18-447
Computer Architecture
Lecture 13: Virtual Memory II

Lecturer: Rachata Ausavarungnirun
Carnegie Mellon University

Spring 2014, 2/17/2014

(with material from Onur Mutlu, Justin Meza and Yoongu Kim)

Announcement

- Lab 2 grades and feedback available tonight
- Lab 3 due this Friday (21st Feb.)
- HW 2 grades and feedback available tonight

- Midterm 1 in two weeks (5th Mar.)

- Paper summary during this week recitations

Two problems with Page Table

- Problem #1: Page table is too large
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s

- Solution: Multi-level page table

Two problems with Page Table

- Problem #1: Page table is too large
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s

- Problem #2: Page table is in memory
  - Before every memory access, always fetch the PTE from the slow memory? ∙ Large performance penalty

Translation Lookaside Buffer (TLB)

- A hardware structure where PTEs are cached
  - Q: How about PDEs? Should they be cached?
- Whenever a virtual address needs to be translated, the TLB is first searched: “hit” vs. “miss”

- Example: 80386
  - 32 entries in the TLB
  - TLB entry: tag + data
    - Tag: 20-bit VPN + 4-bit flag (valid, dirty, R/W, U/S)
    - Data: 20-bit PPN
    - Q: Why is the tag needed?

Context Switches

- Assume that Process **X** is running
  - Process **X**'s VPN 5 is mapped to PPN <u>100</u>
  - The TLB caches this mapping
    - VPN 5 $\rightarrow$ PPN 100
- Now assume a context switch to Process **Y**
  - Process **Y**'s VPN 5 is mapped to PPN <u>200</u>
  - When Process Y tries to access VPN 5, it searches the TLB
    - Process **Y** finds an entry whose tag is 5
    - TLB hit!
    - The PPN must be 100!
    - … Are you sure?

Context Switches (cont'd)

- Approach #1. Flush the TLB
  - Whenever there is a context switch, flush the TLB
    - All TLB entries are invalidated
  - Example: 80836
    - Updating the value of CR3 signals a context switch
    - This automatically triggers a TLB flush

- Approach #2. Associate TLB entries with processes
  - All TLB entries have an extra field in the tag ...
    - That identifies the process to which it belongs
  - Invalidate only the entries belonging to the old process
  - Example: Modern x86, MIPS

Handling TLB Misses

- The TLB is small; it cannot hold <u>all</u> PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page directory/table
    - Large performance penalty

- Who handles TLB misses?
  1. Hardware-Managed TLB
  2. Software-Managed TLB

Handling TLB Misses (cont'd)

- Approach #1. **Hardware-Managed** (e.g., x86)
  - The hardware does the **page walk**
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - All of this is done transparently

- Approach #2. **Software-Managed** (e.g., MIPS)
  - The hardware raises an exception
  - The operating system does the **page walk**
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

Handling TLB Misses (cont'd)

- Hardware-Managed TLB
  - Pro: No exceptions. Instruction just stalls
  - Pro: Independent instructions may continue
  - Pro: Small footprint (no extra instructions/data)
  - Con: Page directory/table organization is etched in stone

- Software-Managed TLB
  - Pro: The OS can design the page directory/table
  - Pro: More advanced TLB replacement policy
  - Con: Flushes pipeline
  - Con: Performance overhead

Protection with Virtual Memory

- A normal user process should not be able to:
  - Read/write another process' memory
  - Write into shared library data
- How does virtual memory help?
  - Address space isolation
  - Protection information in page table
  - Efficient clearing of data on newly allocated pages

Protection: Leaked Information

- Example (with the virtual memory we've discussed so far):
  - Process A writes "my password = …" to virtual address 2
  - OS maps virtual address 2 to physical page 4 in page table
  - Process A no longer needs virtual address 2
  - OS unmaps virtual address 2 from physical page 4 in page table
- Attack vector:
  - Sneaky Process B continually allocates pages and searches for "my password = <string>"

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages

- Idea: Store access control information on a page basis in the process's page table

