Paging Basics

02/14/2023

Professor Amanda Bienz

Textbook pages 360-376

Concept of Paging

- Permits a process's physical address space to be non-contiguous
- Paging splits up address space into fixed-size unit called a page
  - Segmentation : OS and hardware map a relatively small number of variable sized segments (code, stack, heap, etc)
  - Paging : OS and hardware map a large number of fixed-size segments
- Process's virtual address space and physical memory are divided into pages
- Segment table per CPU become a page table per process

Basic Method

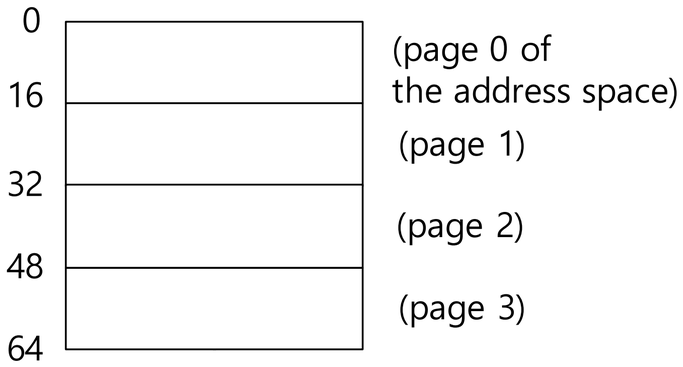
- Break physical memory into fixed-sized blocks called **frames**
- Break logical memory into blocks of the same size called **pages**
- Before executing a process, load its pages into available frames
- A process can have a logical 64-bit address space even though the system has less than 2^64 bytes of physical memory

Advantages of Paging

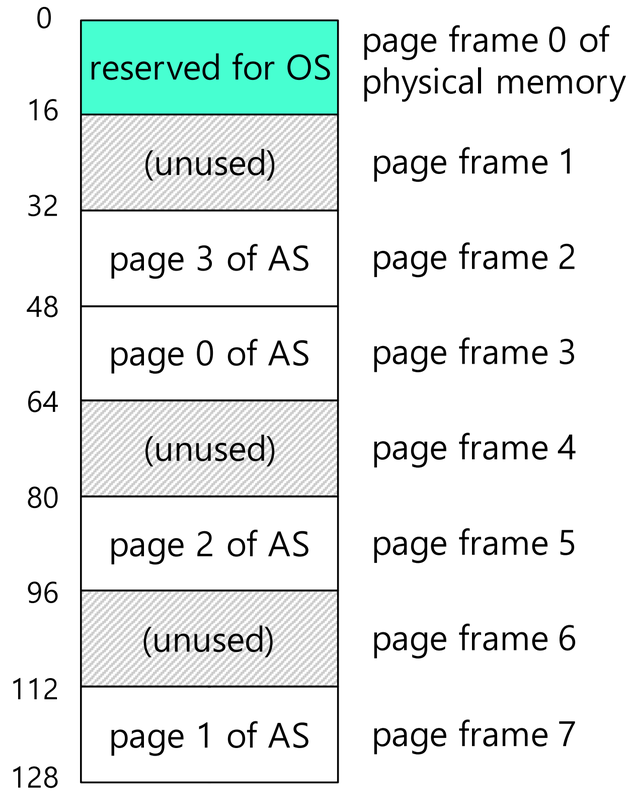
- **Flexibility :** supporting abstraction of address space effectively
  - Don’t need assumption how heap and stack grow and are used
- **Simplicity :** ease of free-space management
  - Page in address space and page frame are same size
  - Easy to allocate and keep a free list
- **Whence fragmentation?**

Example : Simple Paging

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 byte pages



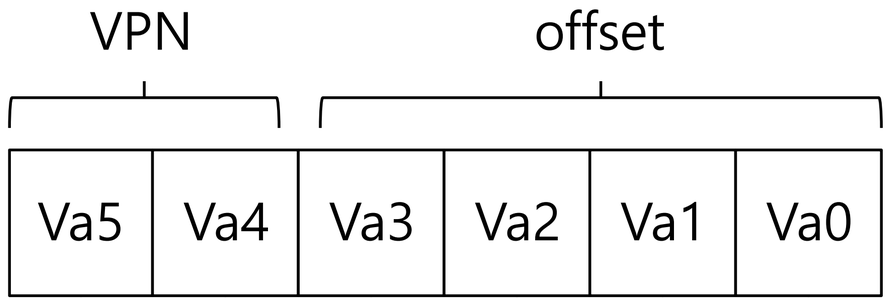
A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

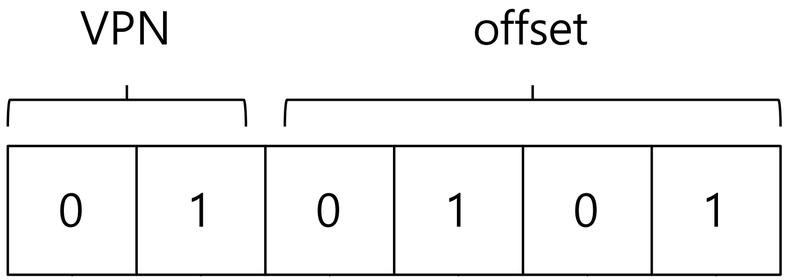
Address Translation

- Two components in virtual address
  - **VPN** : virtual page number
  - **Offset** : offset within the page



  - Page number used as index into per-process **page table**

- Example : virtual address 21 in 64-byte address space



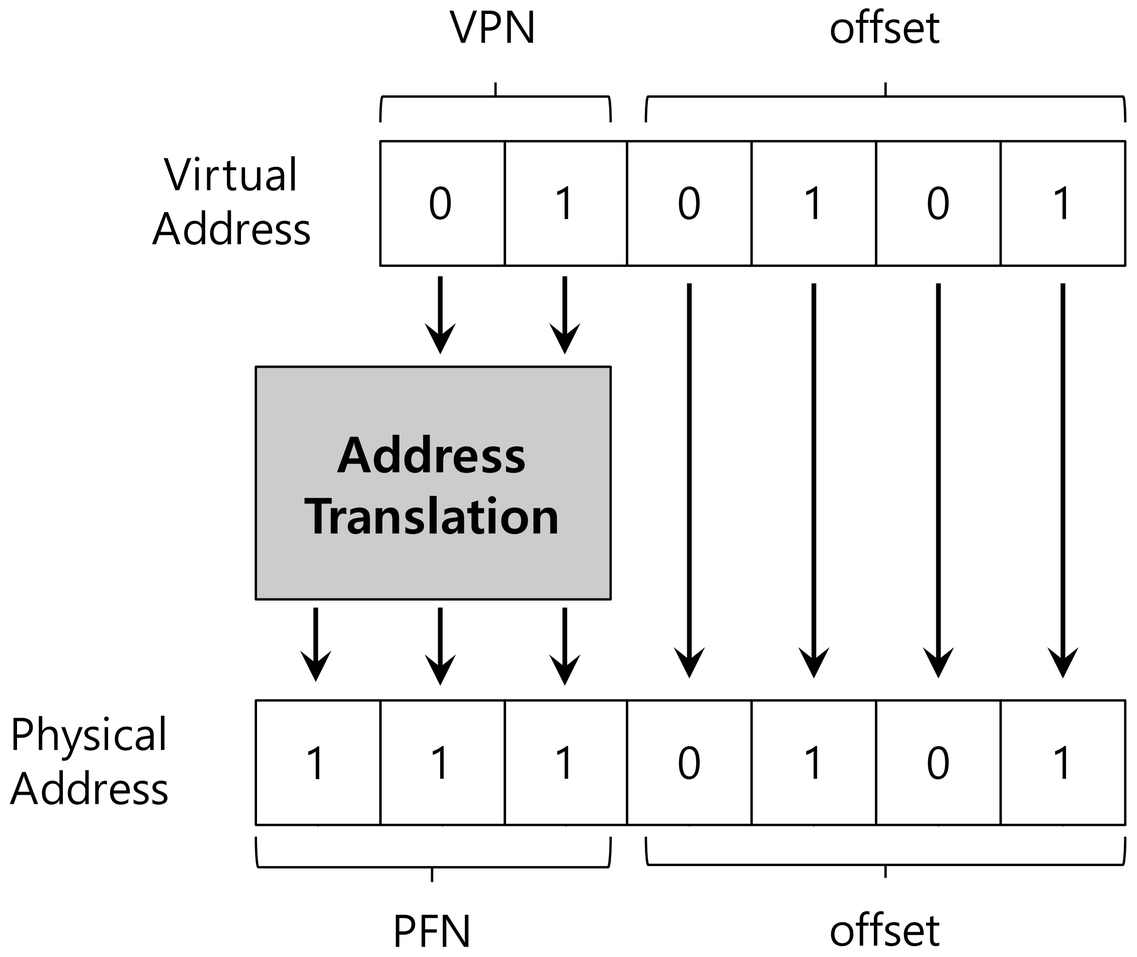
Address Translation

1. Extract page number p and use it as an index into the page table
2. Extract corresponding frame number f from the page table
3. Replace page number p in logical address with frame number f

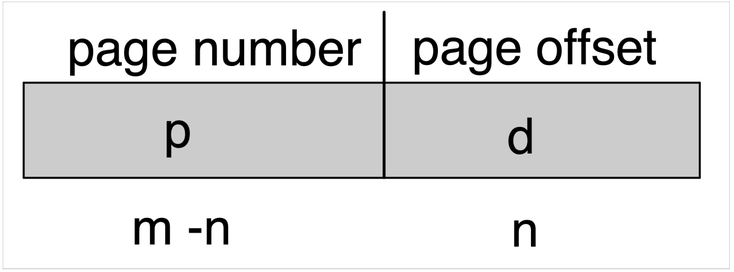
Page size is a power of 2 (typically between 4KB and 1GB). Why?

Example : Address Translation

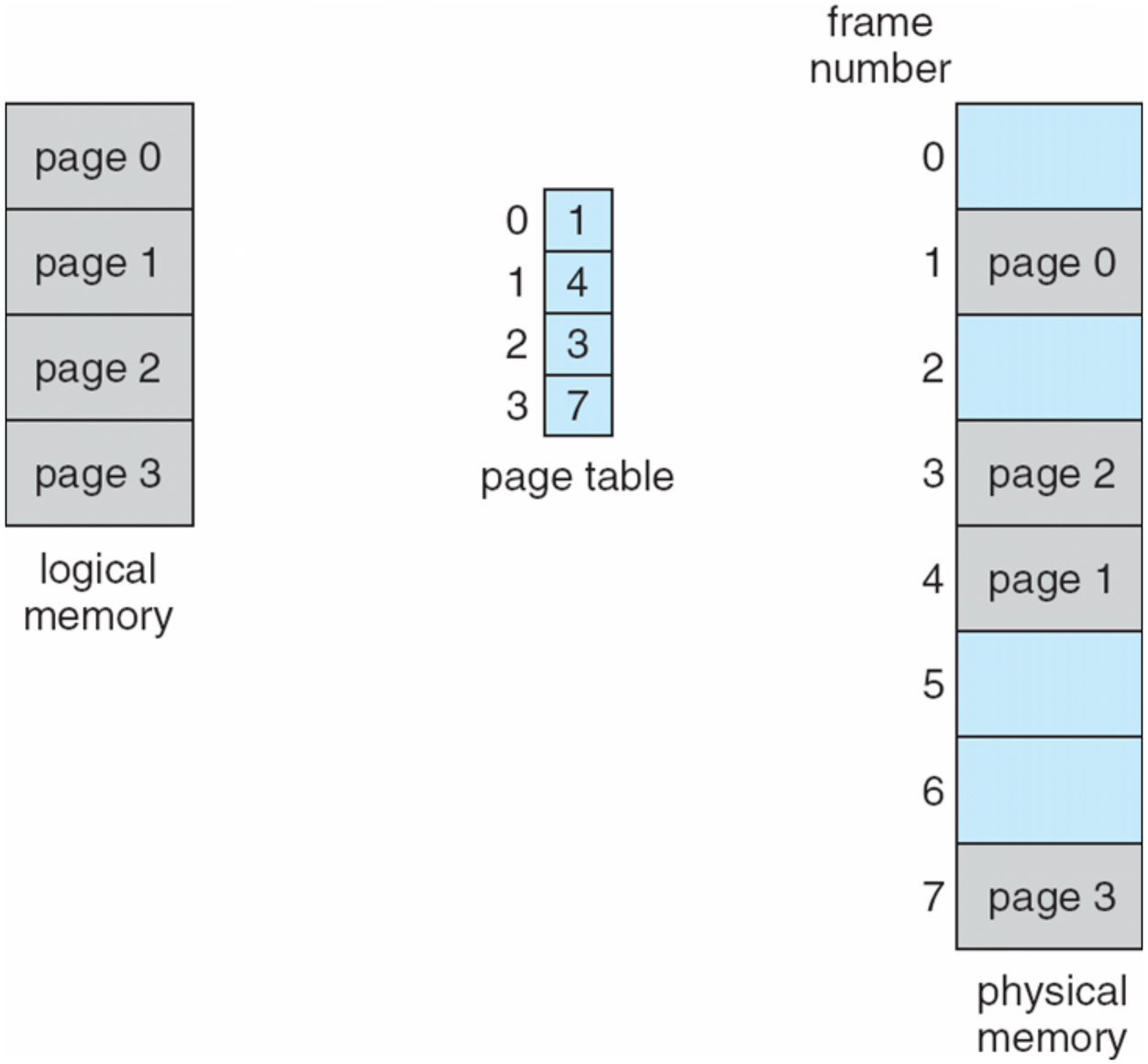
- The virtual address 21 in 64-byte address space



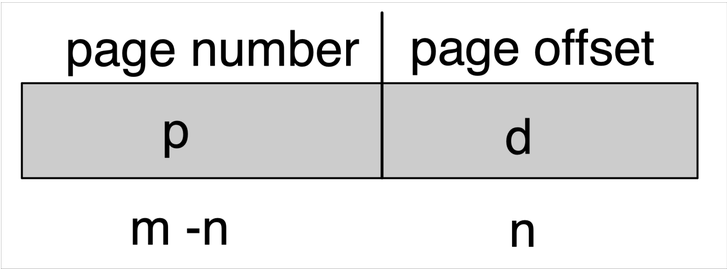
Paging Hardware



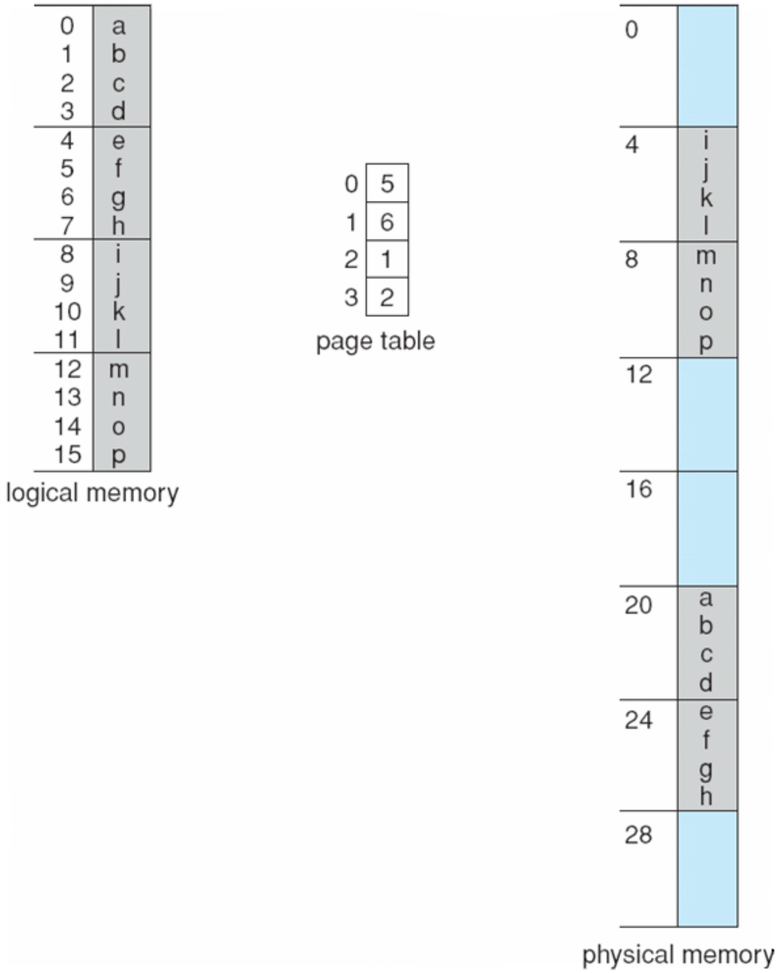
Paging Model of Logical/Physical Memory



Paging Example



- Logical address : n = 2, m = 4
- Using a page size of 4 bytes and a physical memory of 32 bytes
- How many pages? How many frames?



