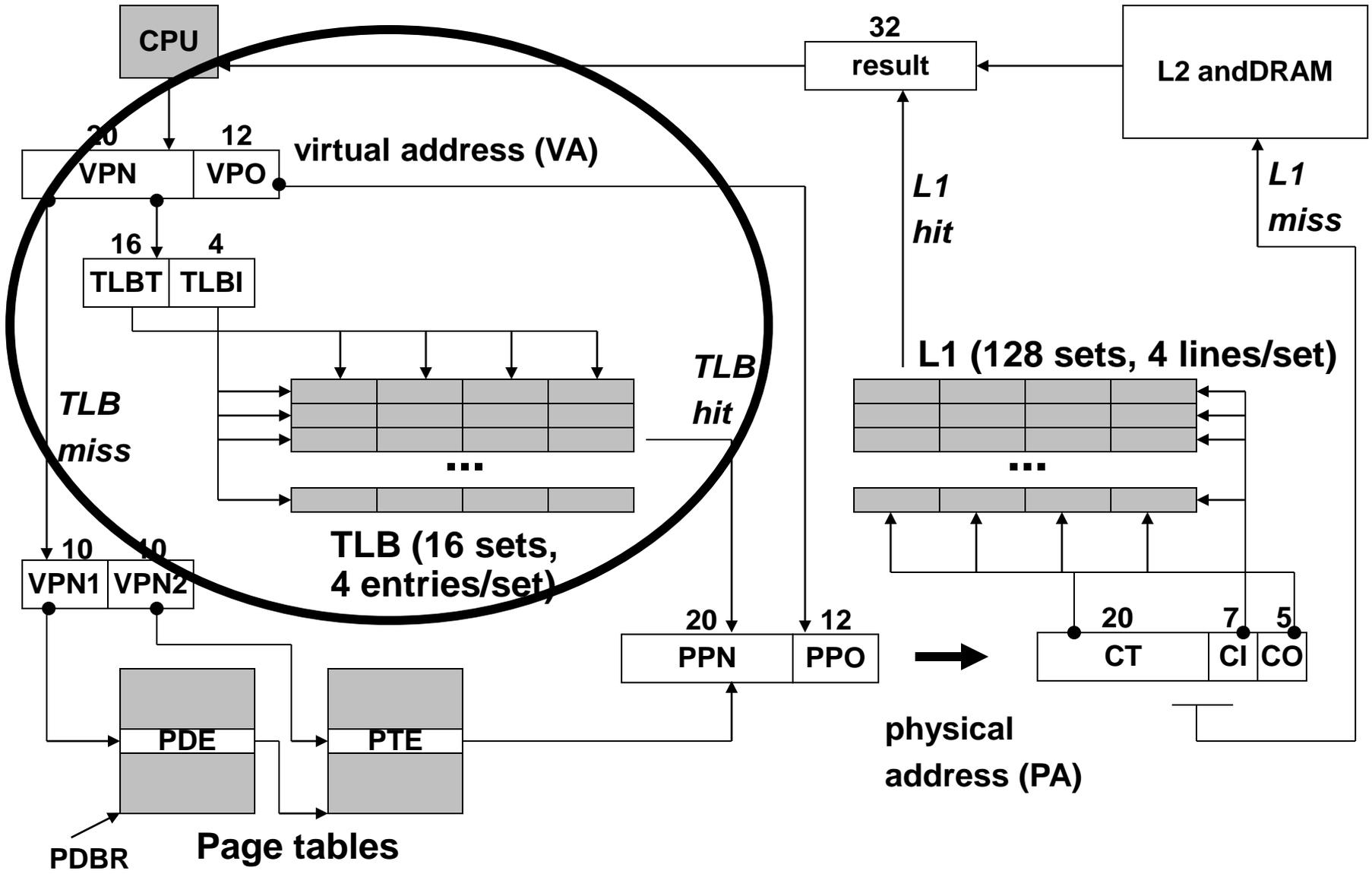


Representation of Virtual Address Space



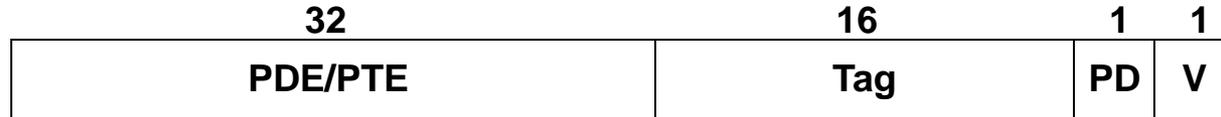
- Simplified Example
 - 16 page virtual address space
- Flags
 - P: Is entry in physical memory?
 - M: Has this part of VA space been mapped?

P6 TLB Translation

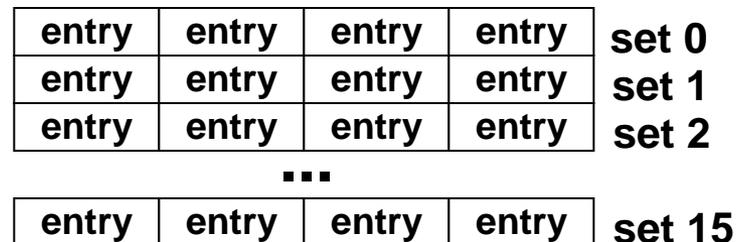


P6 TLB

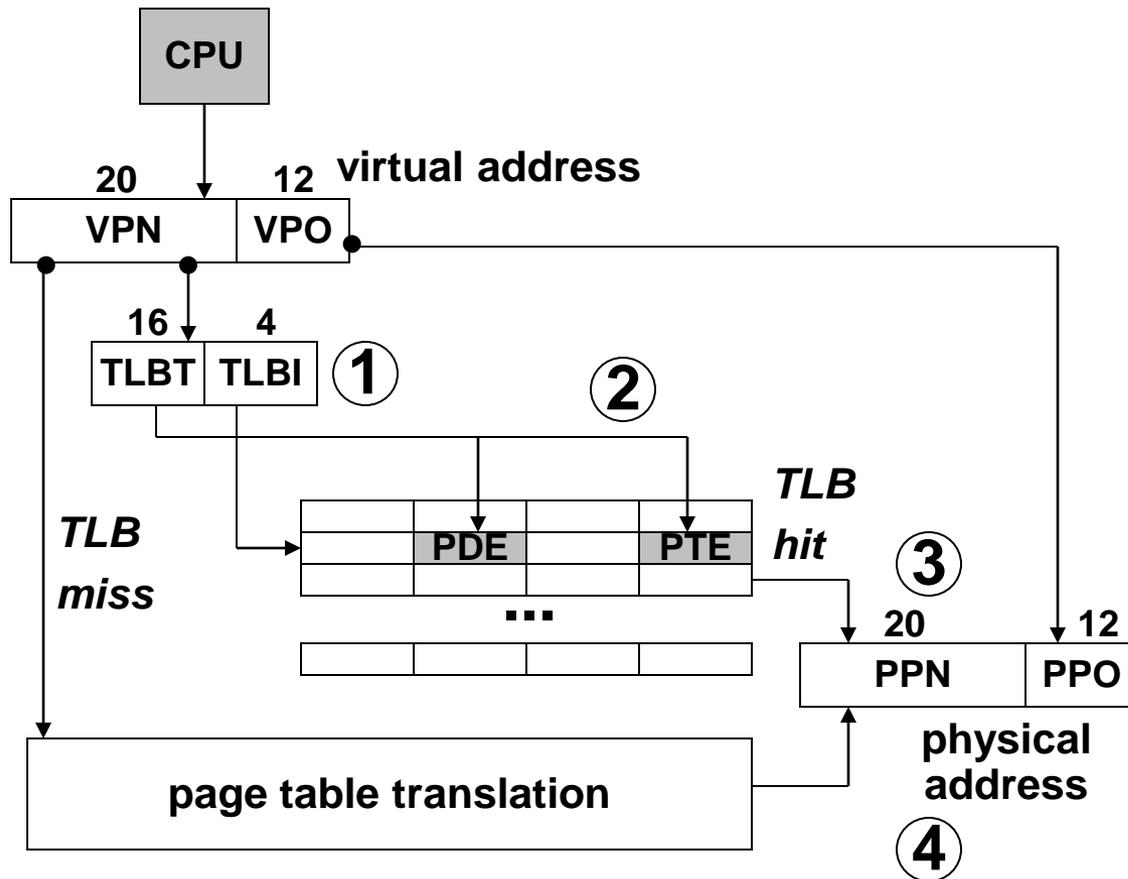
- TLB entry (not all documented, so this is speculative):