- Enforce access control at the same time as translation

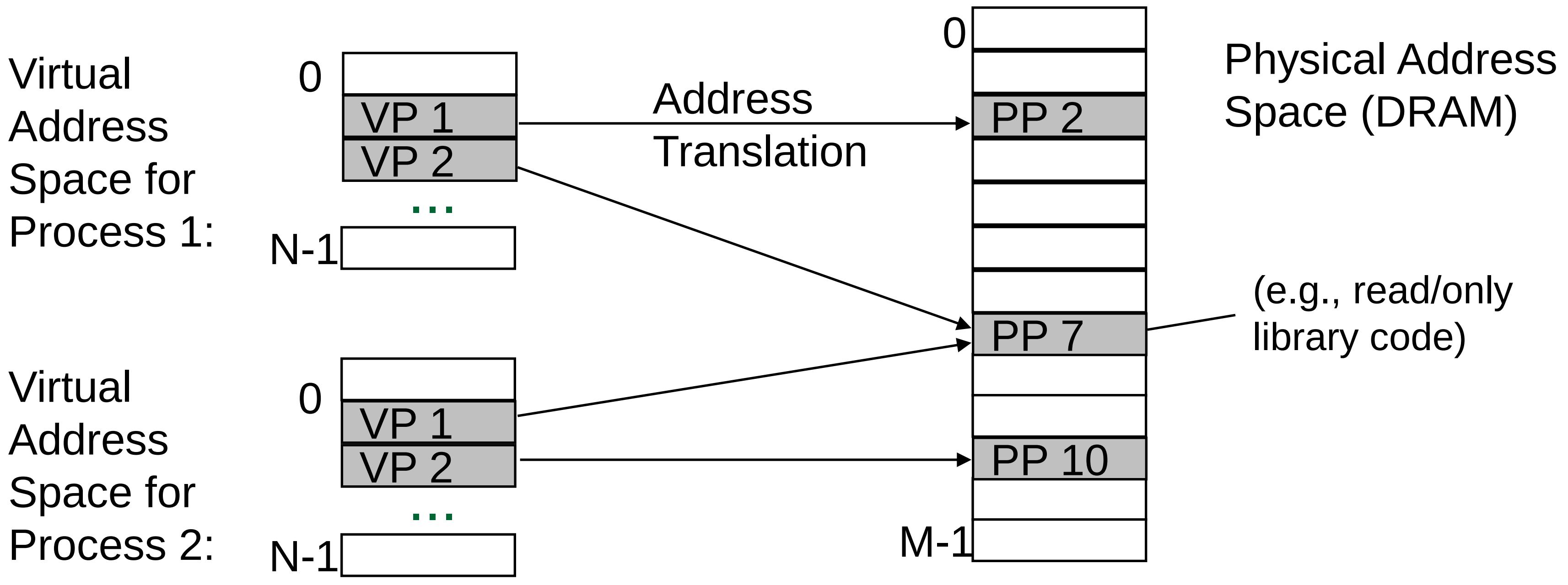
→ Virtual memory system serves two functions today

Address translation (for illusion of large physical memory)

Access control (protection)

Page Table is Per Process

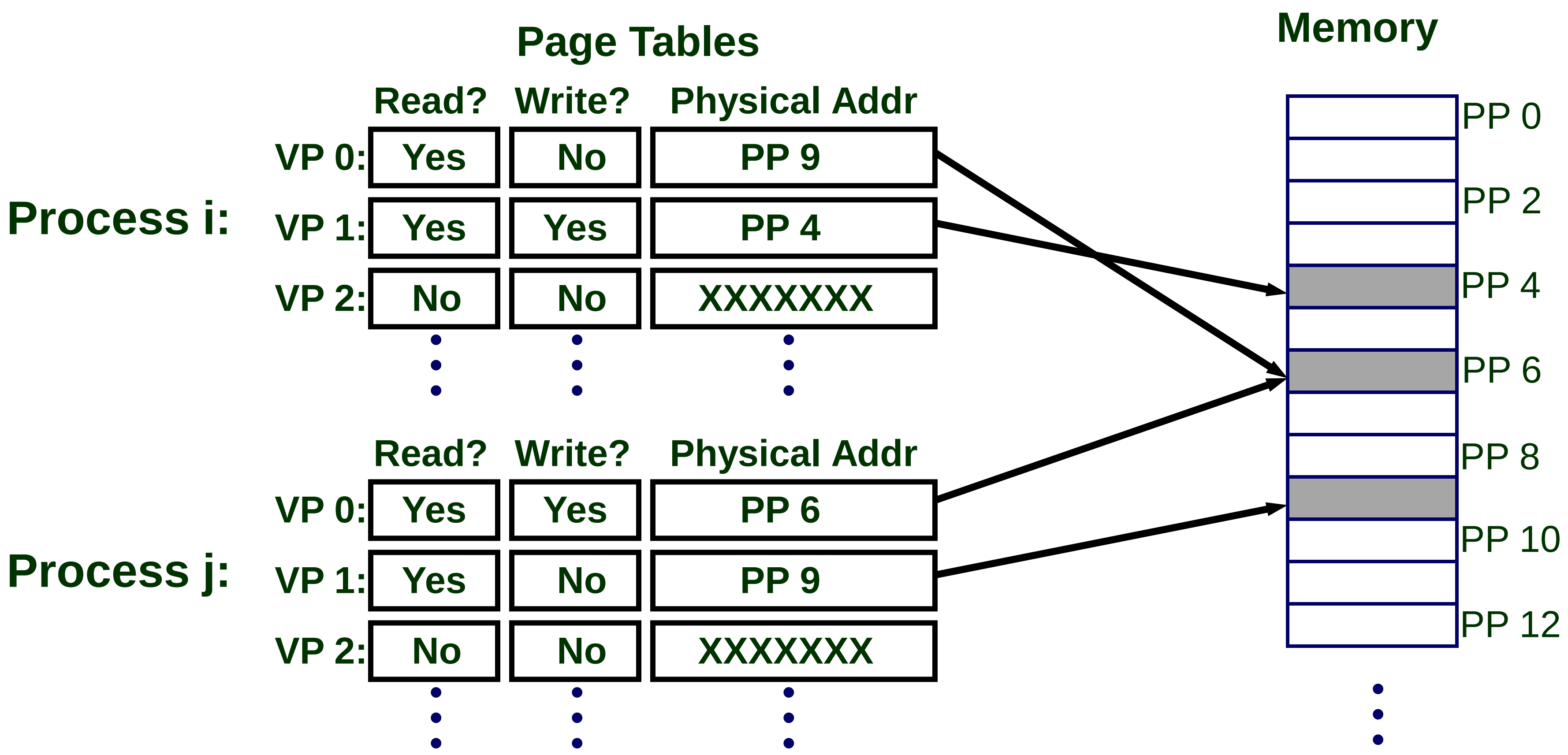
- Each process has its own virtual address space
    - Full address space for each program
    - Simplifies memory allocation, sharing, linking and loading.



Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Page fault handler checks these before remapping
  - If violated, generate exception (Access Protection exception)



Privilege Levels in x86

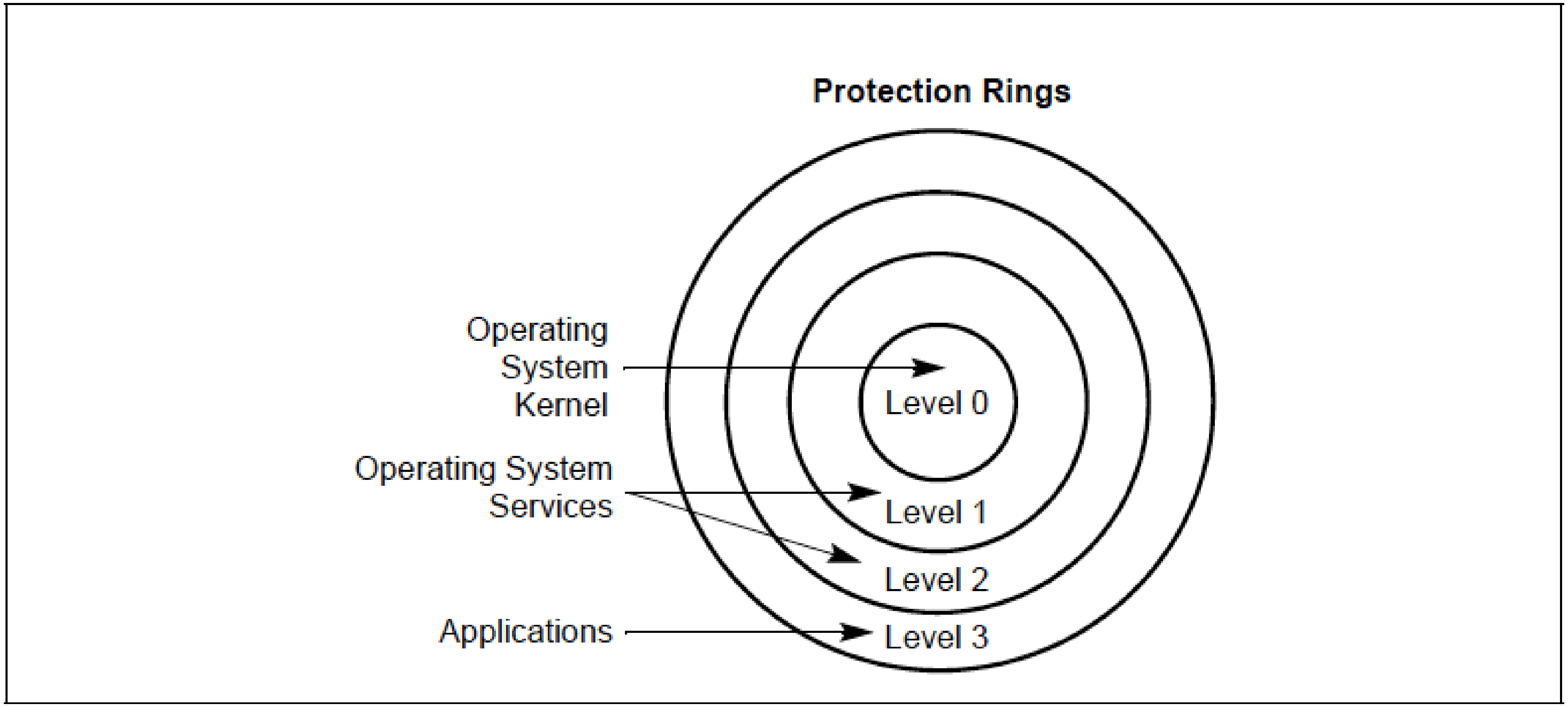


Figure 5-3. Protection Rings

x86: Privilege Level (Review)