Fragmentation

- Does paging increase **external** fragmentation over other approaches?
- Does paging increase **internal** fragmentation over other approaches?

Internal Fragmentation Example

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- How many pages are needed for this process?
- What is the internal fragmentation?

Paging - Internal Fragmentation

- For frame size **f**, what is the *worst* case internal fragmentation?

- For frame size **f**, what is the *average* case internal fragmentation?

Paging - Internal Fragmentation

- For frame size **f**, what is the *worst* case internal fragmentation?
  f - 1 bytes

- For frame size **f**, what is the *average* case internal fragmentation?
  1/2 f

Frame Size

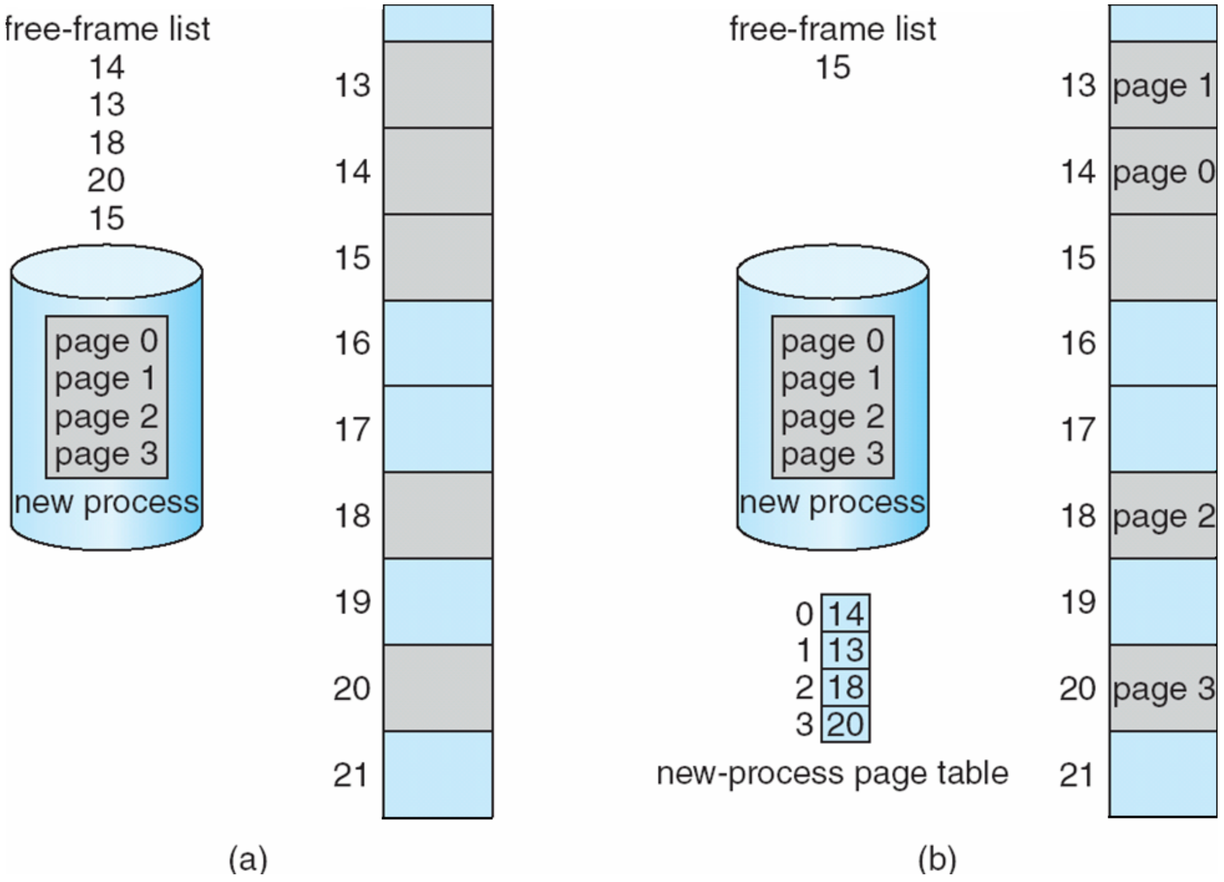
- What are the pros and cons of small frame sizes?

Frame Size

- What are the pros and cons of small frame sizes?
- Pros :
  - Less internal fragmentation
- Cons :
  - Each page table entry takes memory to track
- Many systems support two page sizes (huge pages)

Free Frames

- Like free lists, but list of available frames



Before allocation

After allocation

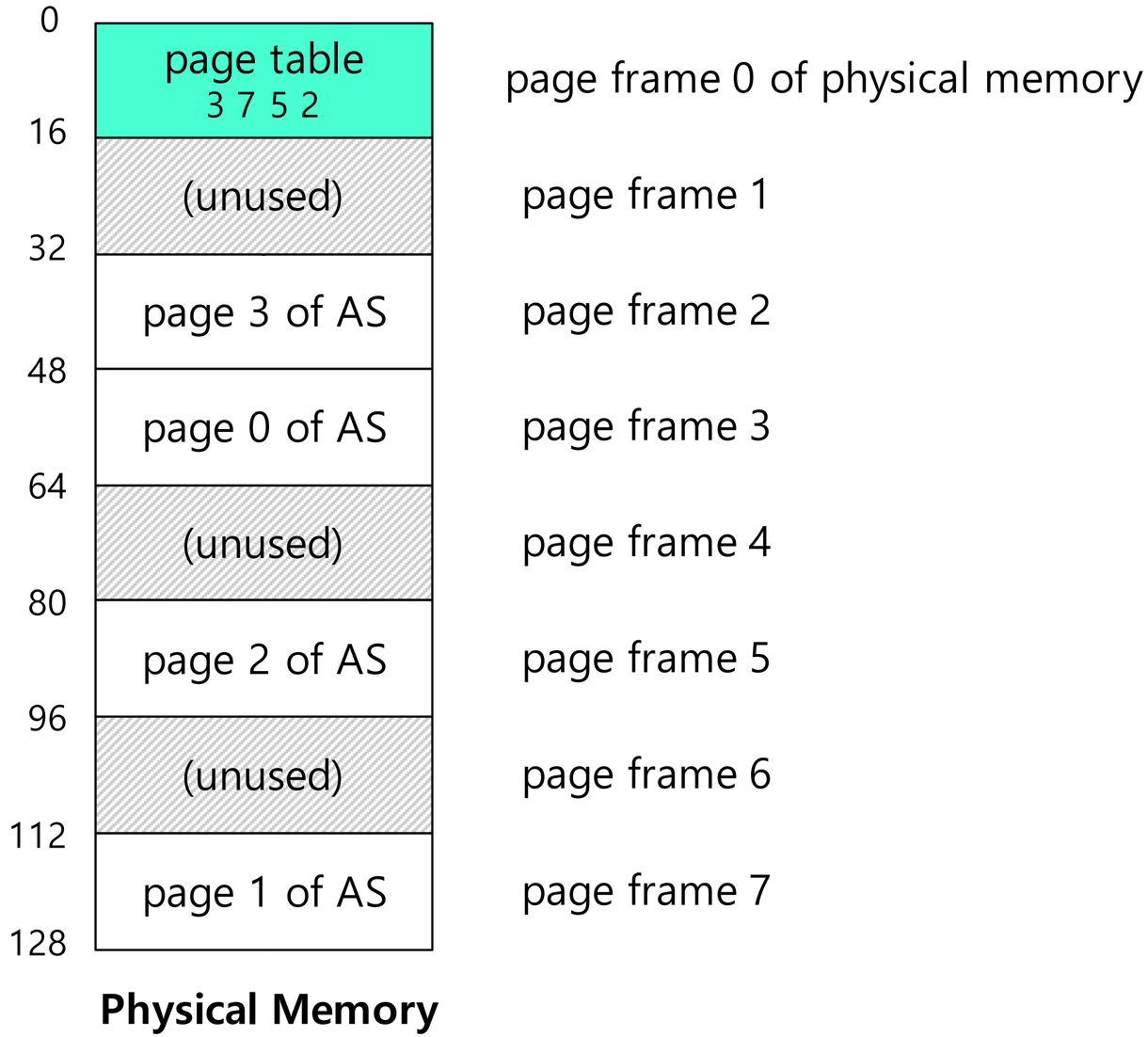
Implementation of Page Table

- Page table kept in main memory
  - Page-table base register (PTBR) : points to the page table
  - Page-table length register (PTLR) : indicates size of page table
- In this scheme, every data / instruction access requires two memory accesses
  - One for page table, one for data / instruction

Where are Page Tables Stored?

- Page tables can get awfully large
  - 32-bit address space with 4KB pages, 20 bits for VPN
    - 4MB = 2^(20) entries * 4 bytes per page table entry
- Segment tables were generally on the CPU (small set of registers)
- Page tables for each process are stored in memory
- Processor has an OS-visible register that stores address of current process's page table

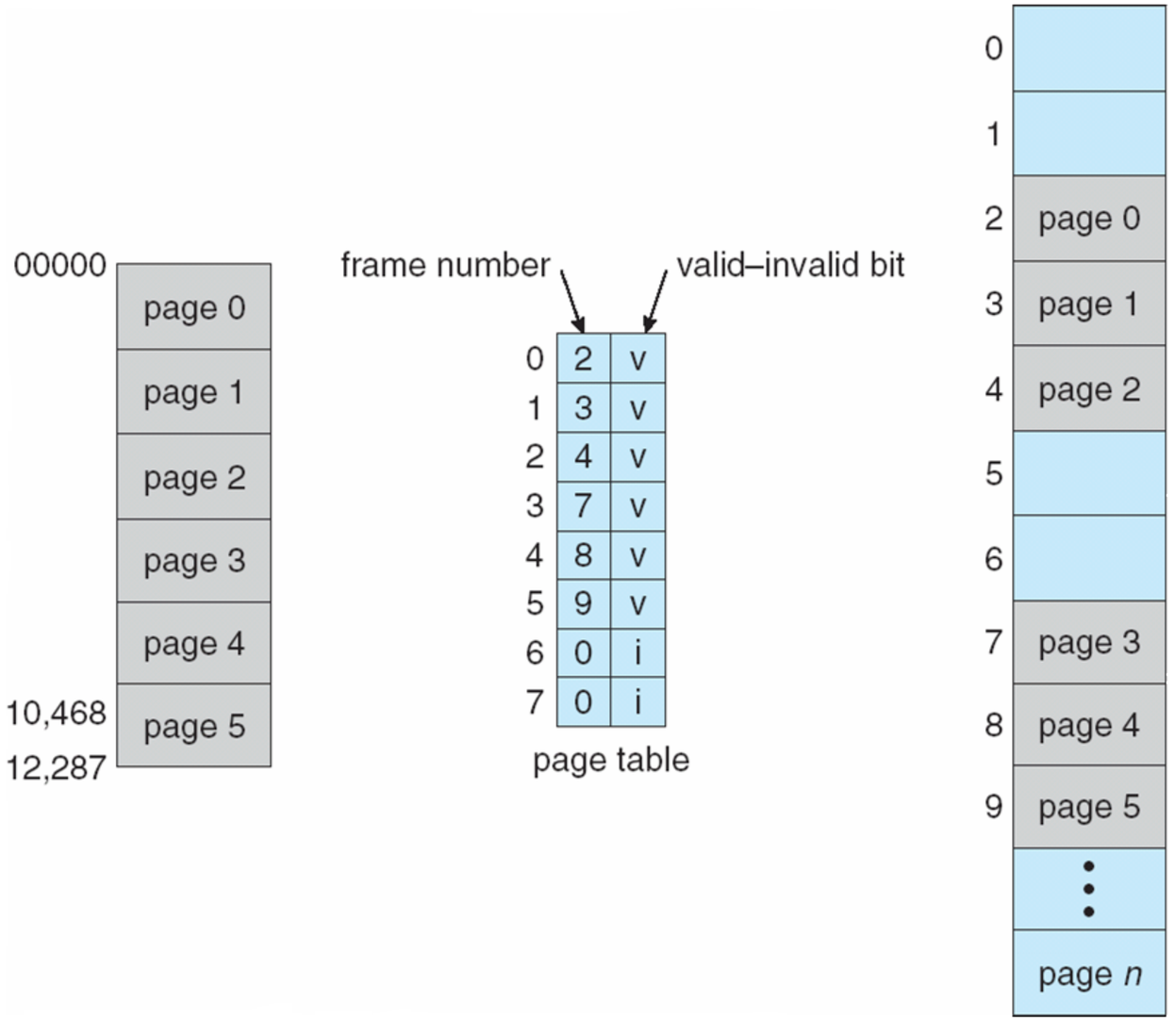
Example : Page Table in Kernel Physical Memory



What is in the Page Table?