- V: indicates a valid (1) or invalid (0) TLB entry
 - PD: is this entry a PDE (1) or a PTE (0)?
 - tag: disambiguates entries cached in the same set
 - PDE/PTE: page directory or page table entry
- Structure of the data TLB:
 - 16 sets, 4 entries/set

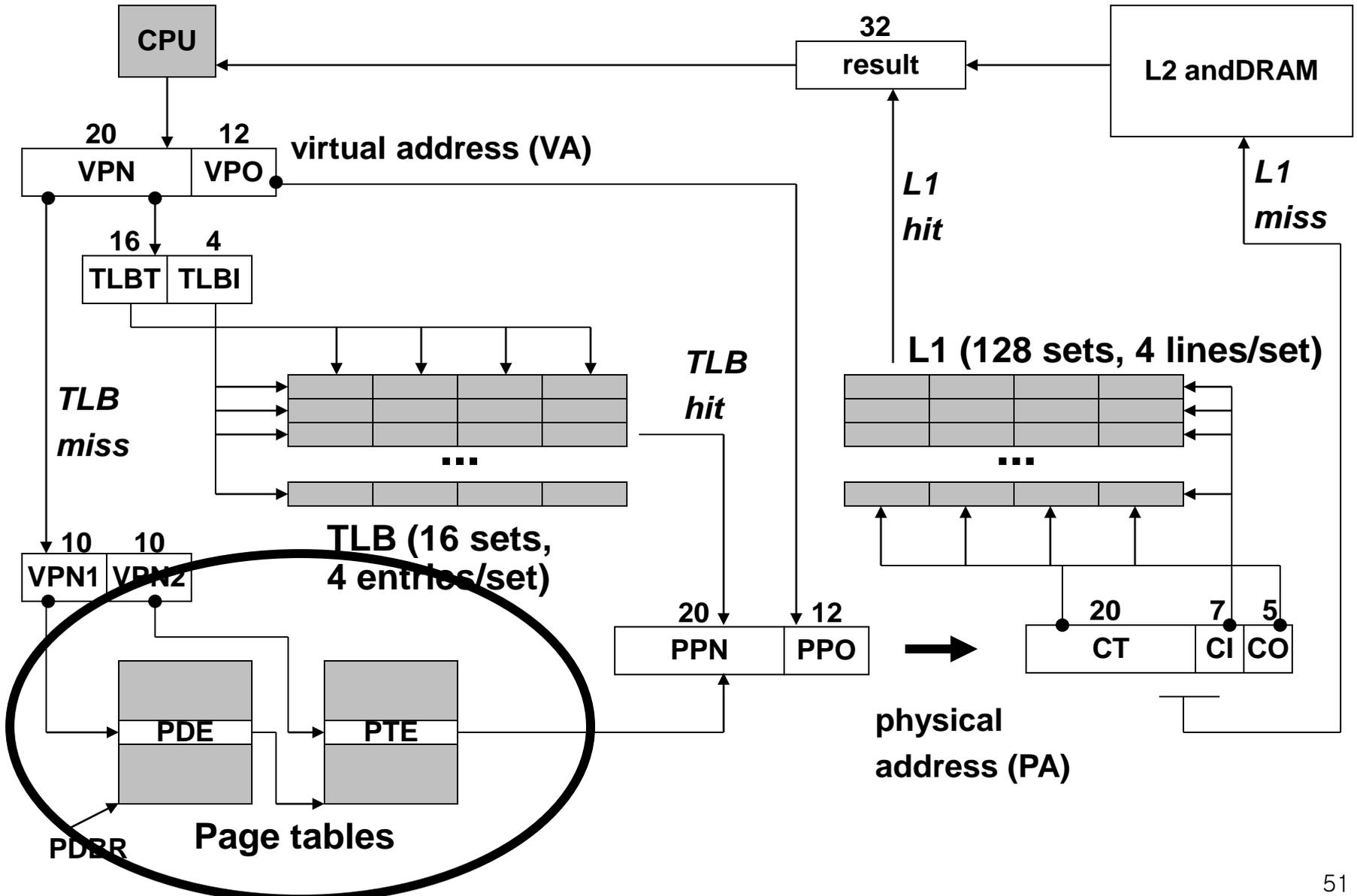


Translating with the P6 TLB

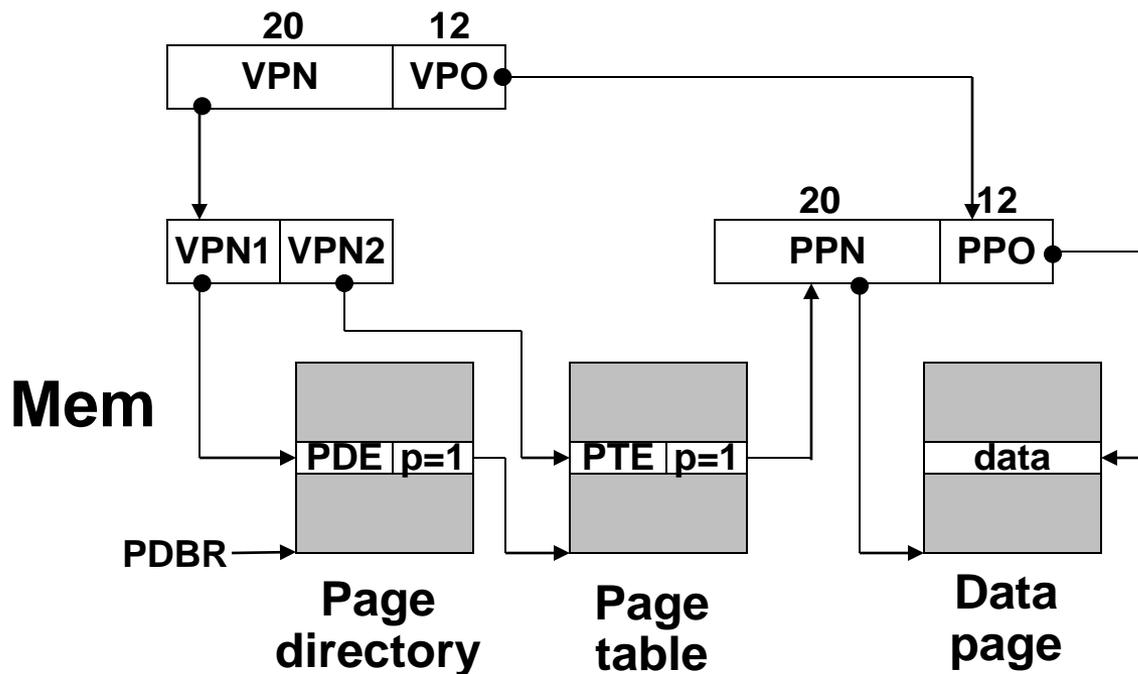


- 1. Partition VPN into TLBT and TLBI.
- 2. Is the PTE for VPN cached in set TLBI?
 - 3. Yes: then build physical address.
- 4. No: then read PTE (and PDE if not cached) from memory and build physical address.