- Four **privilege levels** in x86 (referred to as **rings**)
  - Ring 0: Highest privilege (operating system)
  - Ring 1: Not widely used
  - Ring 2: Not widely used
  - Ring 3: Lowest privilege (user applications)

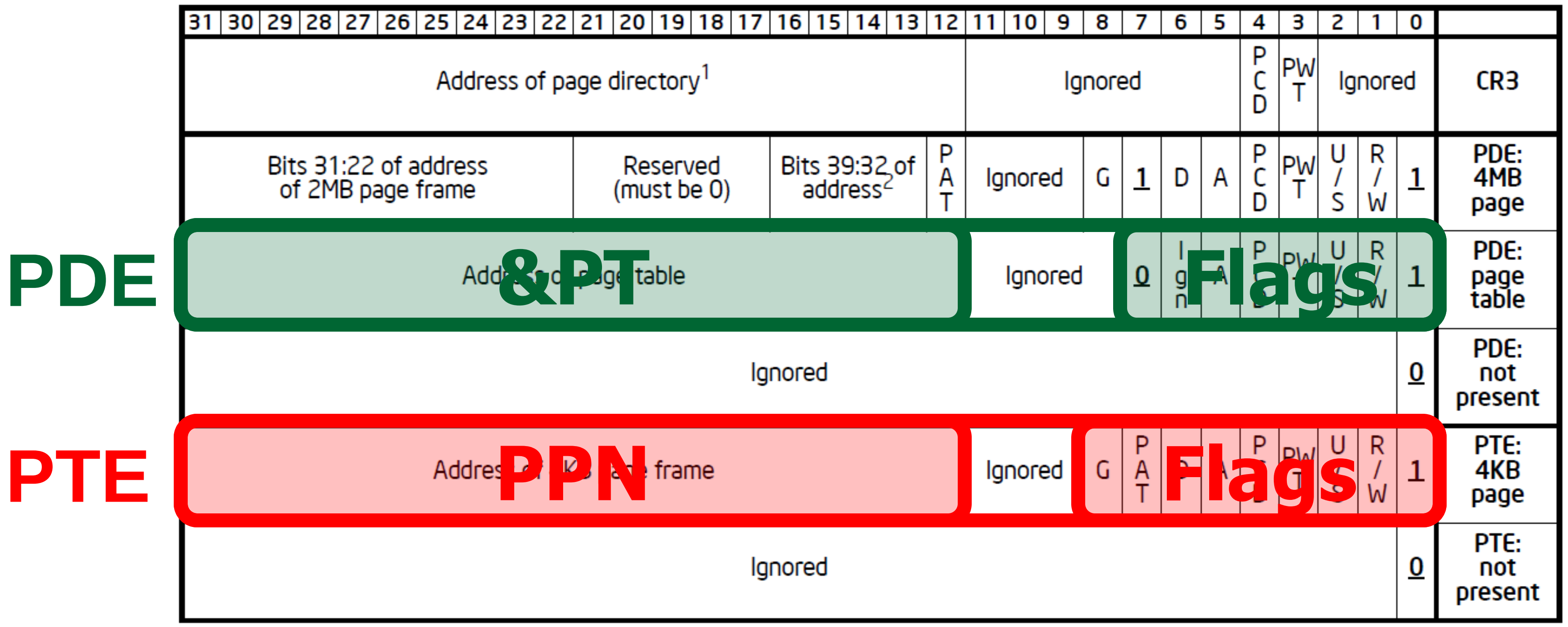
"Supervisor"

"User"

- **Current Privilege Level** (CPL) determined by:
  - Address of the instruction that you are executing
  - Specifically, the **Descriptor Privilege Level** (DPL) of the code segment

x86: A Closer Look at the PDE/PTE

- **PDE:** Page Directory Entry (32 bits)
- **PTE:** Page Table Entry (32 bits)