- Page table : data structure that is used to map virtual address to physical address
  - Simplest form : a linear page table, an array
  - Lots of ways to structure page tables
- OS indexes array by VPN and looks up page-table entry
- Like segment table, can put other things in each page table entry

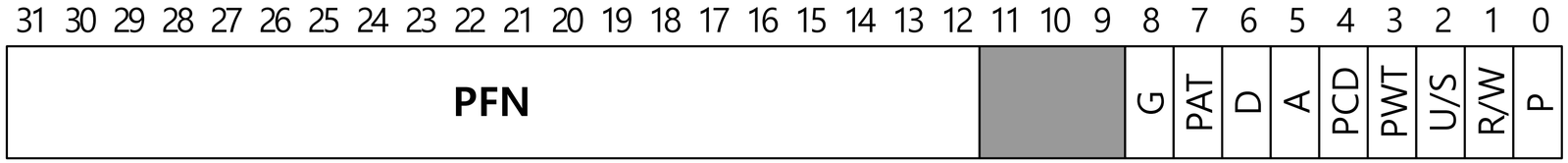
Valid (v) or Invalid (i) Bit in Page Table



Common Flags of Page Table Entry

- Valid bit : indicates whether particular translation is valid
- Protection bit : indicates whether page could be read from, written to, or executed from
- Present bit : indicates whether this page is in physical memory or on disk (swapped out)
- Dirty bit : indicates whether page has been modified since it was brought into memory
- Reference bit (accessed bit) : indicates that a page has been accessed
- Supervisor bit : distinguish pages of kernel memory from user process

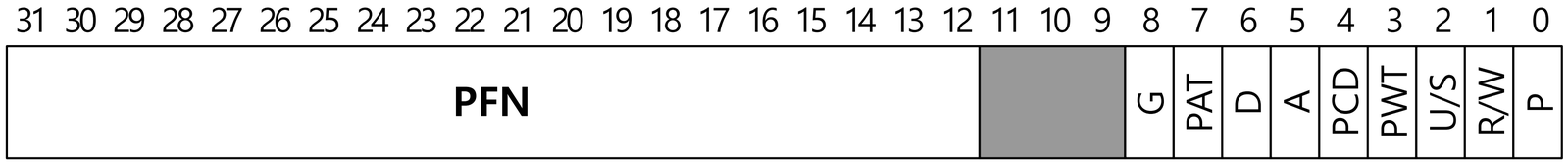
Example : x86 Page Table Entry



An x86 Page Table Entry(PTE)

- **P :** Present bit : indicates whether this page is in physical memory or on disk (swapped out)
- **R / W :** read/write bit
- **U / S :** Supervisor bit : distinguish pages of kernel memory from user process
- **D :** Dirty bit : indicates whiter page has been modified since it was brought into memory
- **A :** Accessed bit : indicates that a page has been accessed
- **PFN :** Page Frame Number

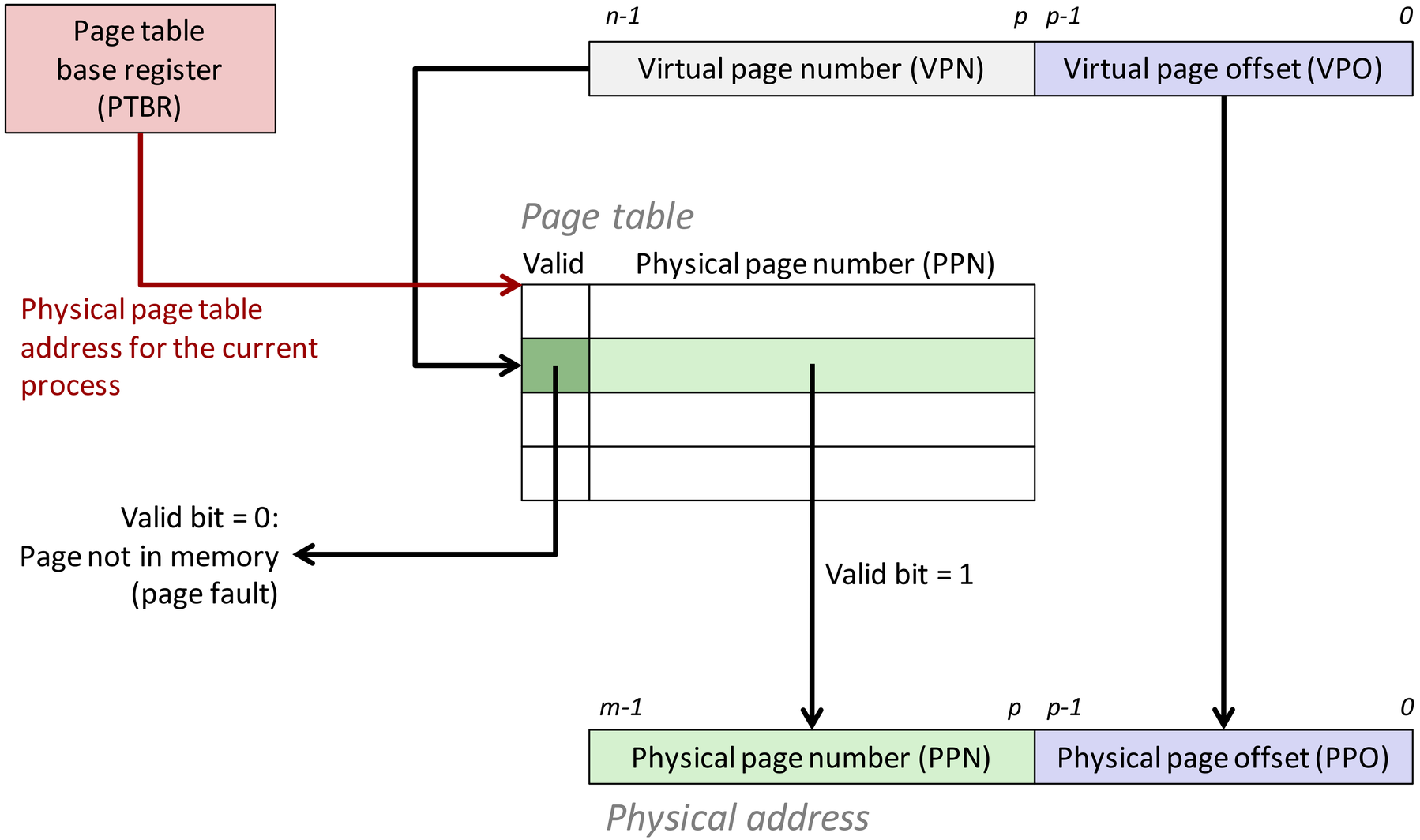
Example : x86 Page Table Entry



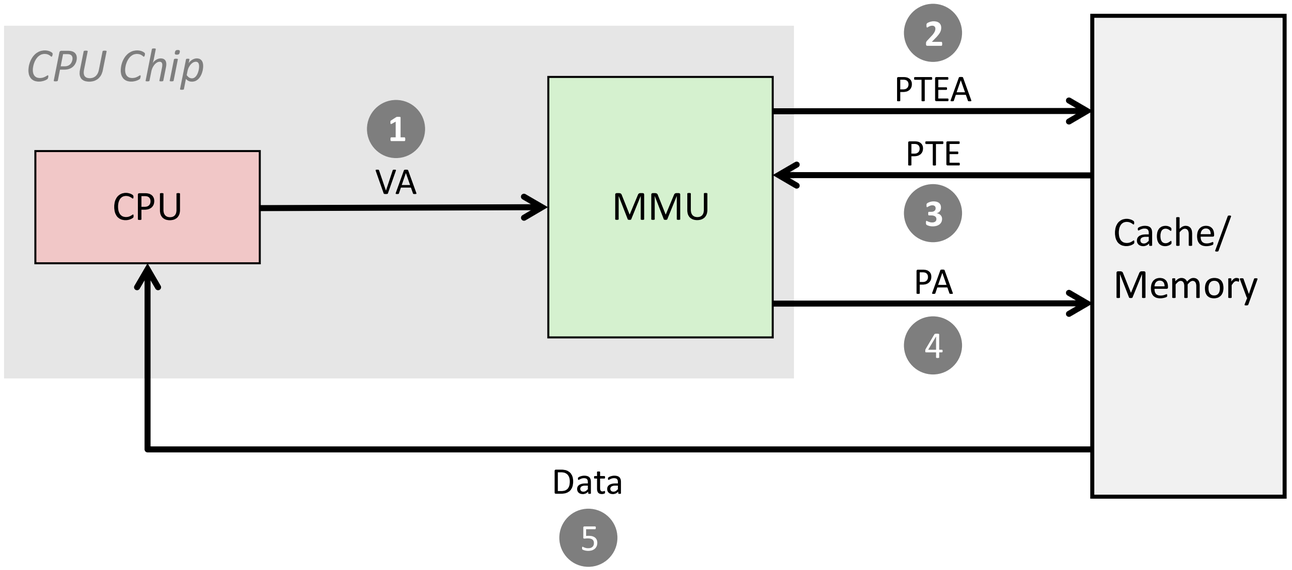
An x86 Page Table Entry(PTE)

- **G :** Global page flag - marks page as global when set. Prevents frequently used pages (OS kernels) from being flushed
- **PAT :** Page Attribute Table - holds memory type for pages. Only supported in some processors
- **PCD :** Page-level cache disable flag - controls movement of individual pages. If set, caching of associated page is prevented
- **PWT :** Page-level write through flag - controls write-through or write-back caching policy of individual pages or page tables (1: write-through, 2: write-back)

Address Translation with a Page Table



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 cache/memory

5) Cache/memory sends data word to processor

Paging : Too Slow

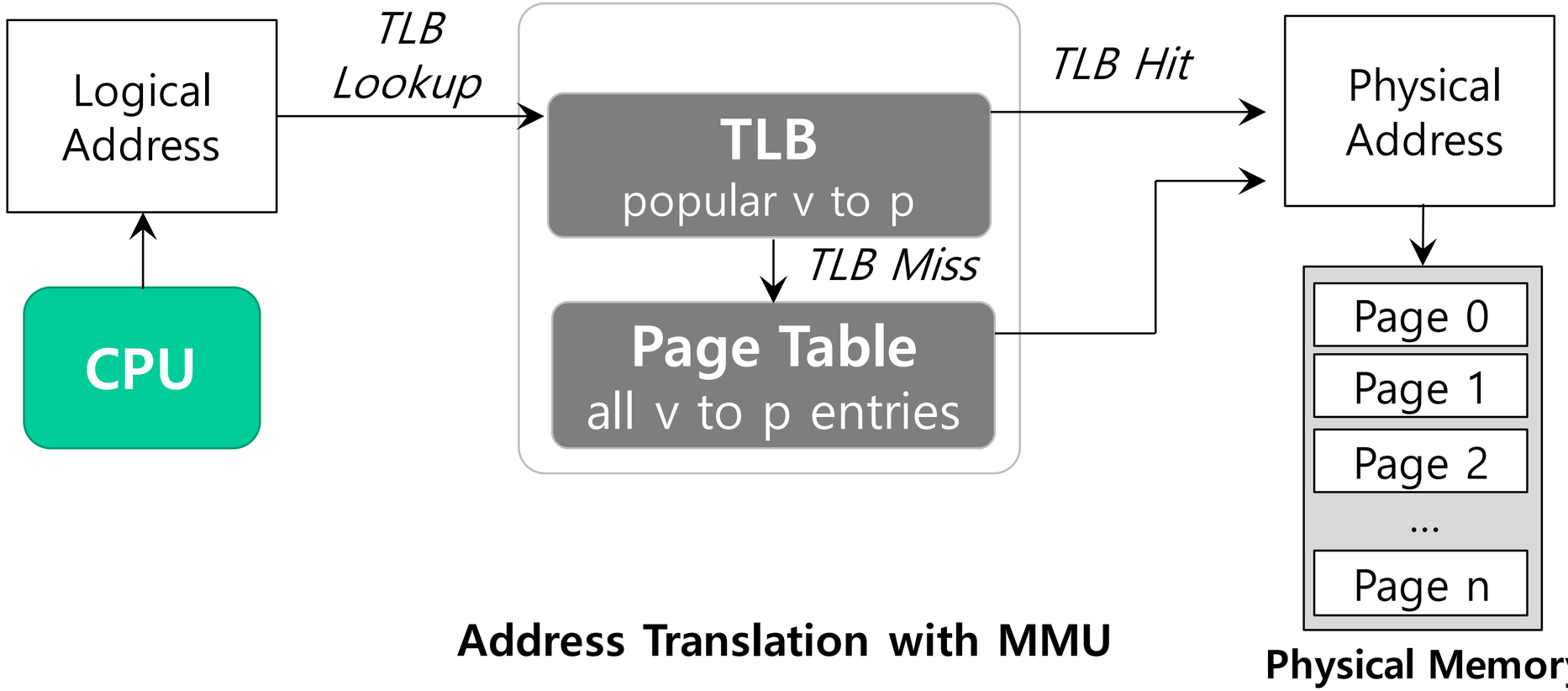
- Page tables are too big to fit on chip
- To find a location of the desired page table entry (PTE), starting location of page table is needed
- For every memory reference, paging requires the OS to perform (at least) one extra memory reference.
  - Look up starting location of page table
- Two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLBs)** - also called associative memory

Speeding up Translation with TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution :** Translation Lookaside Buffer (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages (64 to 1,024)
  - On TLB miss, value loaded into TLB for faster access next time
    - Some entries can be wired down for permanent fast access

TLB : Translation Lookaside Buffer

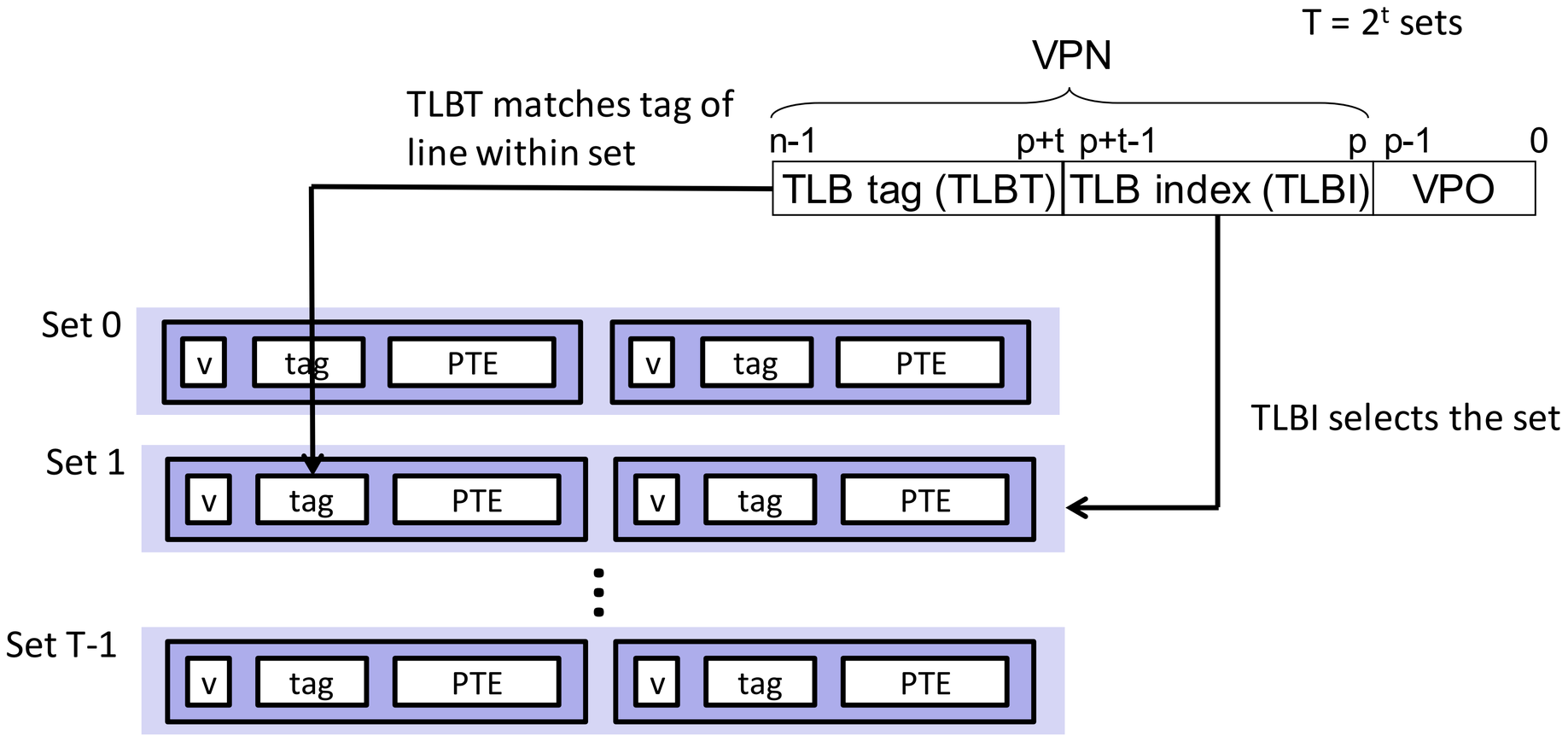
- Part of chip's memory management unit (MMU)
- Hardware cache of popular virtual-to-physical address translation