P6 page table translation

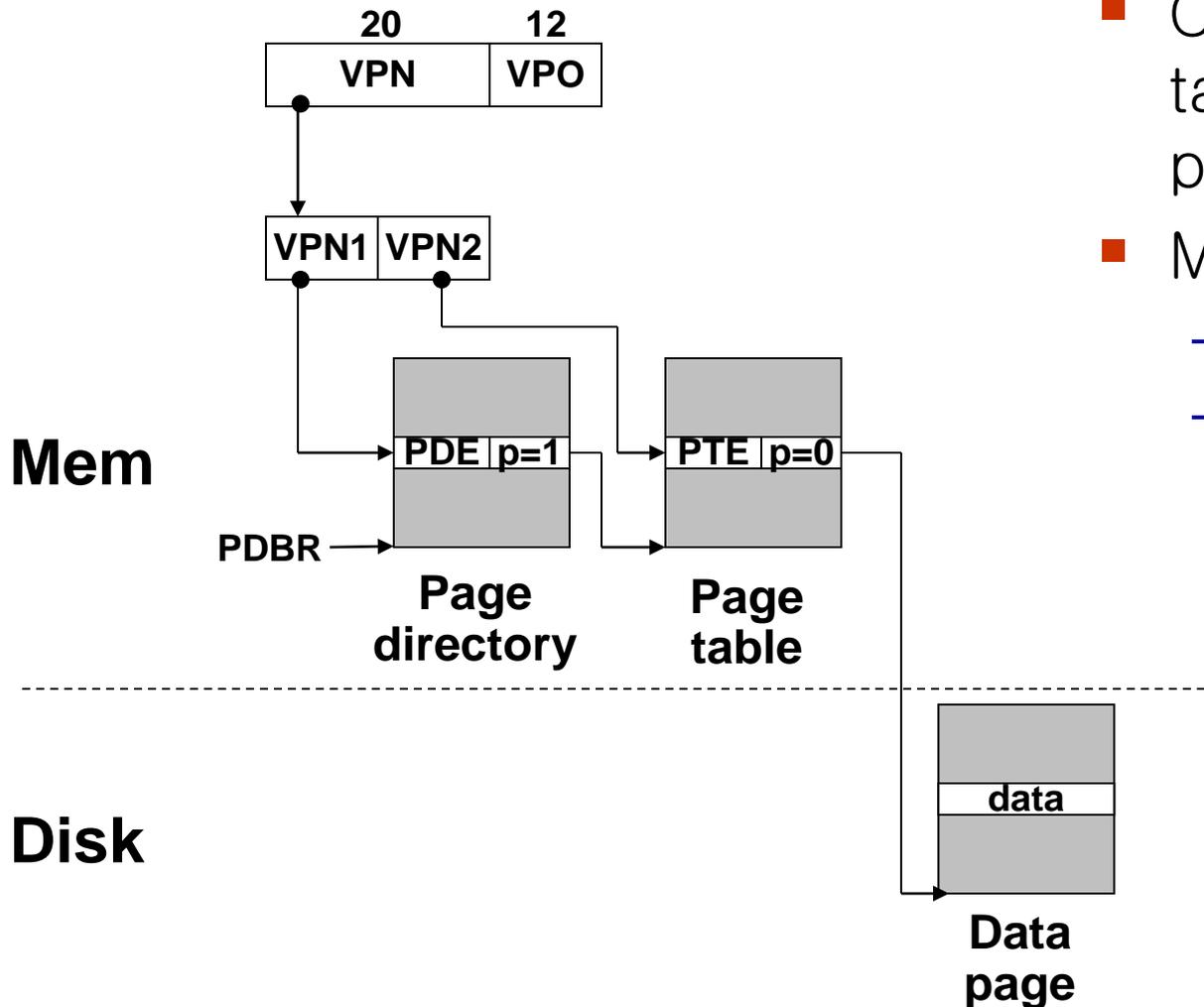


Translating with the P6 Page Tables (case 1/1)



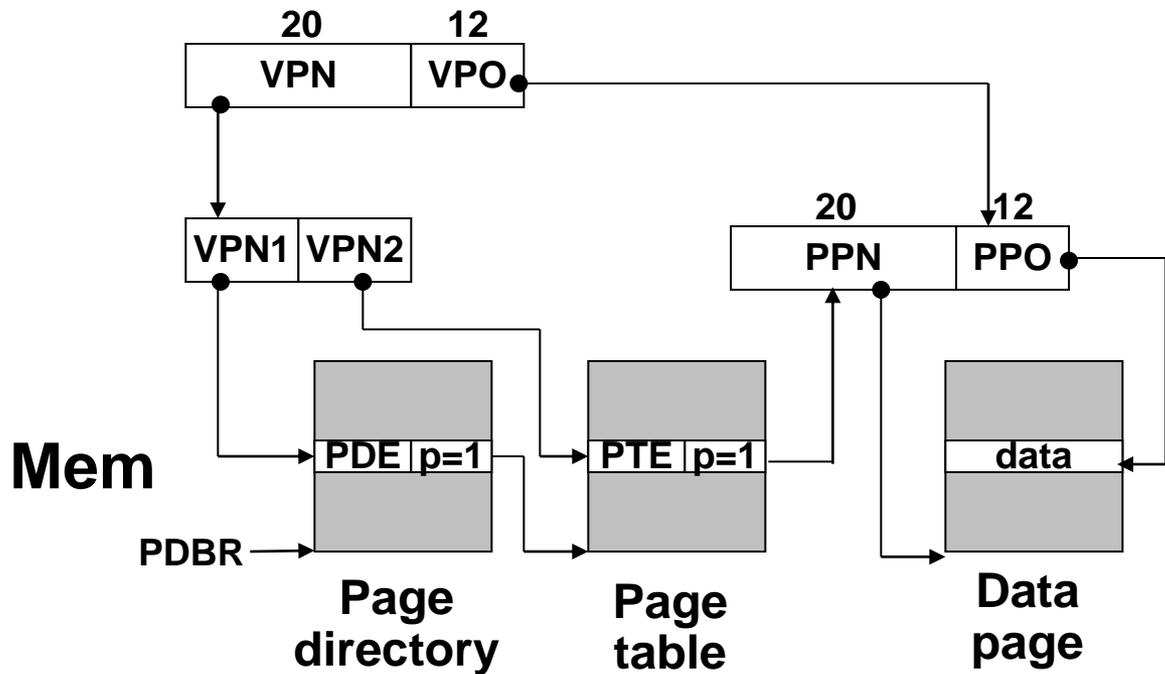
- Case 1/1: page table and page present.
- MMU Action:
 - MMU builds physical address and fetches data word.
- OS action
 - none

Translating with the P6 Page Tables (case 1/0)