PDE
&PT
Flags
PTE
PPN
Flags

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Protection: PDE's Flags

- Protects all 1024 pages in a page table

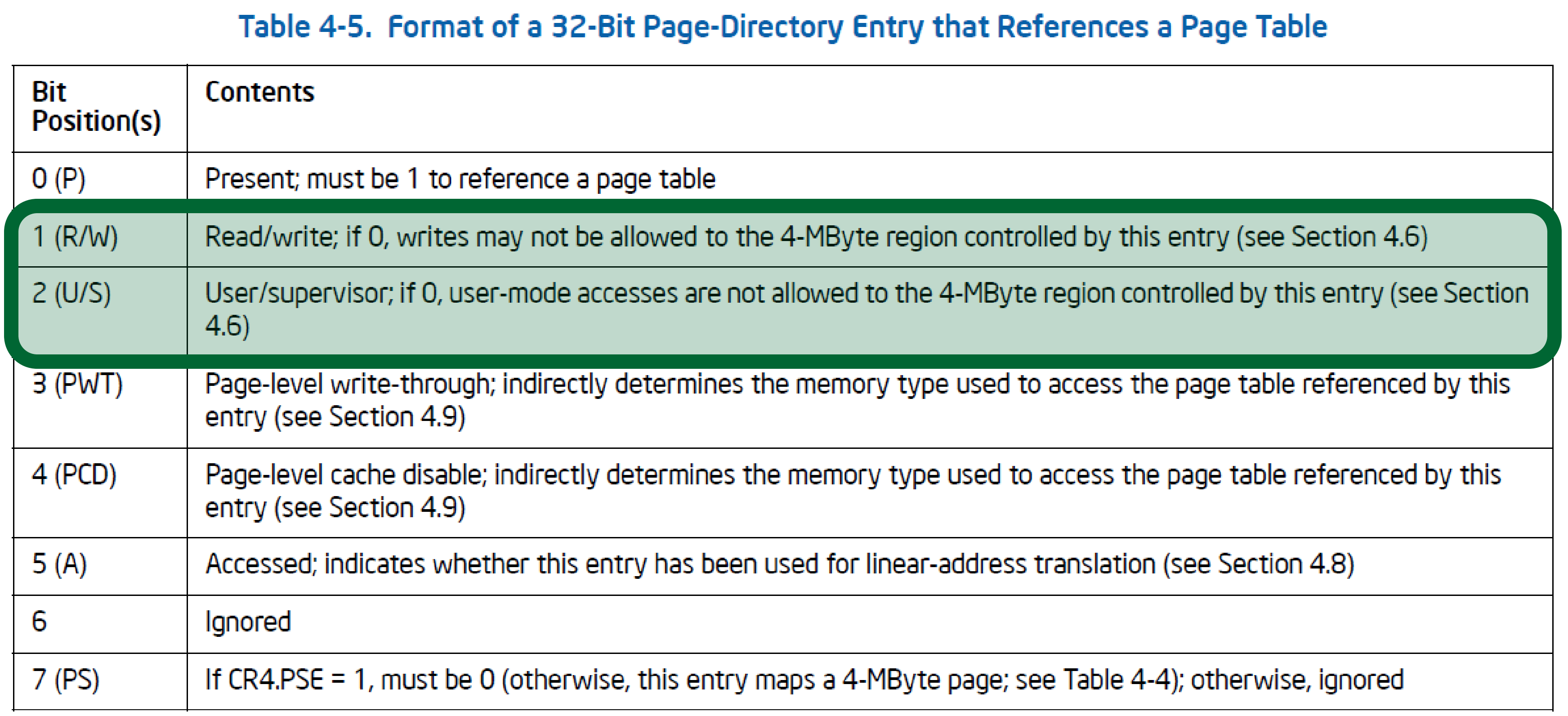


Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored

Protection: PTE's Flags

- Protects one page at a time

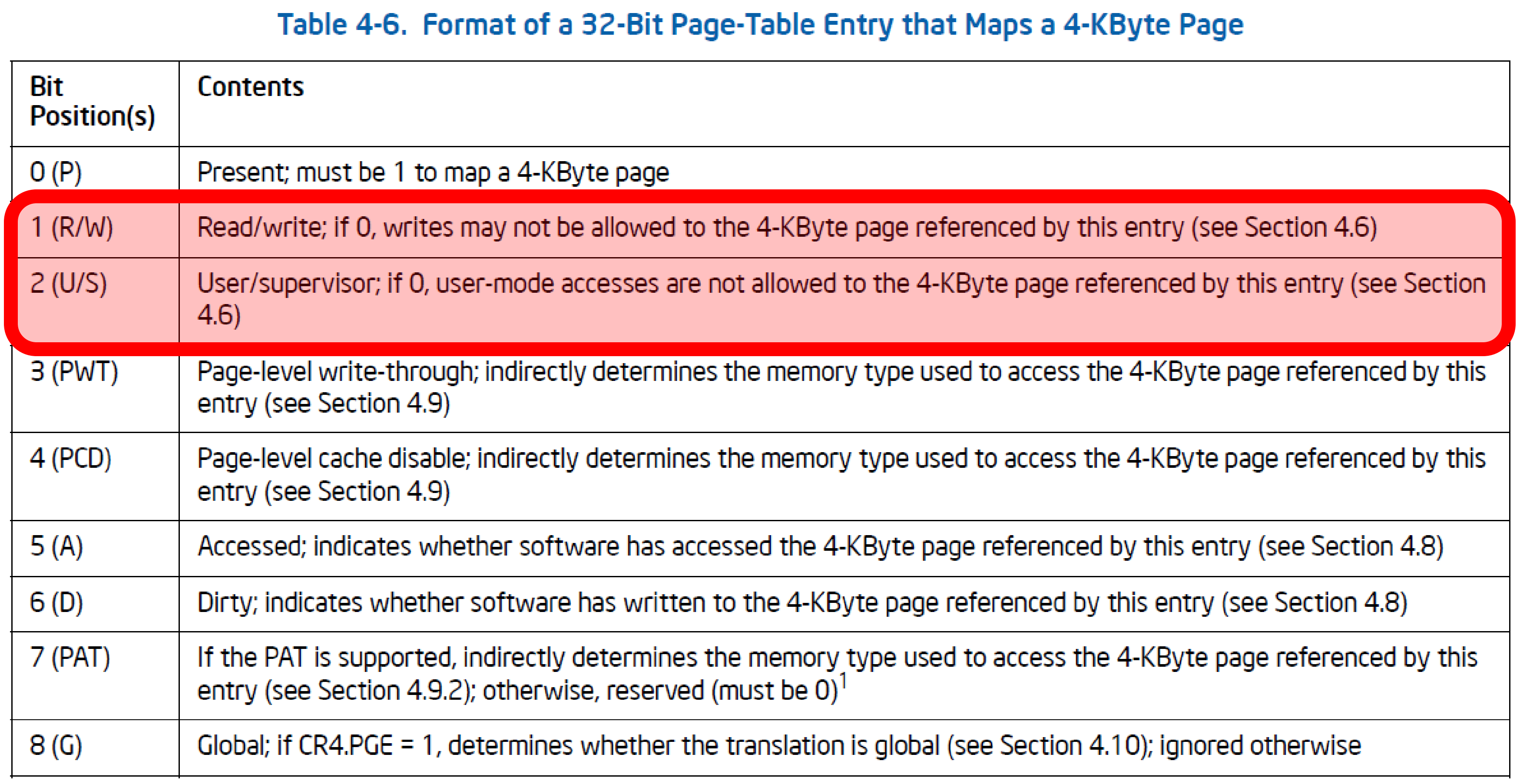


Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Protection: PDE + PTE = ???