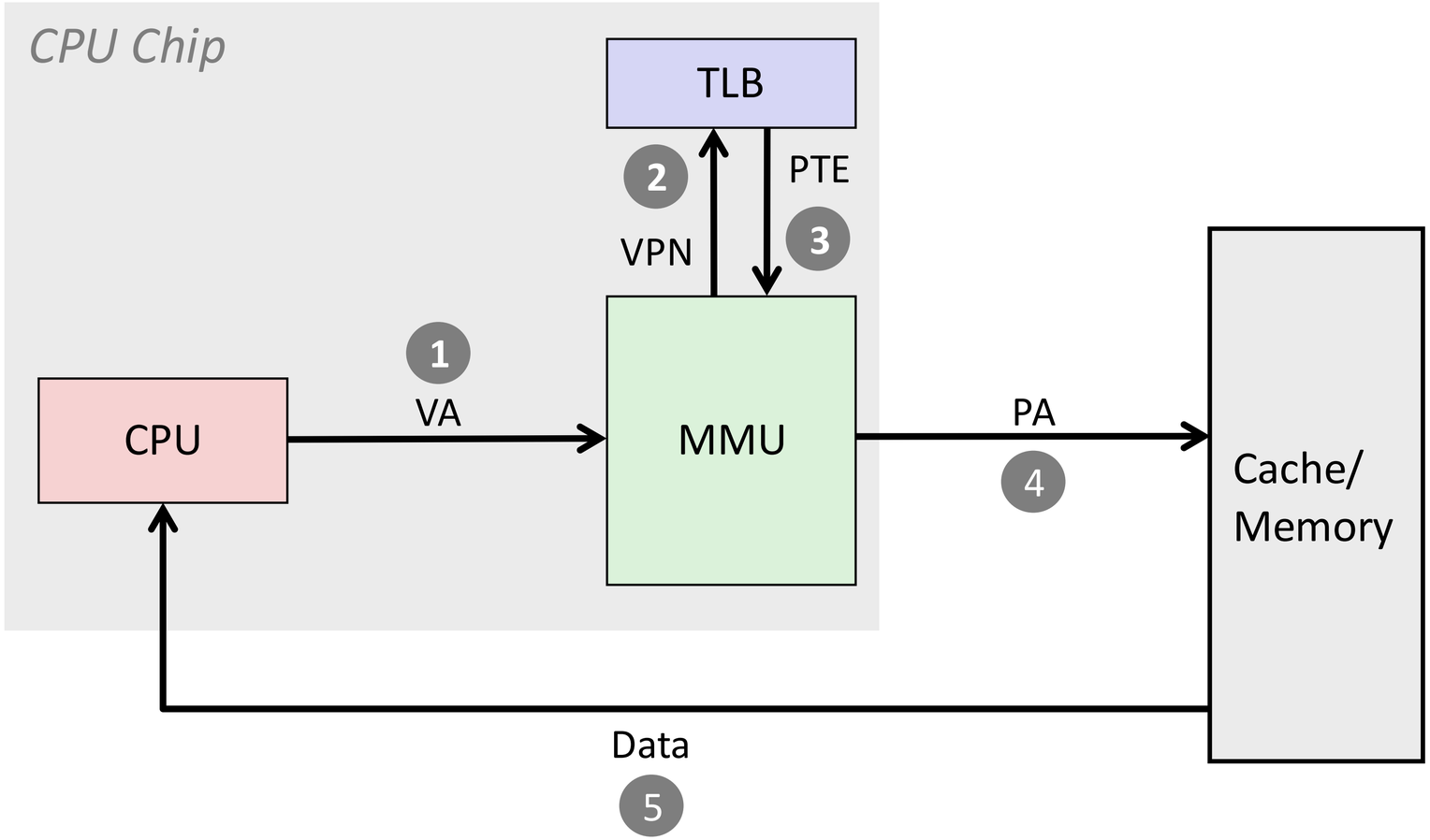
Address Translation with MMU

Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB

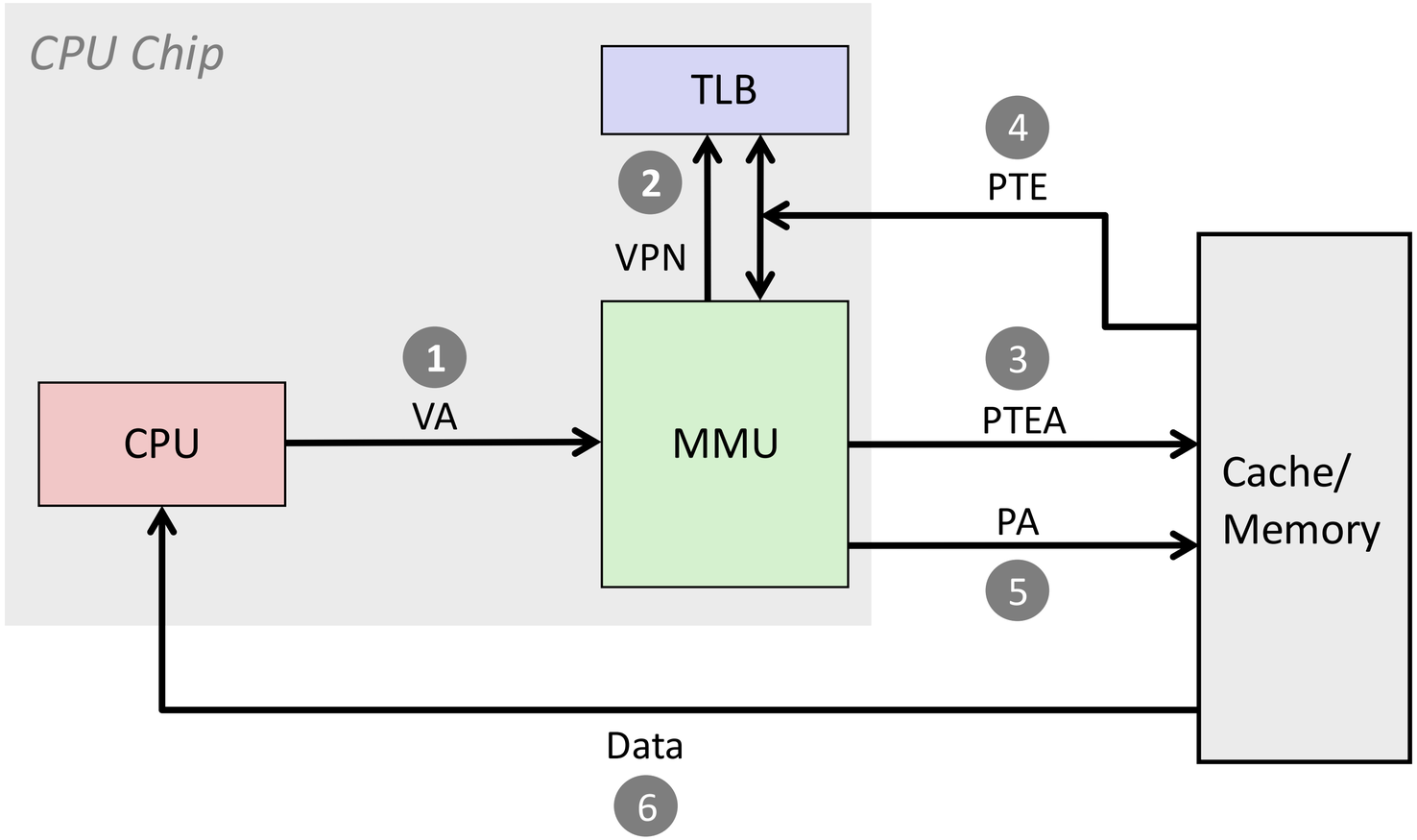


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

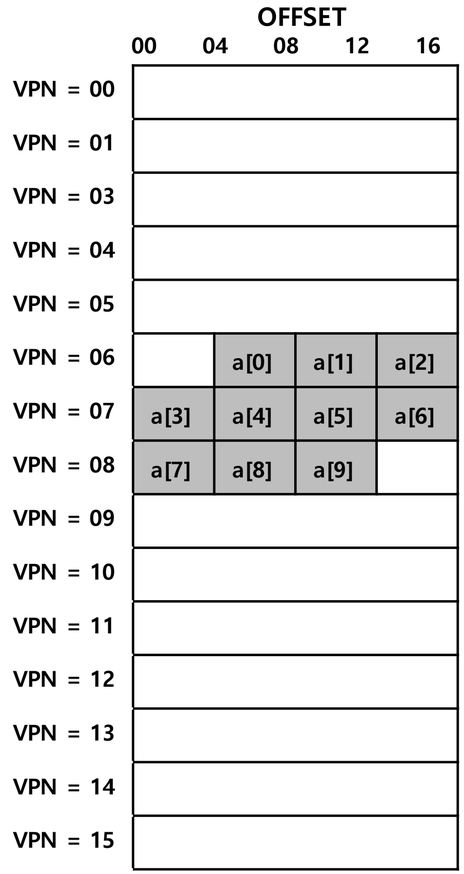


A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

Example : Accessing an Array

- How a TLB can improve its performance




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

The TLB improves performance due to spatial locality

3 misses and 7 hits.
Thus TLB hit rate is 70%.

Effective Access Time

- Hit ratio - percentage of times that page number is found in TLB
- 80% hit ratio means find desired page number in TLB 80% of time
- Suppose it takes 10 nanoseconds to access memory
  - If we find desired page in TLB then a mapped-memory access takes 10ns
  - Otherwise, we need two memory accesses so 20ns
  - **Effective Access Time (EAT):** .80*10 + 0.20*20 = 12ns, implying 20% slowdown in access time
  - Consider more realistic hit ratio of 99% : 0.99*10 + 0.01*20 = 10.1ns, implying only 1% slowdown in access time

Memory Protection

- Memory protection implemented by associated protection bit with each frame to indicate if read-only or read-write

Who Handles TLB Miss?

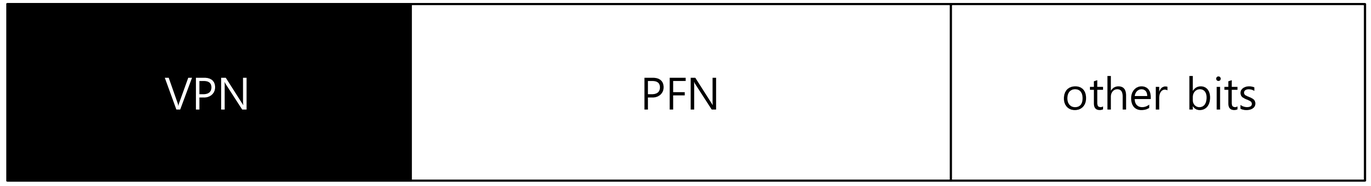
- Option 1 : **Hardware-managed TLB**(x86, ARM)
  - Hardware has to know exactly where the page tables are located in memory
  - Hardware would ‘walk’ the page table, find the correct page-table entry and extract the desired translation, update, and retry instruction
  - Hardware specifies the exact format of the page table

Who Handles TLB Miss?

- Option 2 : **Software-managed TLB** (MIPS, some others)
  - On a TLB miss, hardware raises exception (trap handler)
    - **Trap handler is code** within OS that is written with the express purpose of handling TLB miss
  - Allows for much wider range of page table organizations

TLB Entry

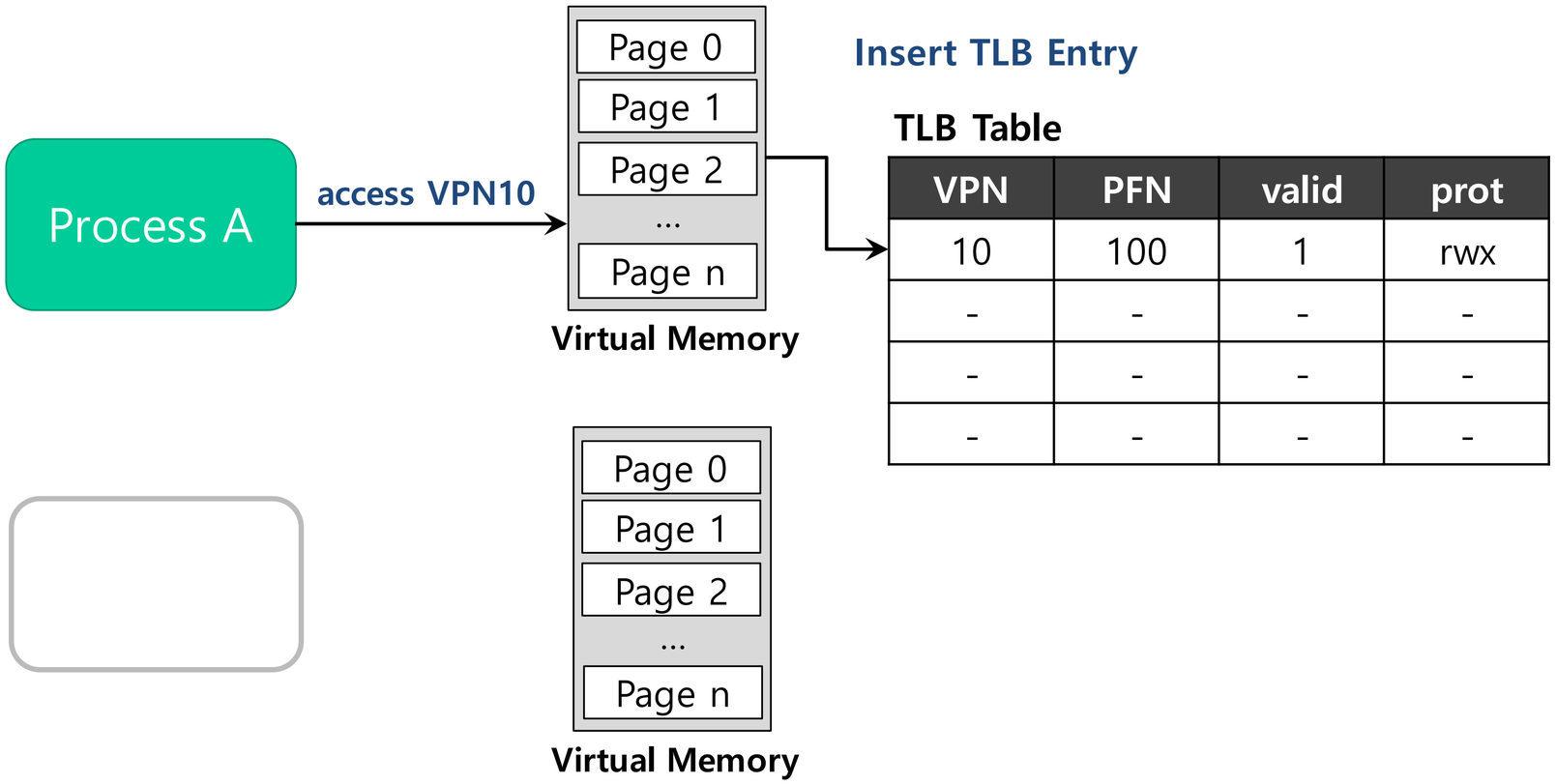
- TLB is generally a small fully associative cache
  - Typical TLB might have 32, 64, or 128 entries
  - Hardware search the entire TLB in parallel to find desired translation
  - Other bits : valid bits, protection bits, address-space identifier, dirty bit