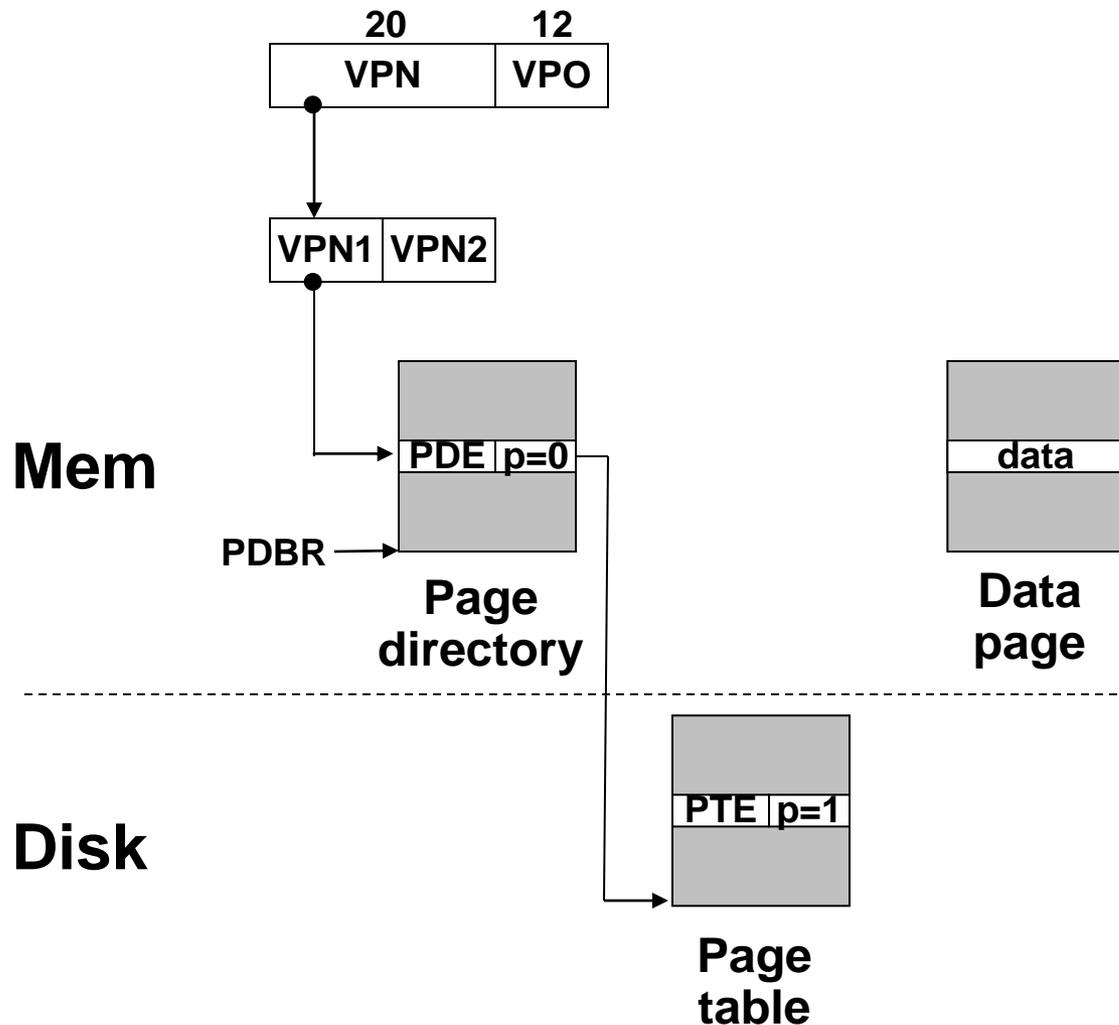
- Case 1/0: page table present but page missing.
- MMU Action:
 - page fault exception
 - handler receives the following args:
 - VA that caused fault
 - fault caused by non-present page or page-level protection violation
 - read/write
 - user/supervisor

Translating with the P6 Page Tables (case 1/0, cont)



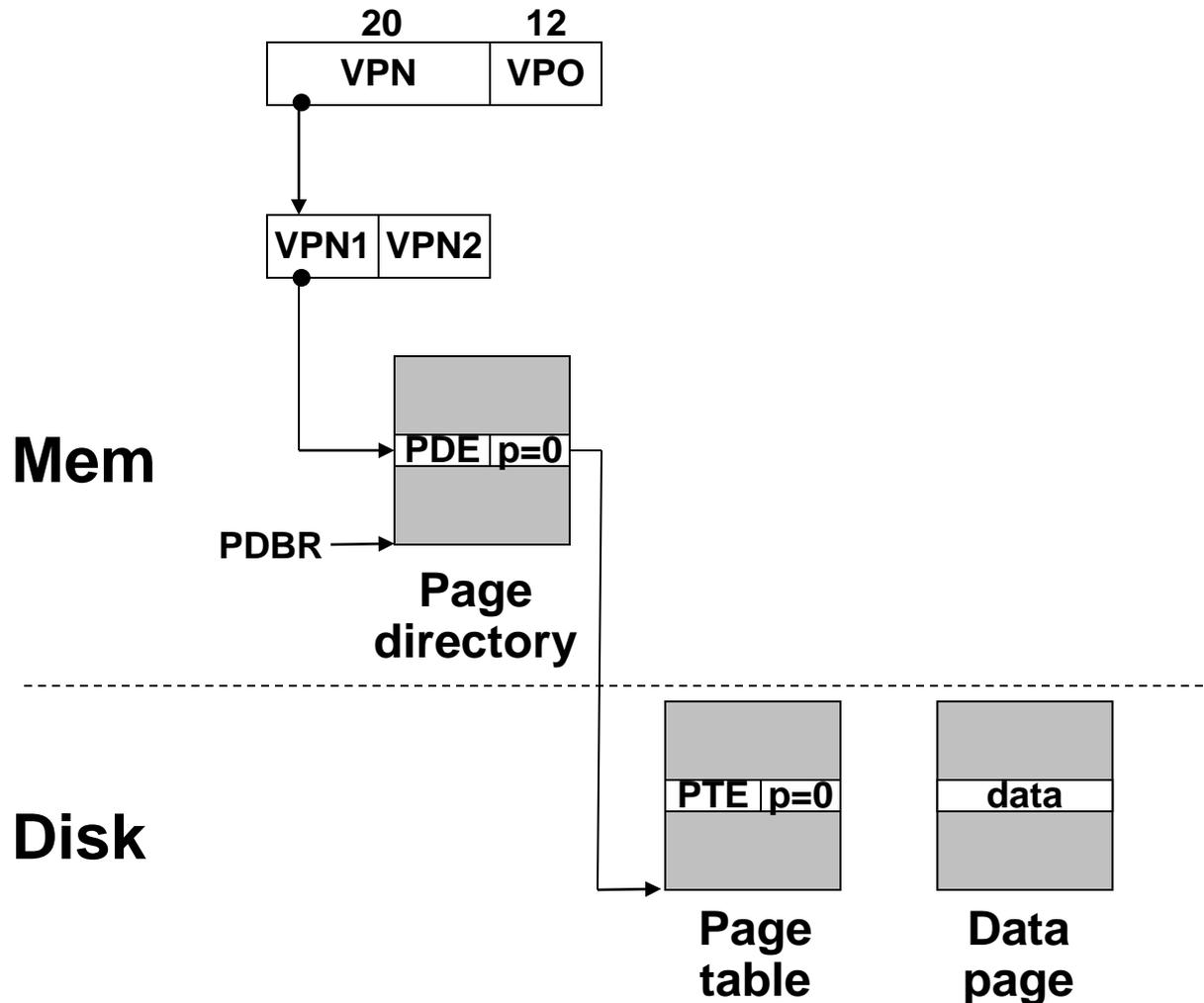
- OS Action:
 - Check for a legal virtual address.
 - Read PTE through PDE.
 - Find free physical page (swapping out current page if necessary)
 - Read virtual page from disk and copy to virtual page
 - Restart faulting instruction by returning from exception handler.