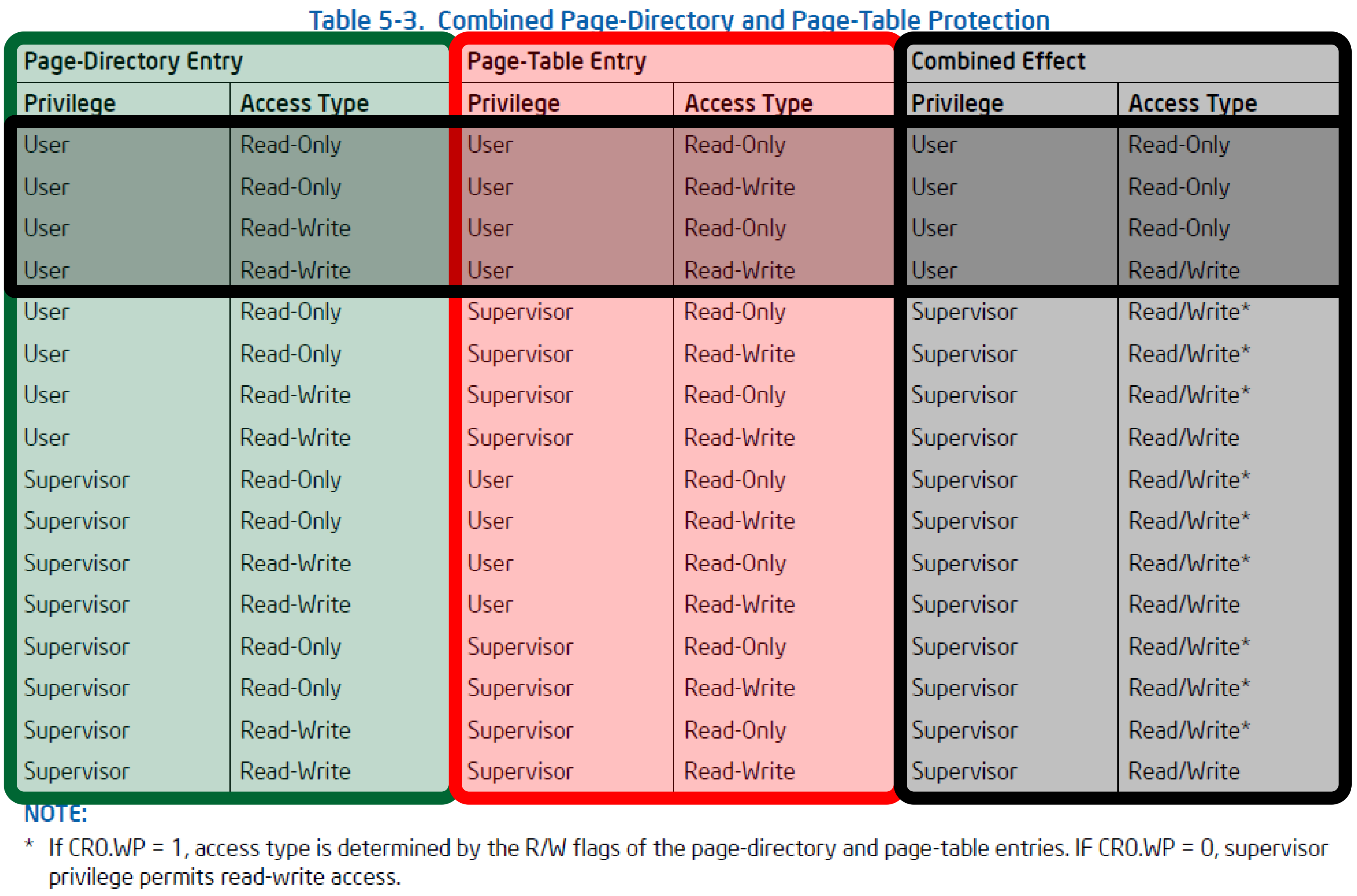


Table 5-3. Combined Page-Directory and Page-Table Protection

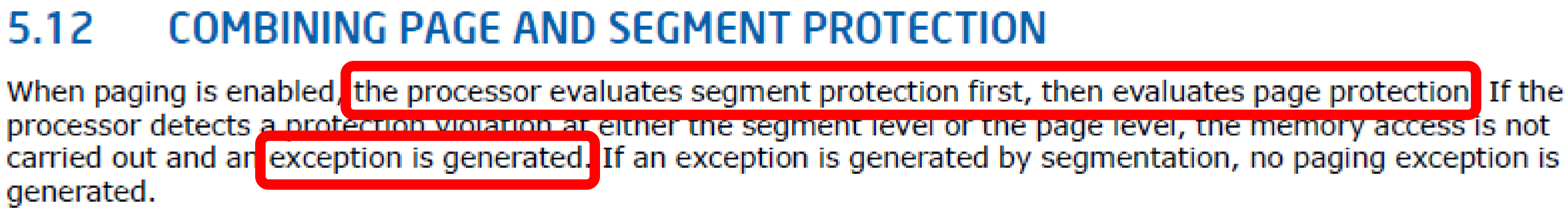
Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

Protection: Segmentation + Paging

- **Paging** provides protection
  - Flags in the PDE/PTE (x86)
    - Read/Write
    - User/Supervisor
    - Executable (x86-64)
- **Segmentation** <u>also</u> provides protection
  - Flags in the Segment Descriptor (x86)
    - Read/Write
    - Descriptor Privilege Level
    - Executable