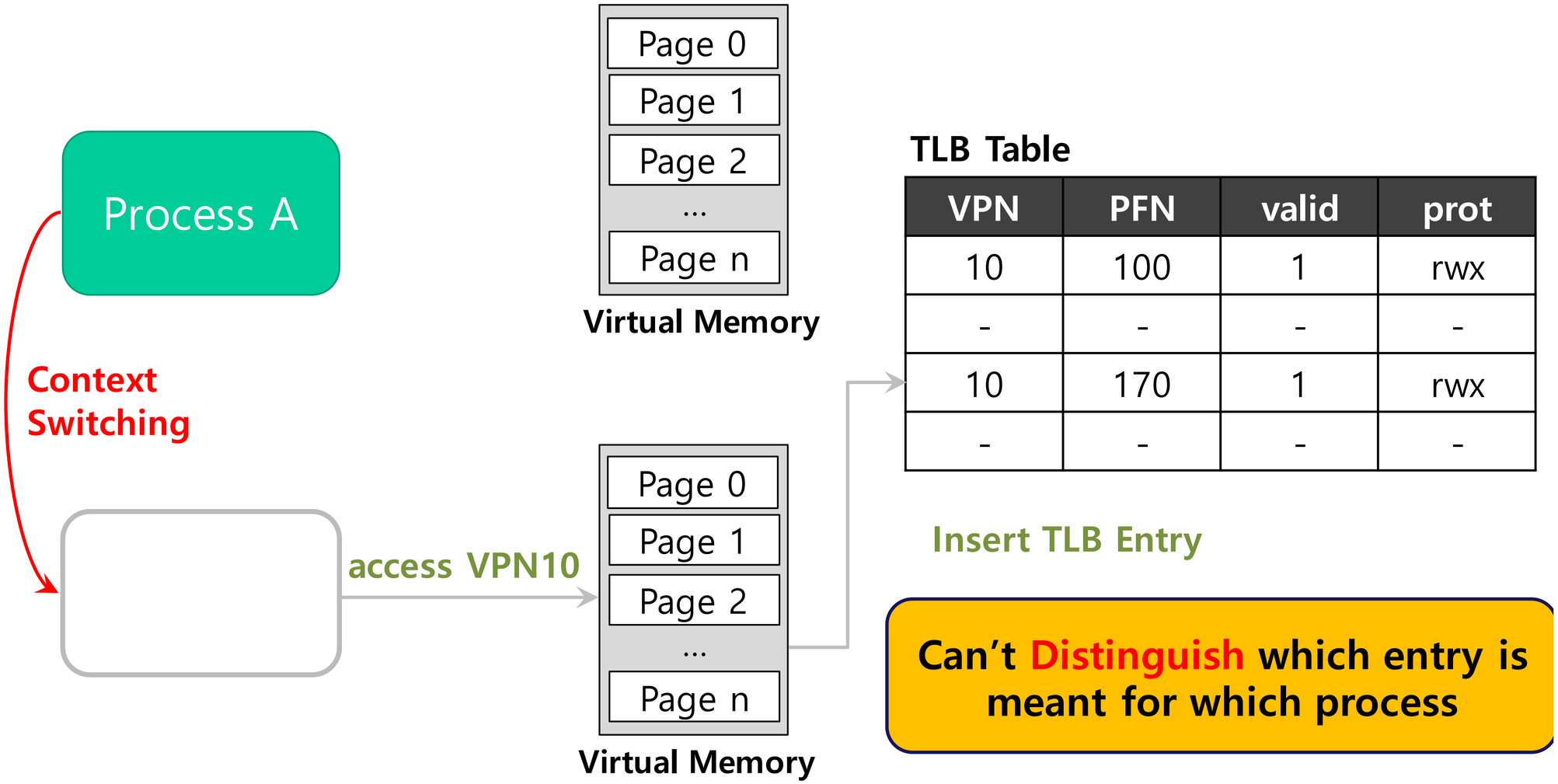
Typical TLB entry look like this

TLB Issue : Context Switching and Shared TLB

- TLB is hardware structure shared by all processes

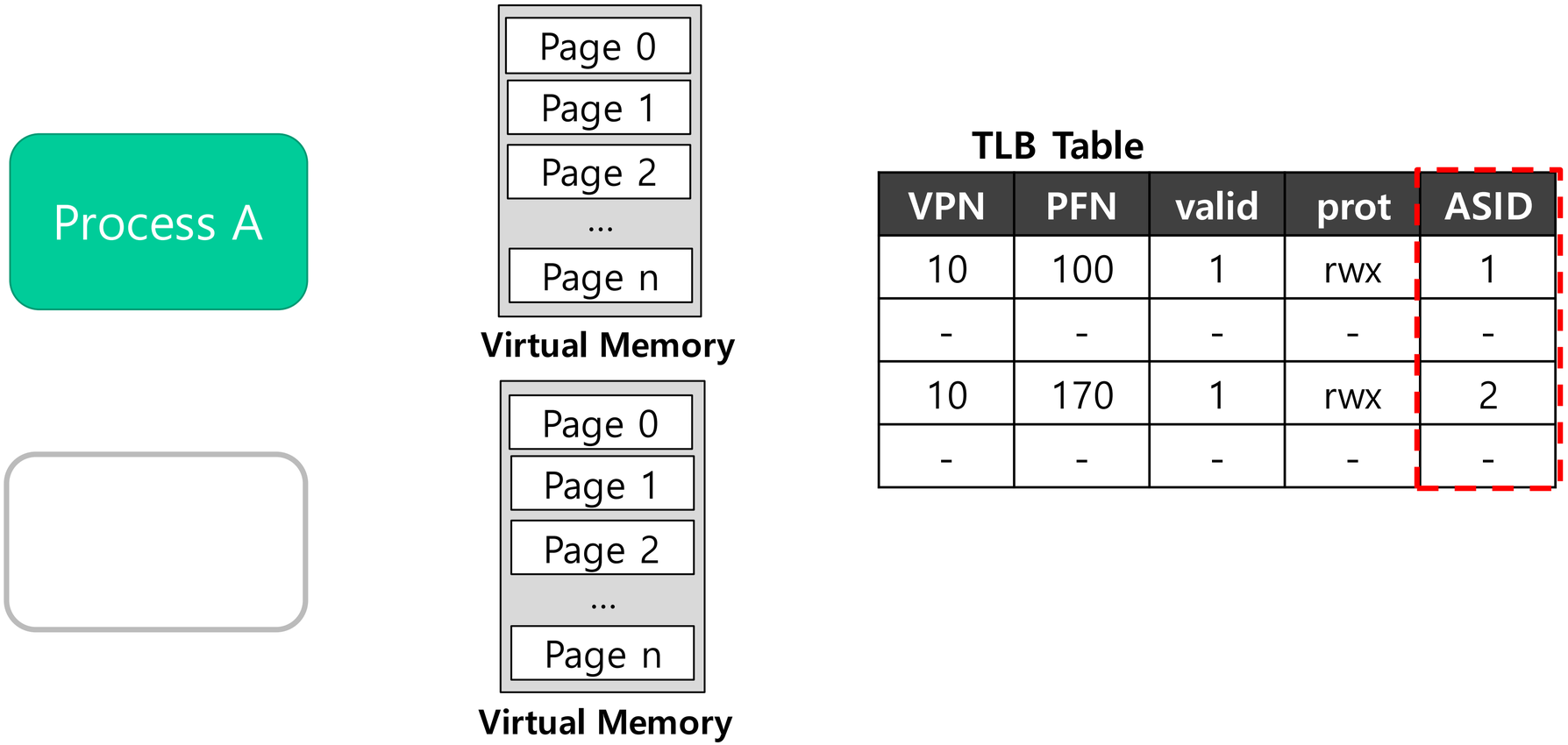


TLB Issue : Context Switching



Options

- Flush TLB on context switch
- Provide address space identifier (ADIS) field in the TLB

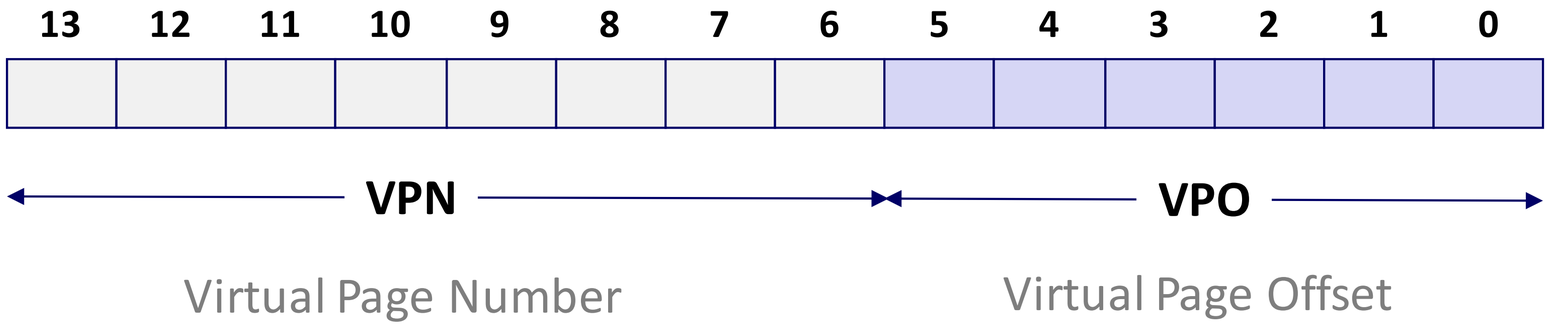


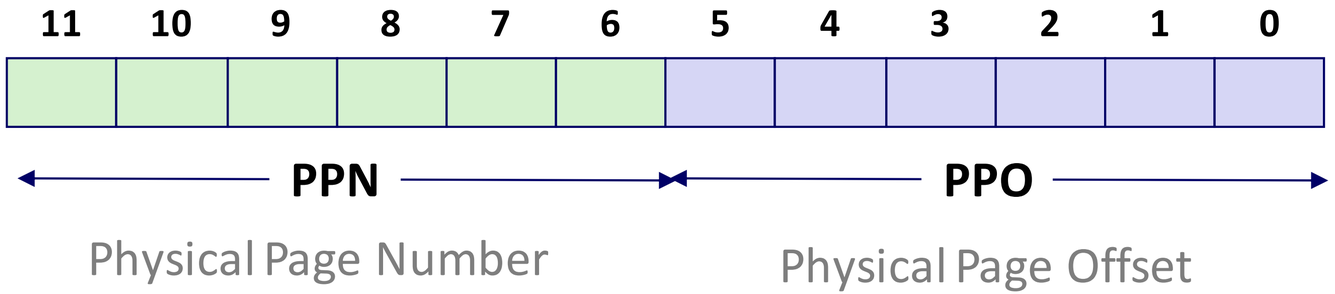
TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Simple Memory System Example

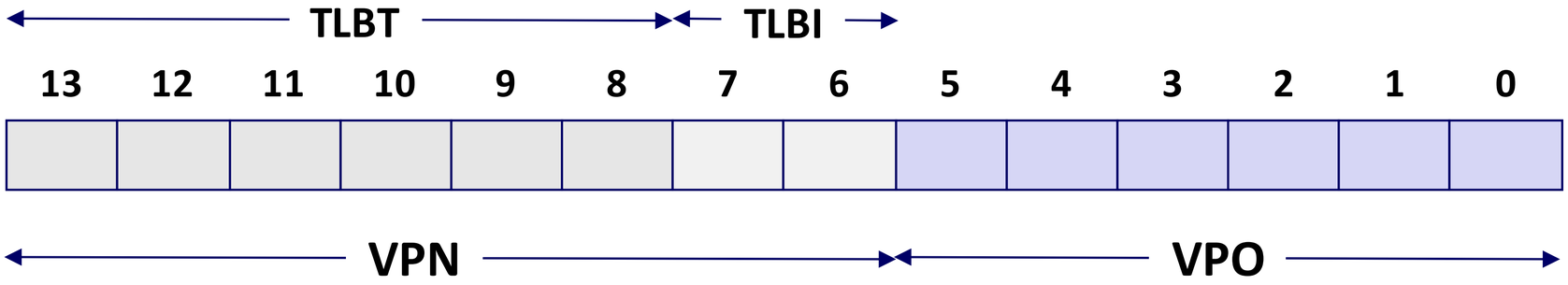
- 14-bit virtual address, 12-bit physical address, page size = 64 bytes





1. Simple Memory System TLB

- 16 entries, 4-way associative



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

2. Simple Memory System Page Table

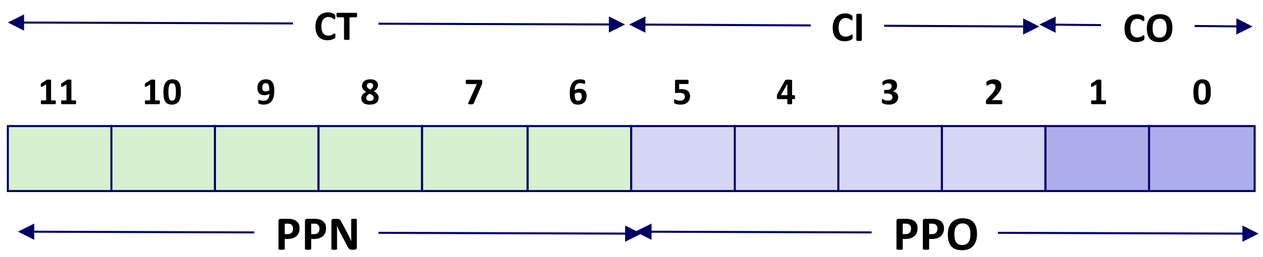
- Only show first 16 entires (out of 256)