Translating with the P6 Page Tables (case 0/1)



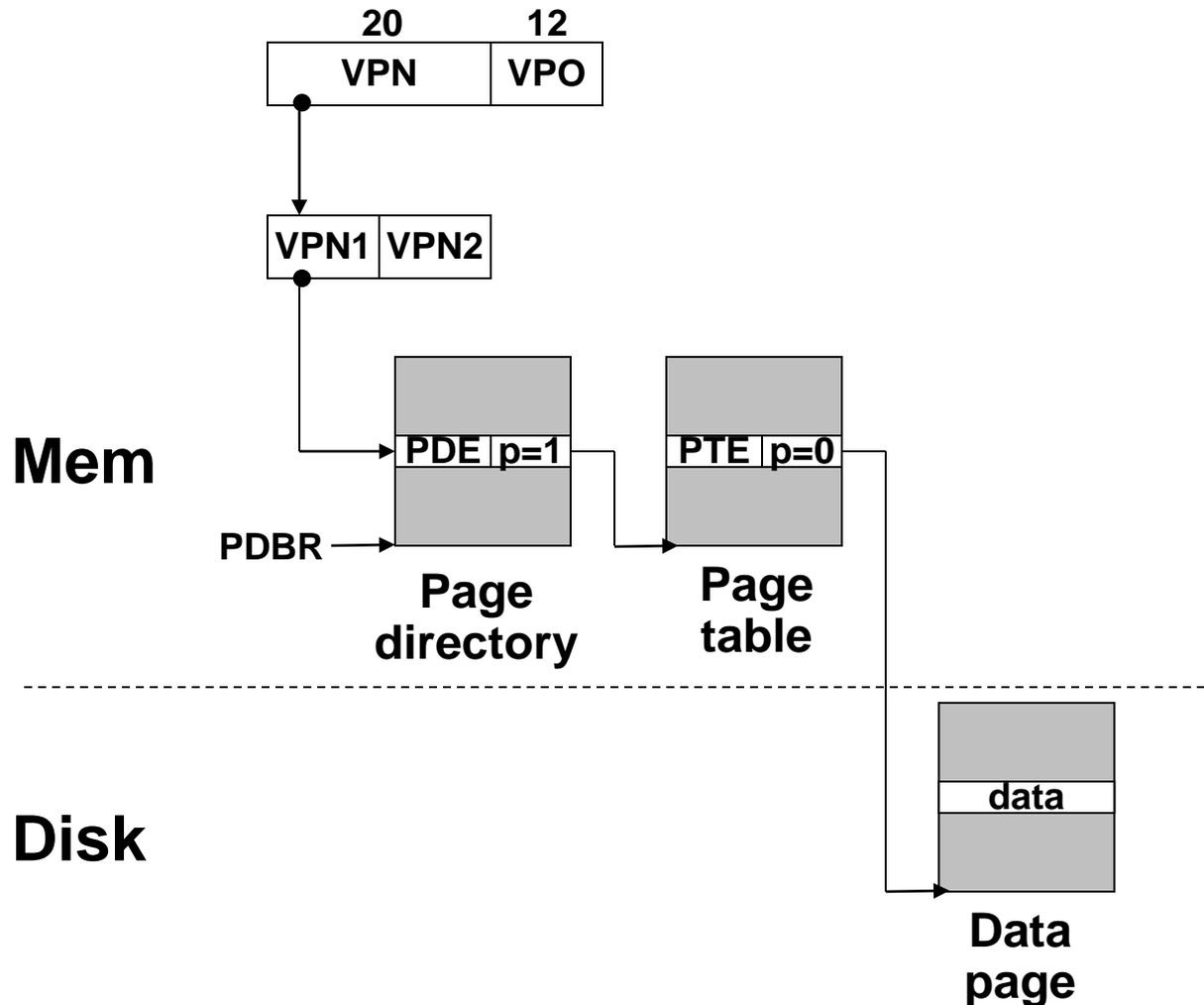
- Case 0/1: page table missing but page present.
- Introduces consistency issue.
 - potentially every page out requires update of disk page table.
- Linux disallows this
 - if a page table is swapped out, then swap out its data pages too.

Translating with the P6 Page Tables (case 0/0)



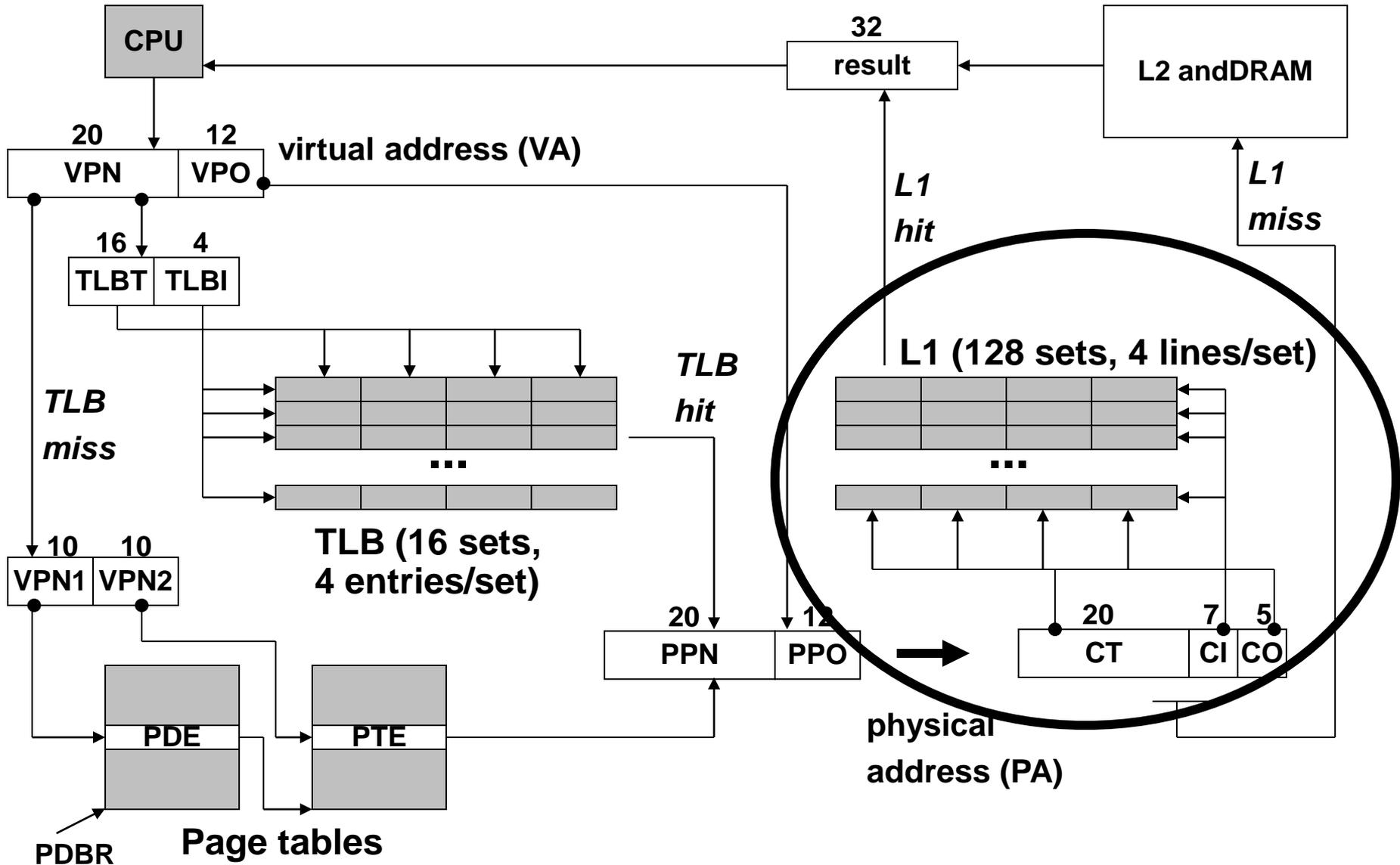
- Case 0/0: page table and page missing.
- MMU Action:
 - page fault exception