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. Simple Memory System Cache

- 16 lines, 4-byte block size; Physically addressed, Direct mapped

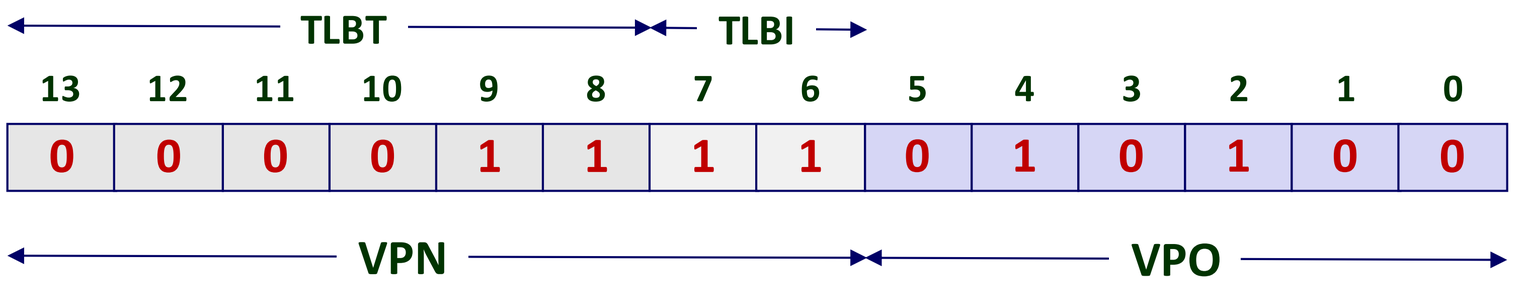


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

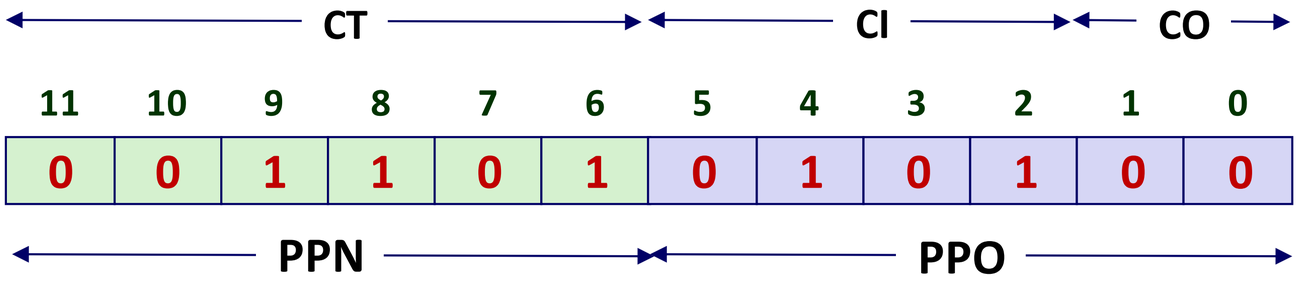
Address Translation Example #1

Virtual Address: 0x03D4



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

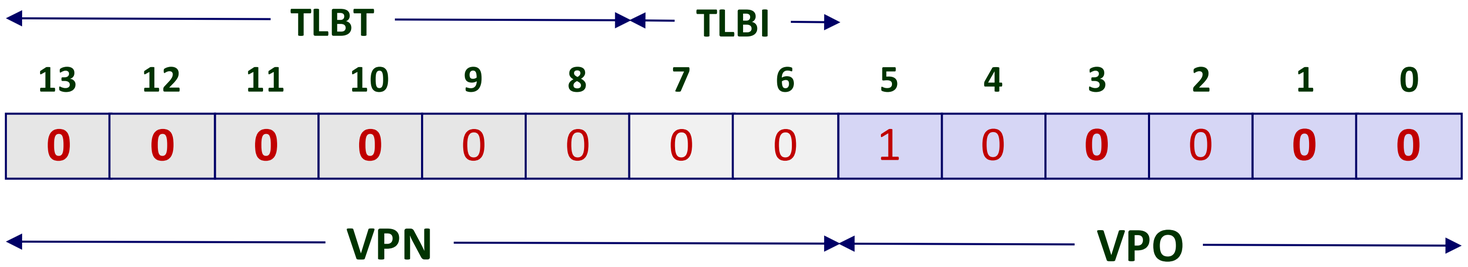
Physical Address



CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

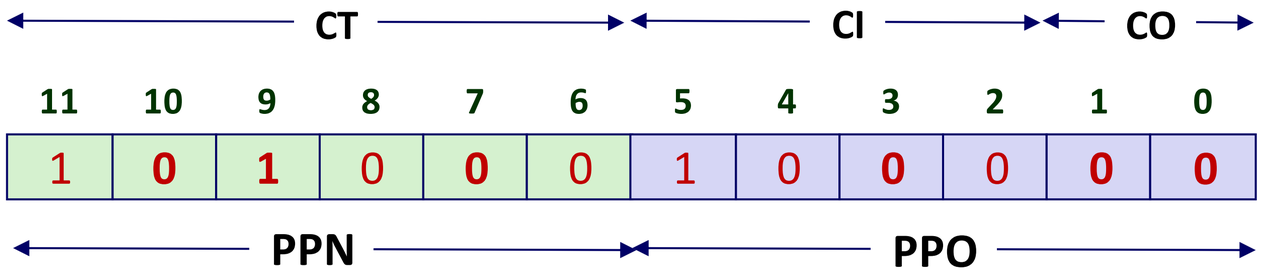
Address Translation Example #2

Virtual Address: 0x0020



VPN ___0x00___ TLBI ___ 0 TLBT __ 0x00 TLB Hit? __ N Page Fault? __ N PPN: ____0x28

Physical Address

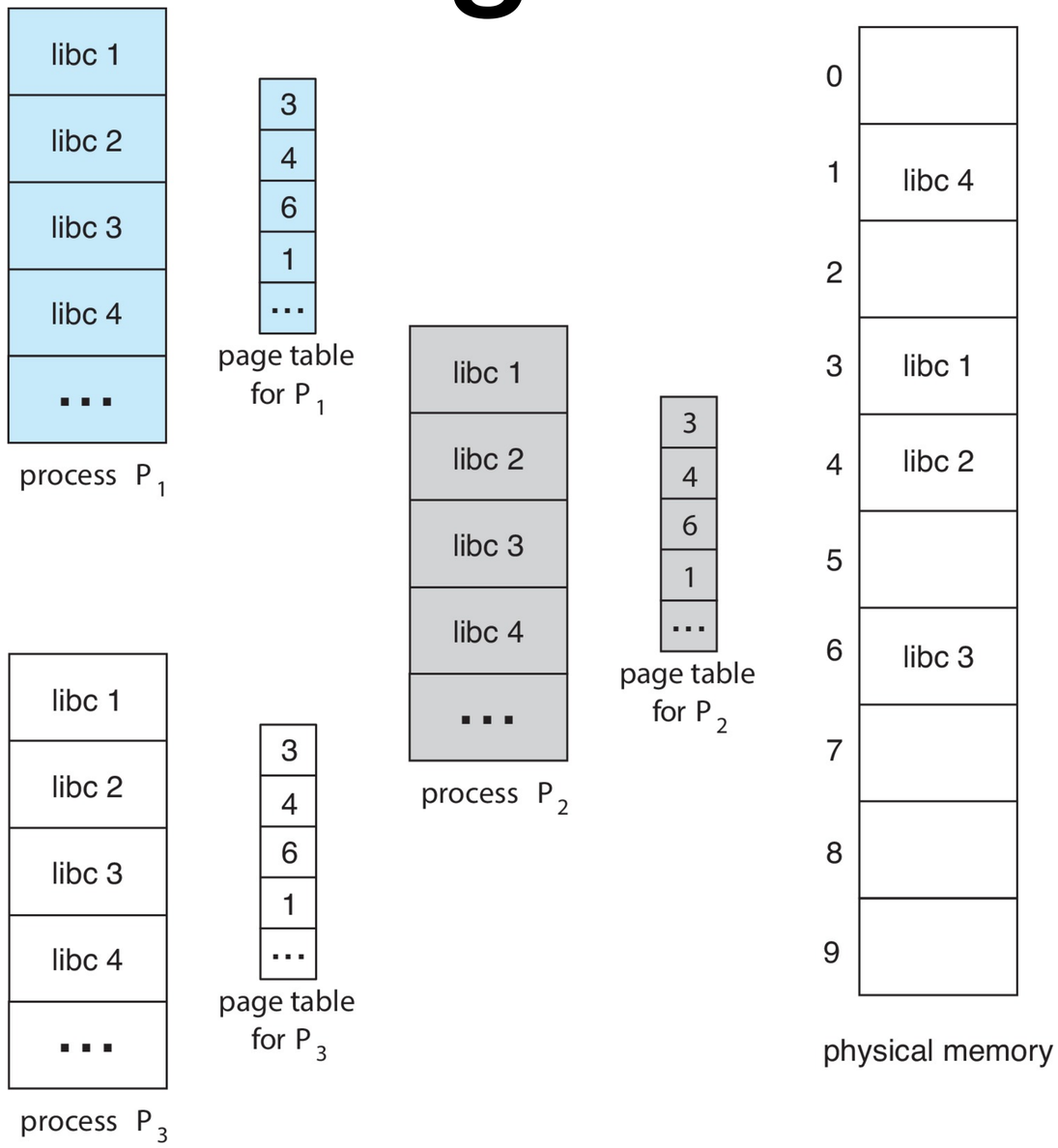


CO___ 0 CI___ CT 0x8 ___ Hit? 0x28 __ Byte: N____ Mem

Shared Pages

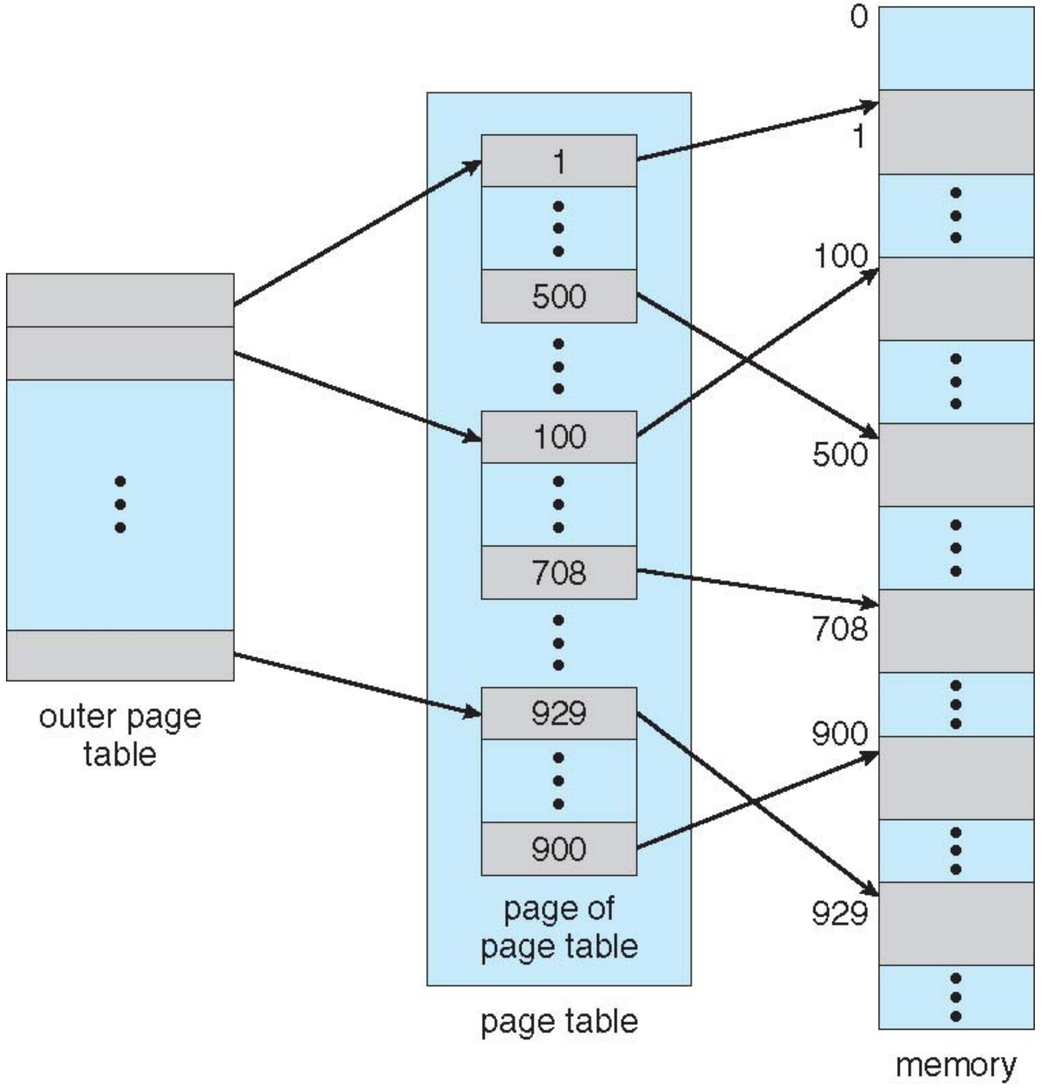
- **Shared code :**
  - One copy of read-only code shared among processes
  - Similar to multiple threads sharing same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data :**
  - Each process keeps a separate copy of the code and data
  - Pages for the private code and data can appear anywhere in logical address space

Shared Pages Example



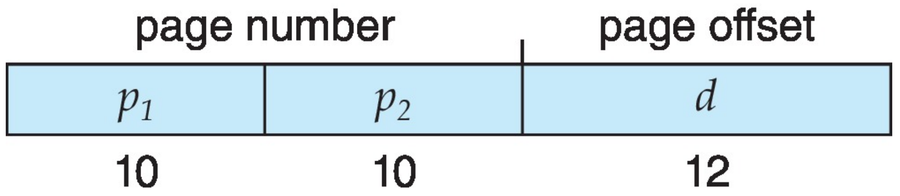
Hierarchical Page Tables

- Break up logical address space into multiple page tables
- E.g. Two-level page table : page of the page table