Translating with the P6 Page Tables (case 0/0, cont)

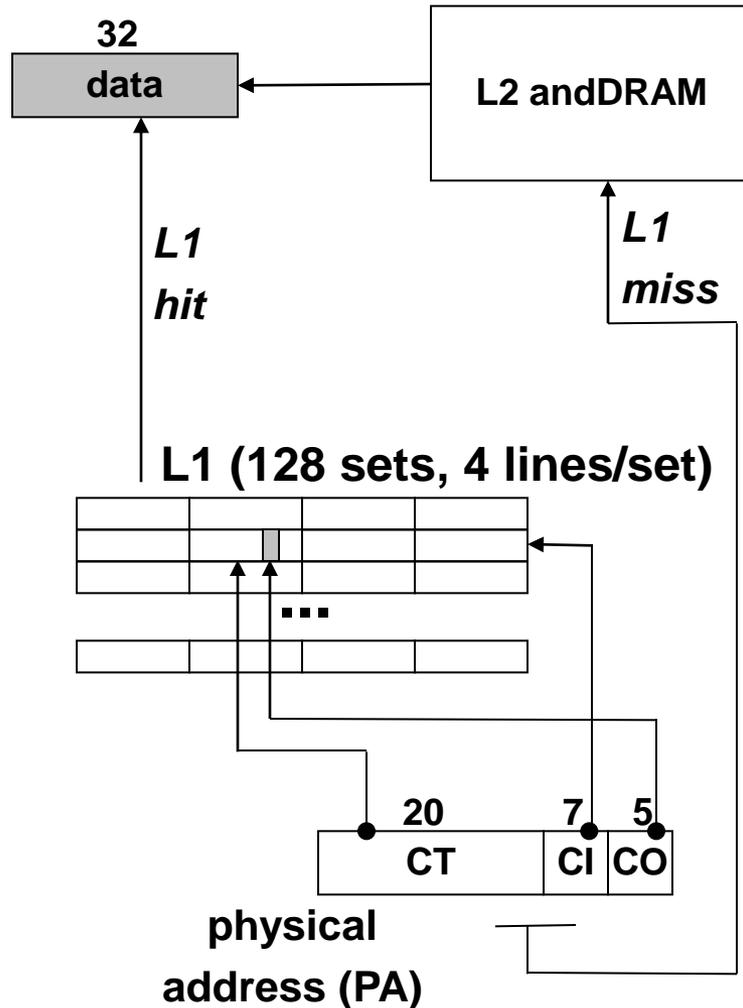


- OS action:
 - swap in page table.
 - restart faulting instruction by returning from handler.
- Like case 1/0 from here on.

P6 L1 Cache Access

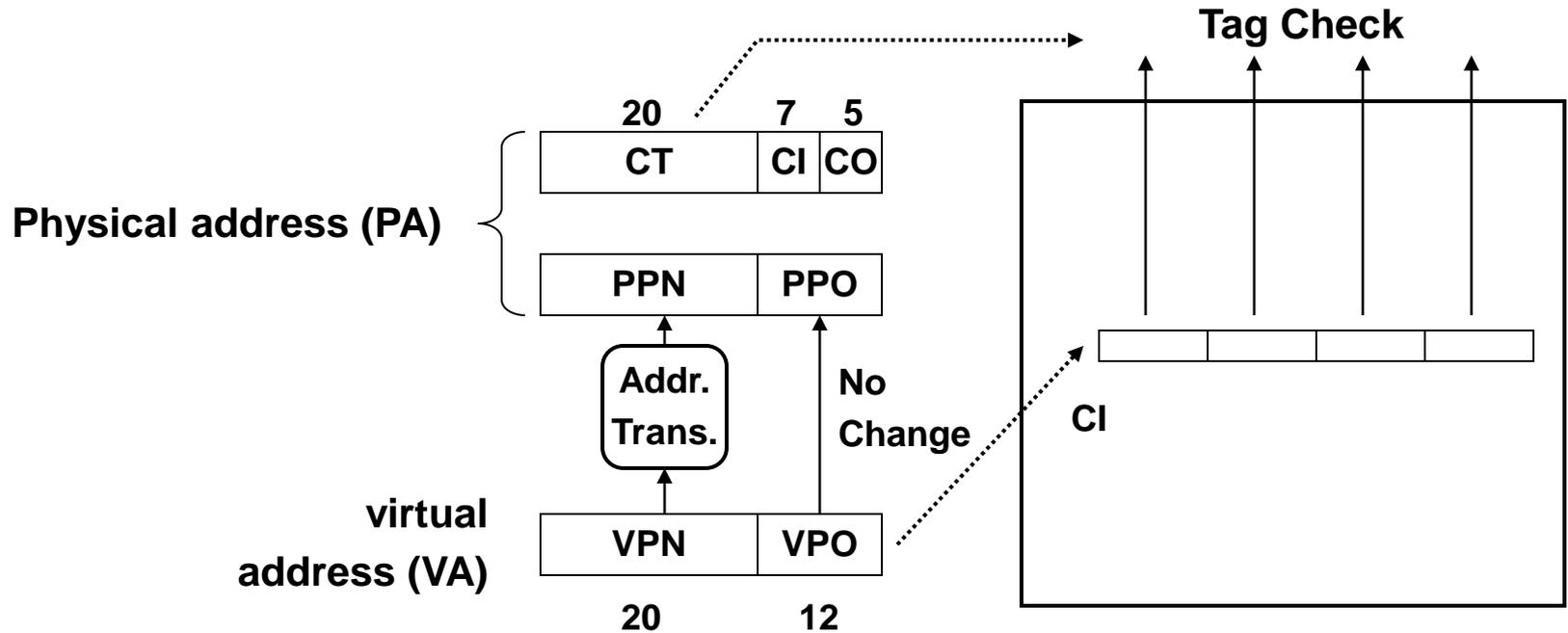


L1 Cache Access



- Partition physical address into CO, CI, and CT.
- Use CT to determine if line containing word at address PA is cached in set CI.
- If no: check L2.
- If yes: extract word at byte offset CO and return to processor.

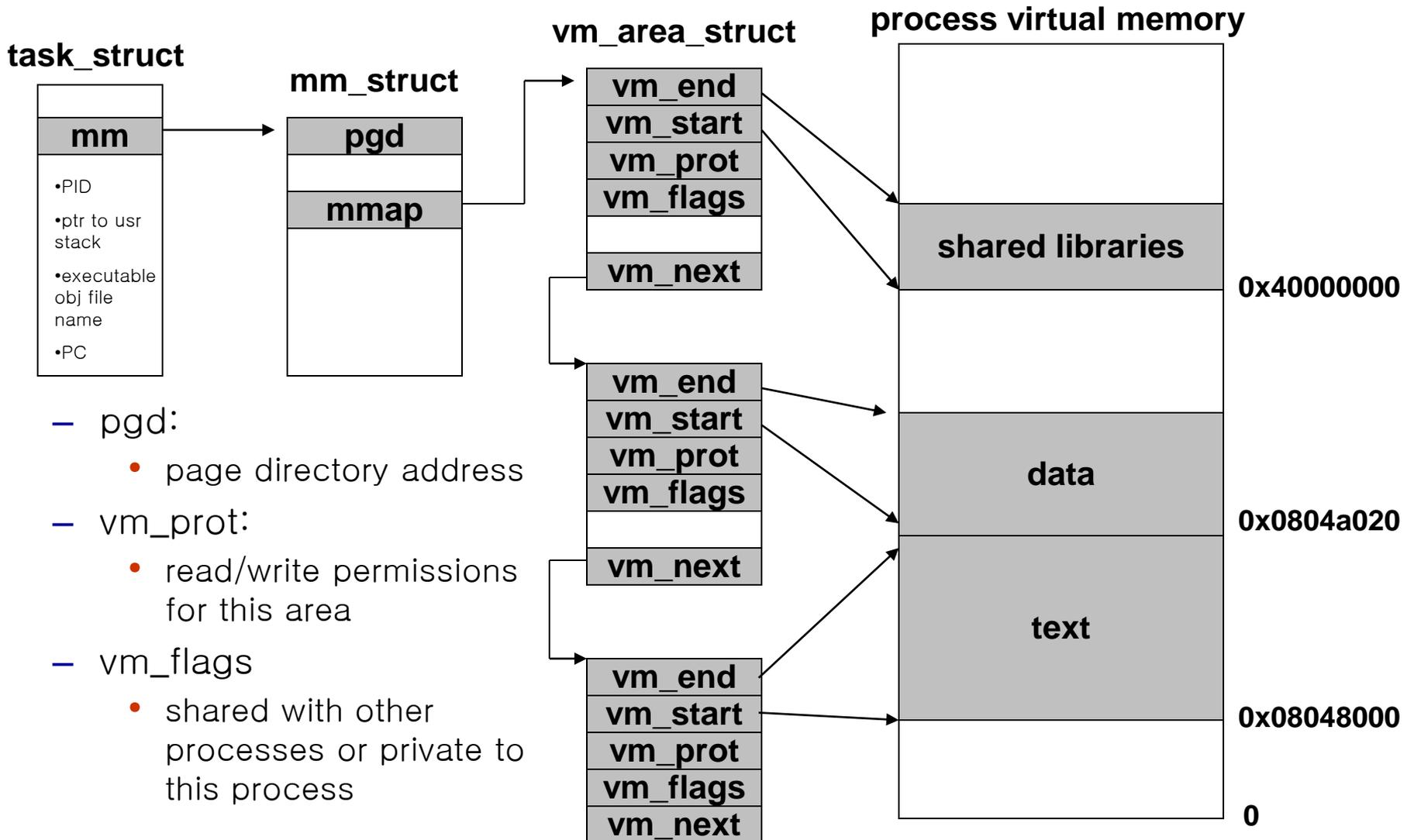
Speeding Up L1 Access



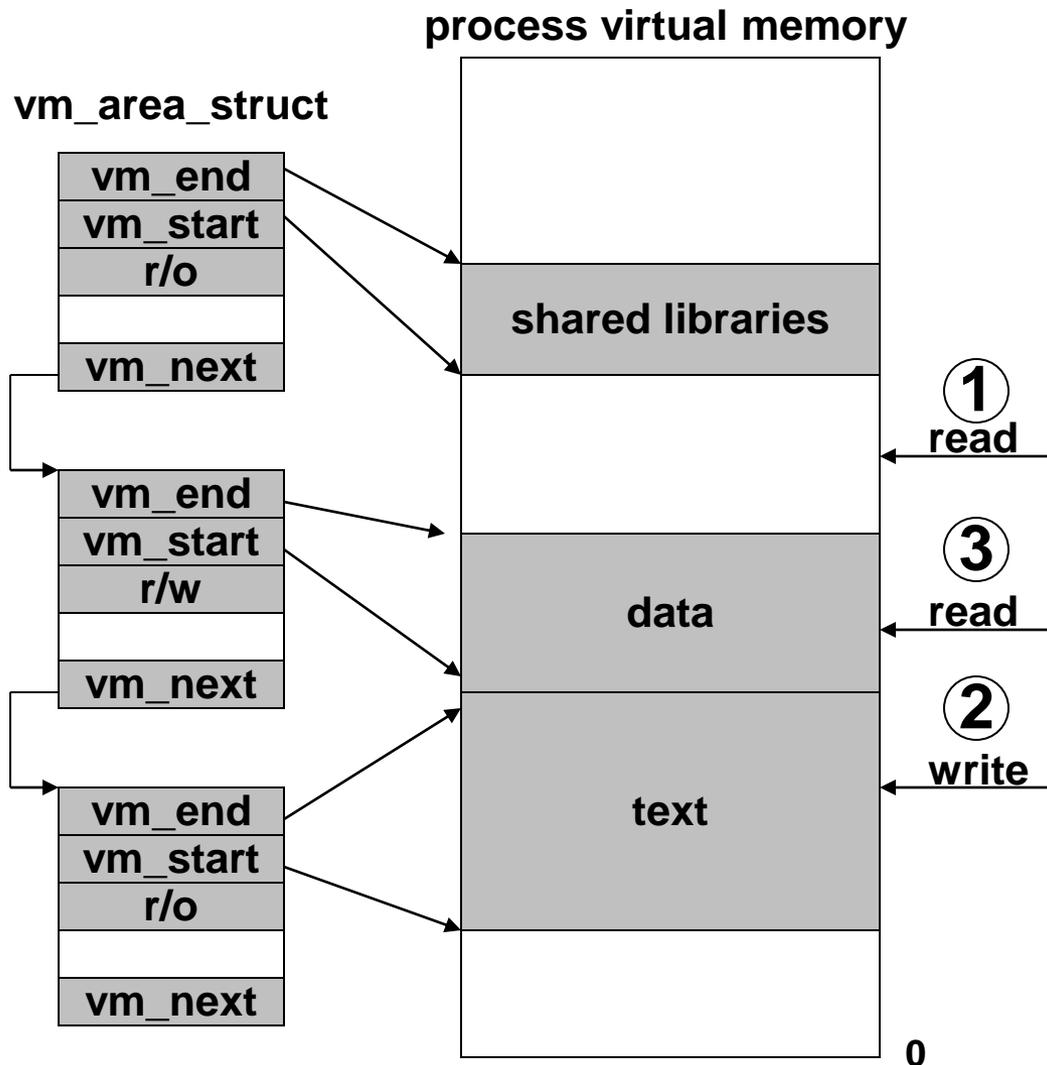
■ Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Then check with CT from physical address
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