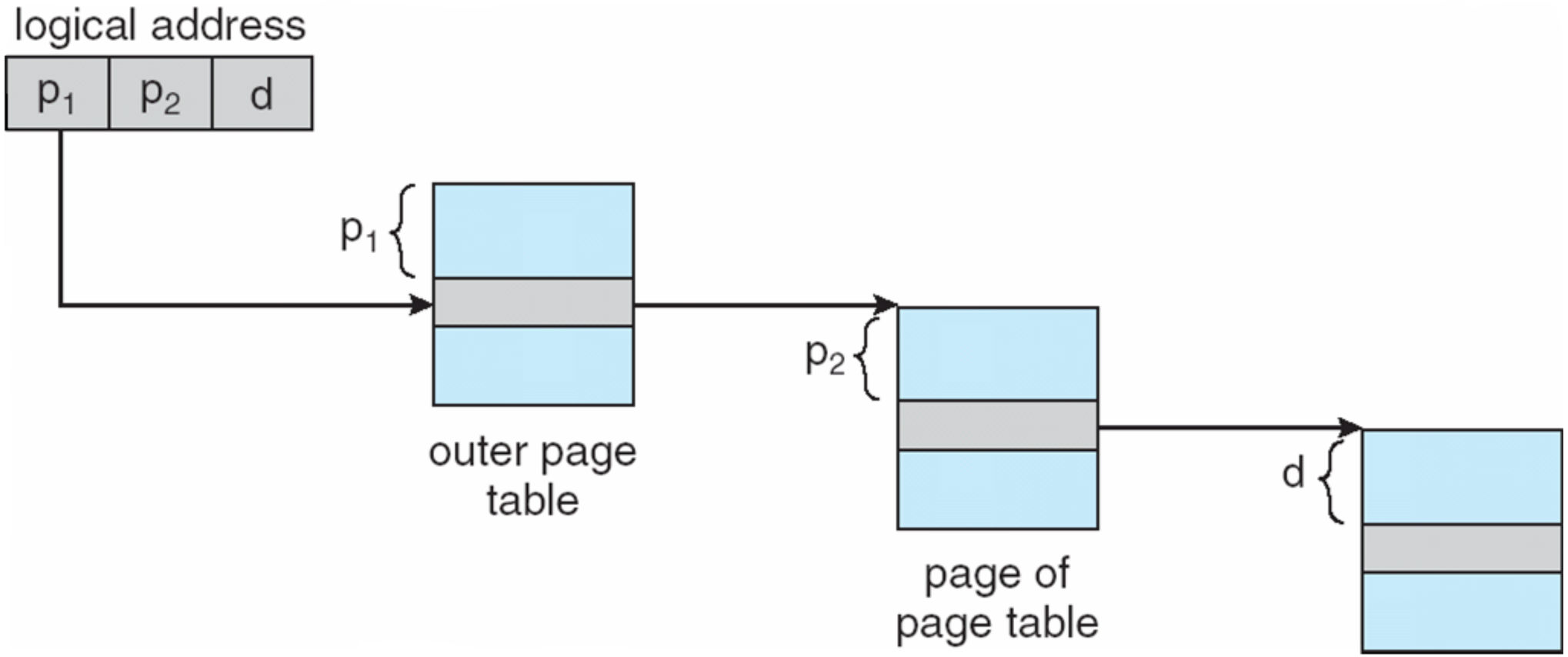
Two-Level Paging Example

- Logical address (32-bit machine with 1K page size) divided into:
  - Page number consisting of 22 bits
  - Page offset consisting of 12 bits
- Since page table is paged, page number is further divided into :
  - 10-bit page number
  - 10-bit offset
- Logical address is as follows:

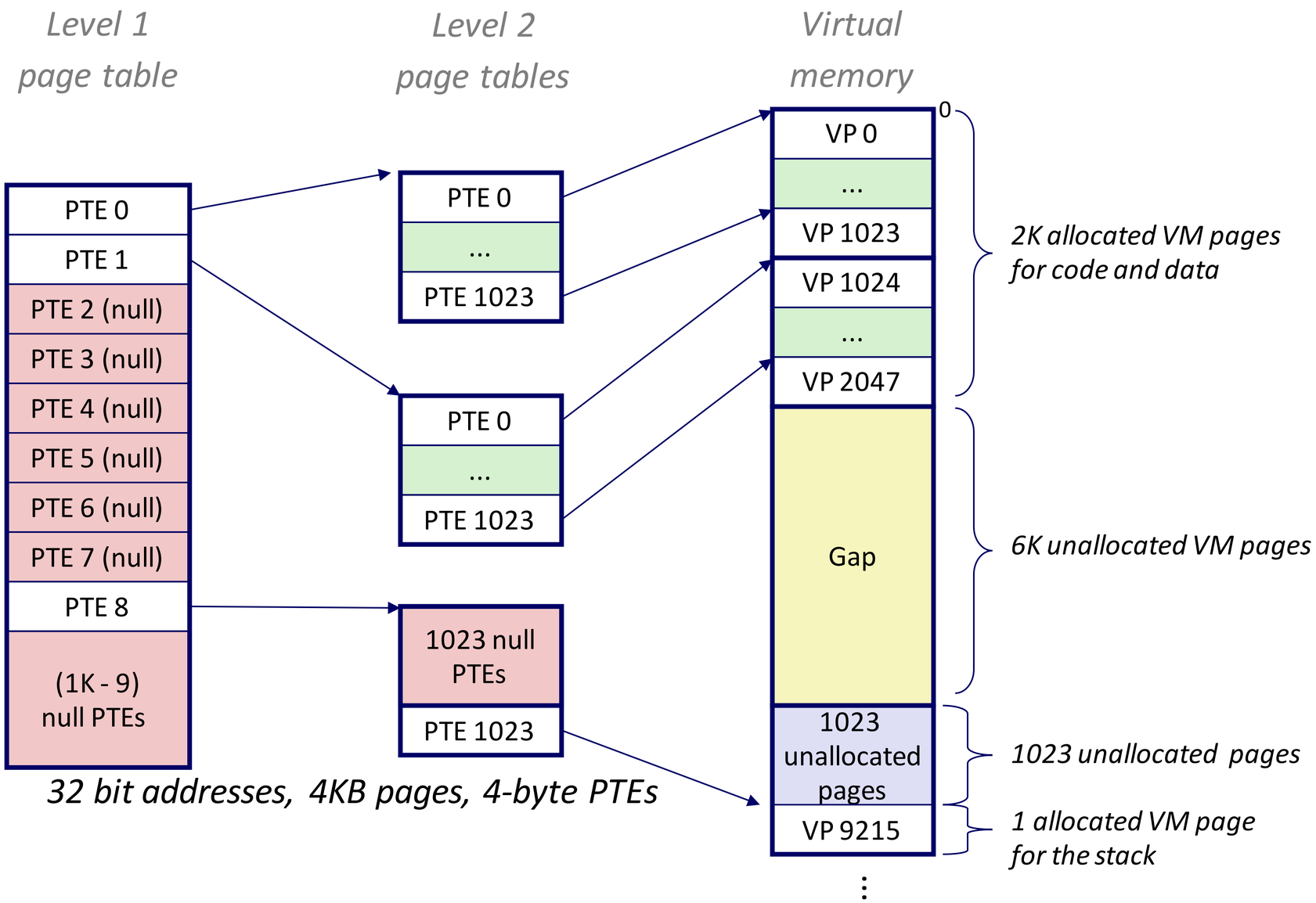


- Where p1 is index into outer page table, p2 is displacement within the page of the inner page table
- **Forward-mapped page table**

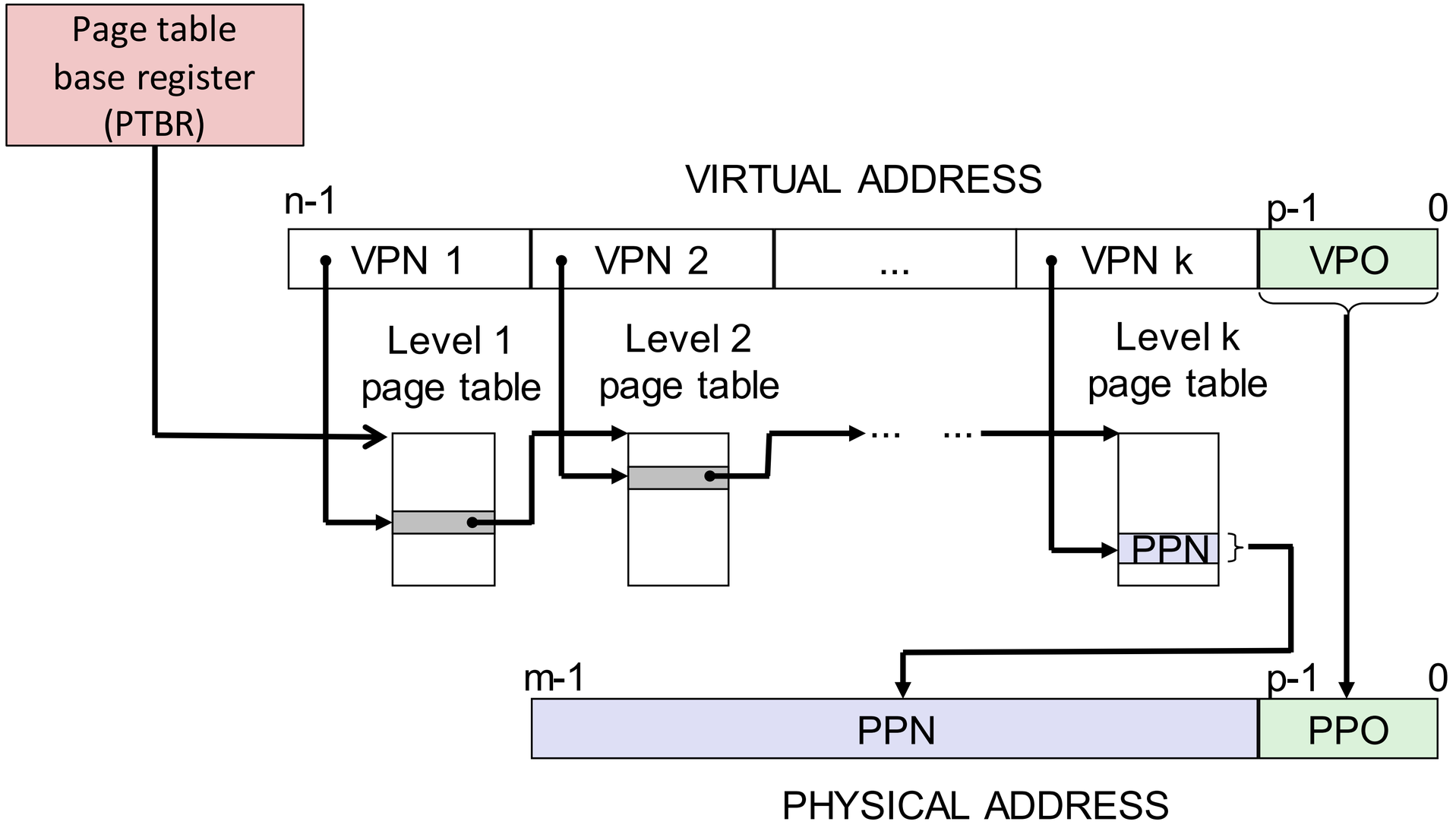
Address Translation Scheme



A Two-Level Page Table Hierarchy

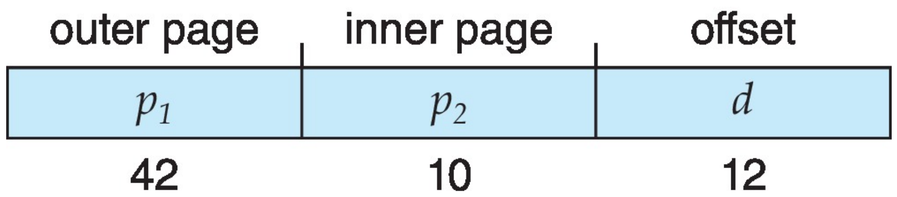


Translating with a K-Level Page Table



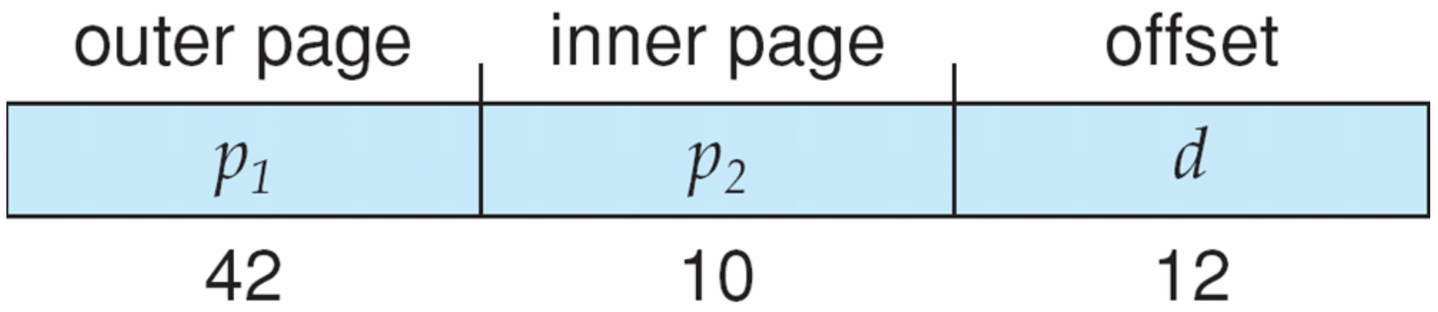
64-bit Logical Address Space

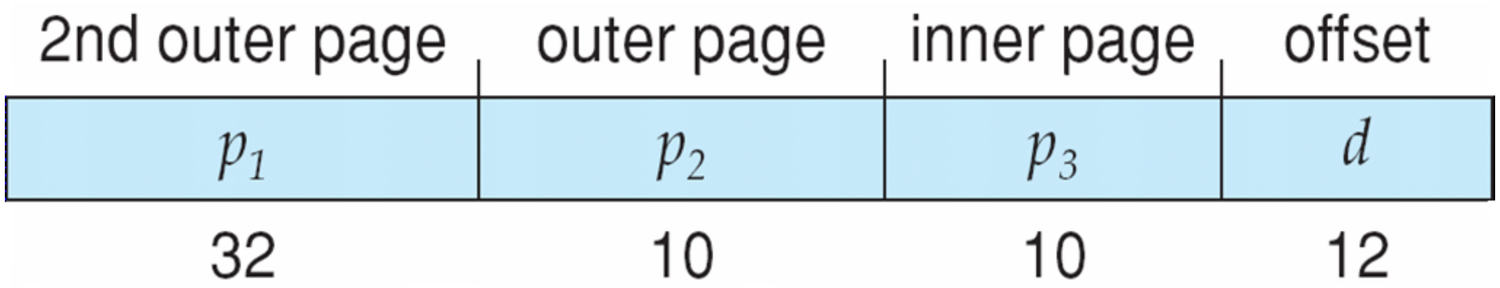
- Even two-level paging scheme not sufficient
- If page size is 4KB (2^12)
  - Page table has 2^52 entries
  - If two level scheme, inner page tables could be 2^10 4-byte entries
  - Address would look like:



  - Outer page table has 2^42 entries (2^44 bytes)
  - One solution : add a 2nd outer page table, but still large outer page table and possibly 4 memory accesses to get one physical memory location

Three-level Paging Scheme

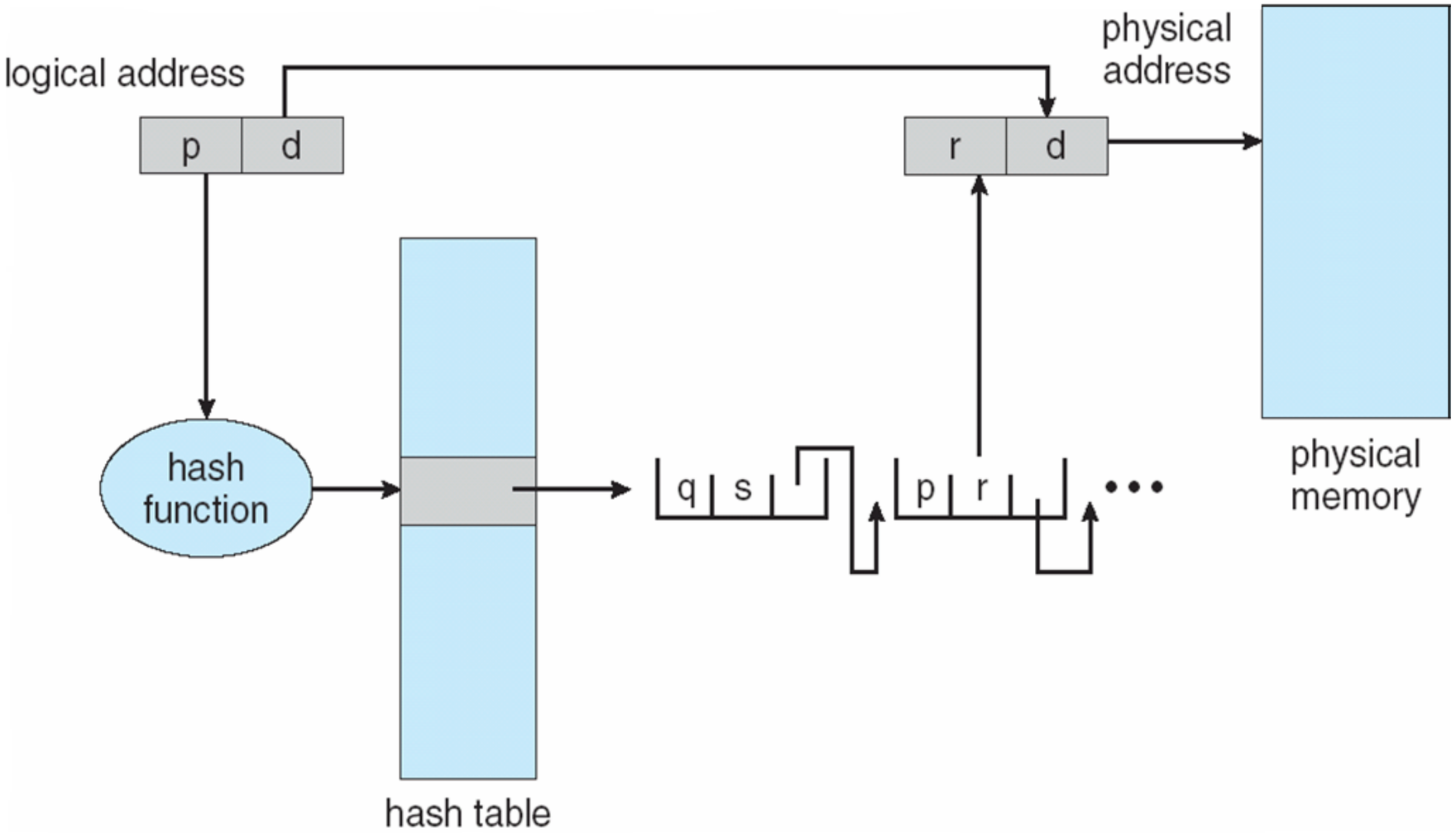




Hashed Page Tables

- Common in address spaces > 32 bits
- Virtual page number is hashed into page table
  - Contains chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If match is found, corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for sparse address spaces (memory references are non-contiguous and scattered)

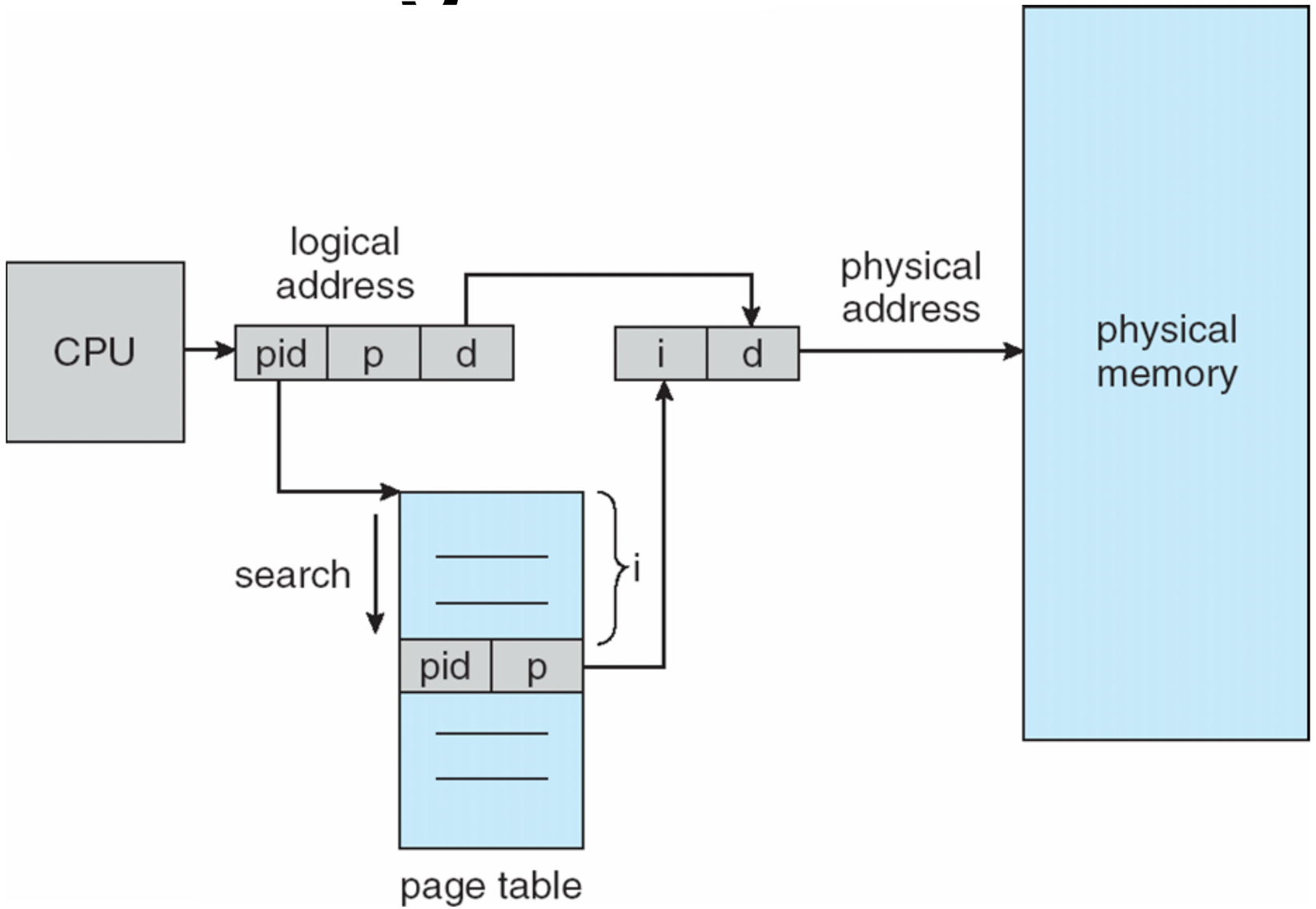
Hashed Page Tables



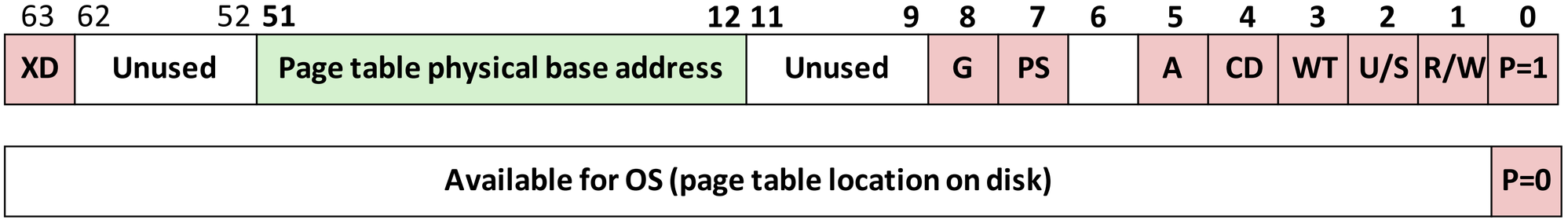
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of virtual address of page stored in that real memory location with information about the process that owns the page
- Decreases memory needed to store each page table, but increases time needed to search table when page reference occurs
- Use hash table to limit the search to one, or at most a few, page-table entries
  - TLB can accelerate accesses
- How to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Core i7 64-bit Level 1-3 Page Table Entries



Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

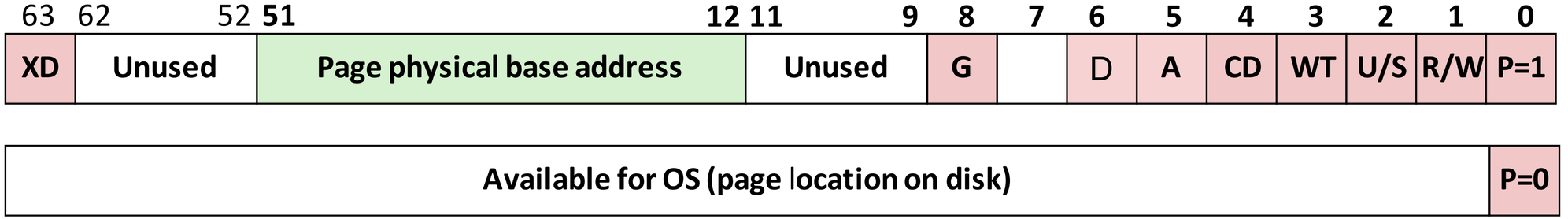
A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 64-bit Level 4 Page Table Entries



Each entry references a 4K child page. Significant fields:

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

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

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

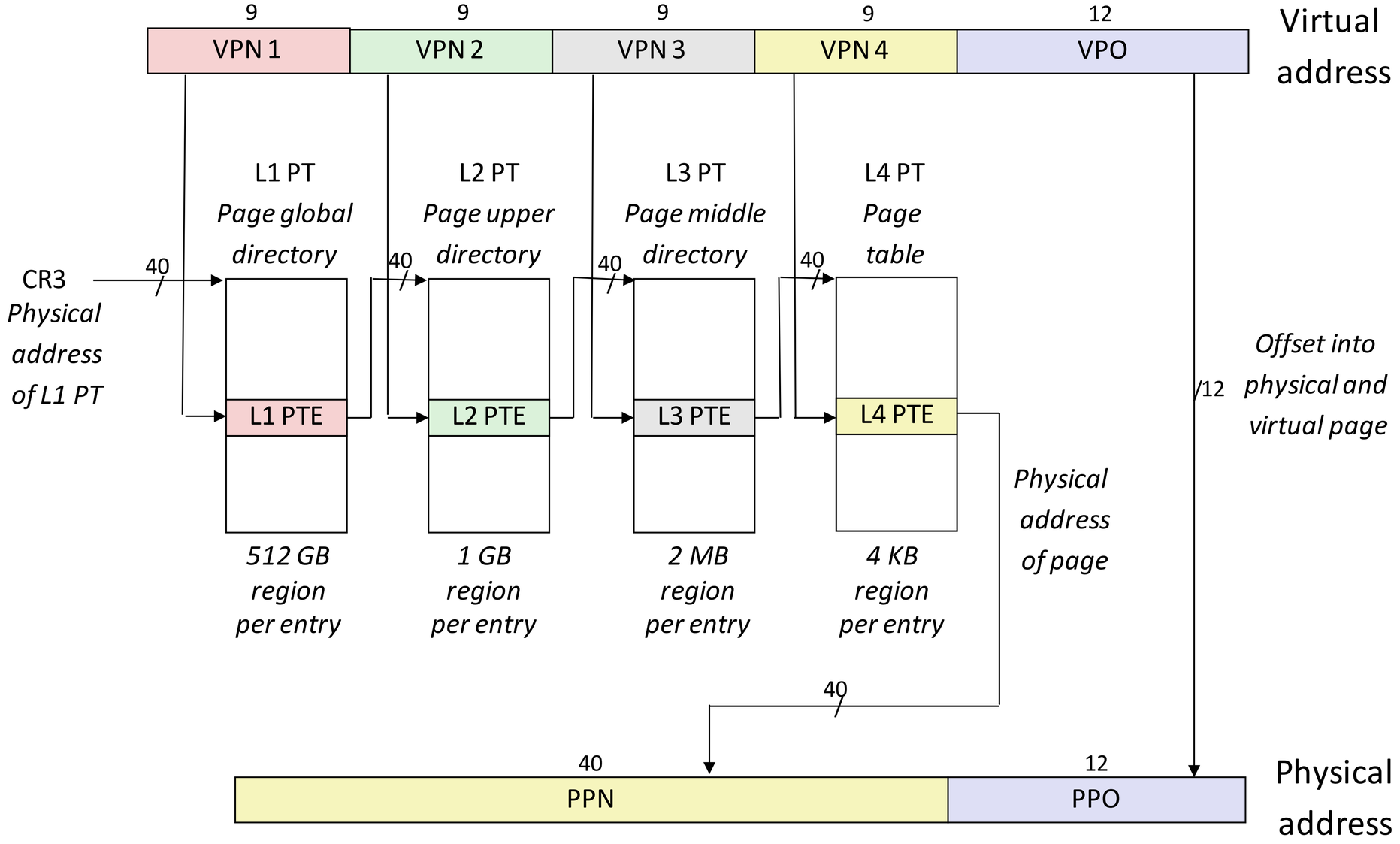
A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Core i7 64-bit Page Table Translation



But the x86 has segments, too!

- X86 has implicit segments - type of access (not address) determines which segment to use
- X86 has lots of history
  - Segmentation works differently in different x86 modes
- Different x86 segmentation / paging modes
  - 8088 (16-bit) - each segment (CS, SS, DS, ES) has a base which is multiplied by 16 and added to offset to yield physical address
  - 80286 (32-bit) : each segment type has an index into a segment table which has a base which generates physical address
  - 80386 (32-bit) - like 80286, but after segmentation address is a linear address which is then paged to created a physical address
  - 80686 (64-bit) - most segments disabled (with one exception) - logical addresses are linear addresses

32-bit X86 Address Translation