Linux Organizes VM as Collection of “Areas”



Linux Page Fault Handling

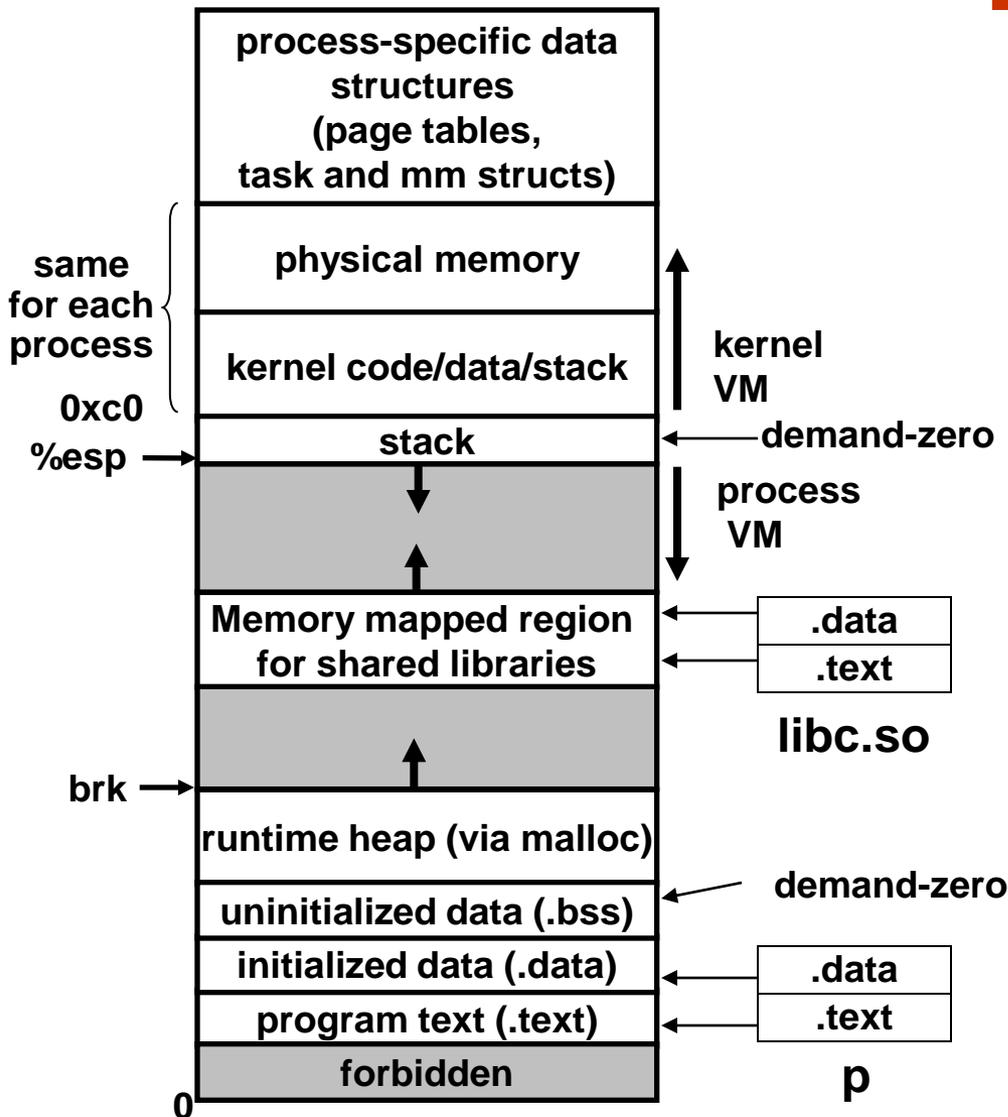


- When Page Hit
 - Protection is checked by hardware's page table retrieval
- When Page Fault, page fault handler checks the following before page-in
 - Is the VA legal?
 - i.e. is it in an area defined by a `vm_area_struct`? (checked by page fault handler)
 - if not then signal segmentation violation (e.g. (1))
 - Is the operation legal?
 - i.e., can the process read/write this area?
 - if not then signal protection violation (e.g., (2))
 - If OK, handle fault
 - e.g., (3)

Memory Mapping

- Creation of new VM *area* done via “memory mapping”
 - create (1) new `vm_area_struct` and (2) page tables for area
 - area can be backed by (i.e., get its initial values from) :
 - regular file on disk (e.g., an executable object file)
 - initial page bytes come from a section of a file
 - nothing (e.g., `bss`)
 - initial page bytes are zeros
 - dirty pages are swapped back and forth between a special swap file.
- Key point: For a new VM area mapping, no virtual pages are copied into physical memory until they are referenced!
 - known as “demand paging”
 - crucial for time and space efficiency

Exec() Revisited



- To run a new program `p` in the current process using `exec()`:
 - free `vm_area_struct`'s and page tables for old areas.
 - create new `vm_area_struct`'s and page tables for new areas.
 - stack, bss, data, text, shared libs.
 - text and data backed by ELF executable object file.
 - bss and stack initialized to zero.
 - set PC to entry point in `.text`
 - Linux will swap in code and data pages as needed.

Fork() Revisited

- To create a new process using `fork()`:
 - make copies of the old process's `mm_struct`, `vm_area_struct`'s, and page tables.
 - at this point the two processes are sharing all of their pages.
 - How to get separate spaces without copying all the virtual pages from one space to another?
 - “copy on write” technique.
 - copy-on-write
 - make pages of writeable areas read-only (in page table entry)
 - flag `vm_area_struct`'s for these areas as private “copy-on-write”.
 - writes by either process to these pages will cause protection faults.
 - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.
 - Net result:
 - copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

User-Level Memory Mapping

- `void *mmap(void *start, int len,`
 - `int prot, int flags, int fd, int offset)`
- map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don't care).
 - `prot`: `MAP_READ`, `MAP_WRITE`
 - `flags`: `MAP_PRIVATE`, `MAP_SHARED`
 - return a pointer to the mapped area.
 - Example: fast file copy
 - useful for applications like Web servers that need to quickly copy files.
 - `mmap` allows file transfers without copying into user space.

mmap() Example: Fast File Copy

```
■ #include <unistd.h>
■ #include <sys/mman.h>
■ #include <sys/types.h>
■ #include <sys/stat.h>
■ #include <fcntl.h>

■ /*
■  * mmap.c - a program that uses mma
■  * p
■  * to copy itself to stdout
■  */
■
```

```
■ int main() {
■     struct stat stat;
■     int i, fd, size;
■     char *bufp;

■     /* open the file & get its size
■     */
■     fd = open("./mmap.c", O_RDONLY)
■     ;
■     fstat(fd, &stat);
■     size = stat.st_size;
■     /* map the file to a new VM area */
■     bufp = mmap(0, size, PROT_READ,
■                 MAP_PRIVATE, fd, 0);

■     /* write the VM area to stdout */
■     write(1, bufp, size);
■ }
```

- read from memory (i.e., bufp) = read from the mapped file
- Kernel (i.e., page fault handler) will actually read the data

Memory System Summary

- Cache Memory
 - Purely a speed-up technique
 - Behavior invisible to application programmer and OS
 - Implemented totally in hardware
- Virtual Memory
 - Supports many OS-related functions
 - Process creation
 - Initial
 - Forking children
 - Task switching
 - Protection
 - Combination of hardware & software implementation
 - Software management of tables, allocations
 - Hardware access of tables
 - Hardware caching of table entries (TLB)