5.12 COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Aside: Protection w/o Virtual Memory

- Question: Do we need virtual memory for protection

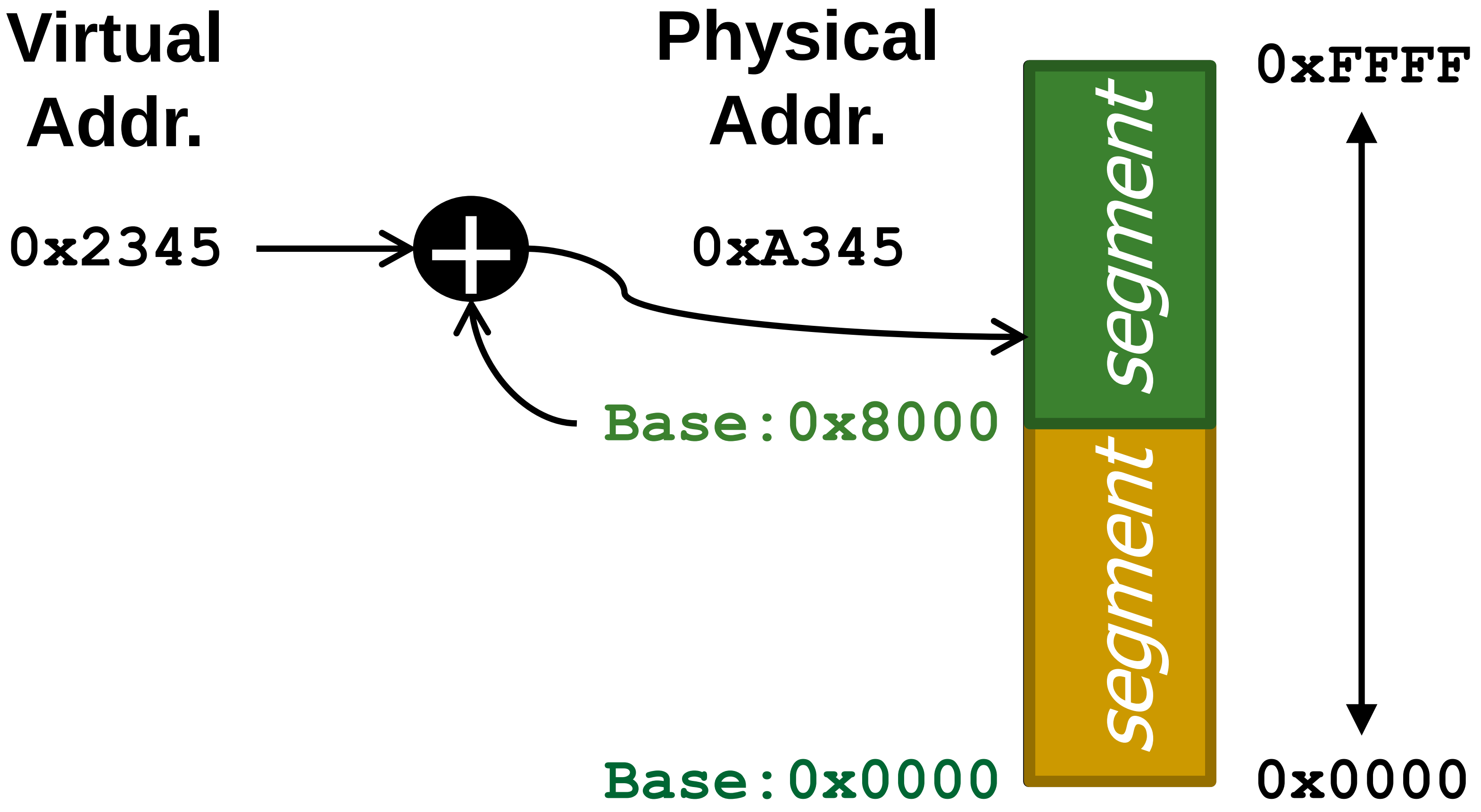
- Answer: No

- Other ways of providing memory protection
  - Base and bound registers
  - Segmentation

- None of these are as elegant as page-based access control
  - They run into complexities as we need more protection capabilites
  - Virtual memory integrates

Overview of Segmentation

- Divide the ***physical address space*** into ***segments***
  - The segments may overlap



Segmentation in Intel 8086

- ***Intel 8086*** (Late 70s)
  - 16-bit processor
  - 4 ***segment registers*** that store the ***base address***

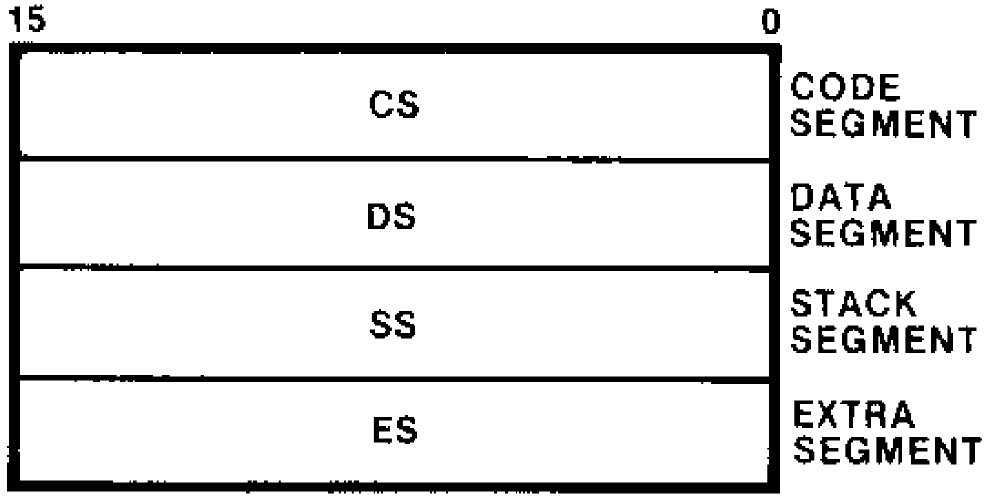
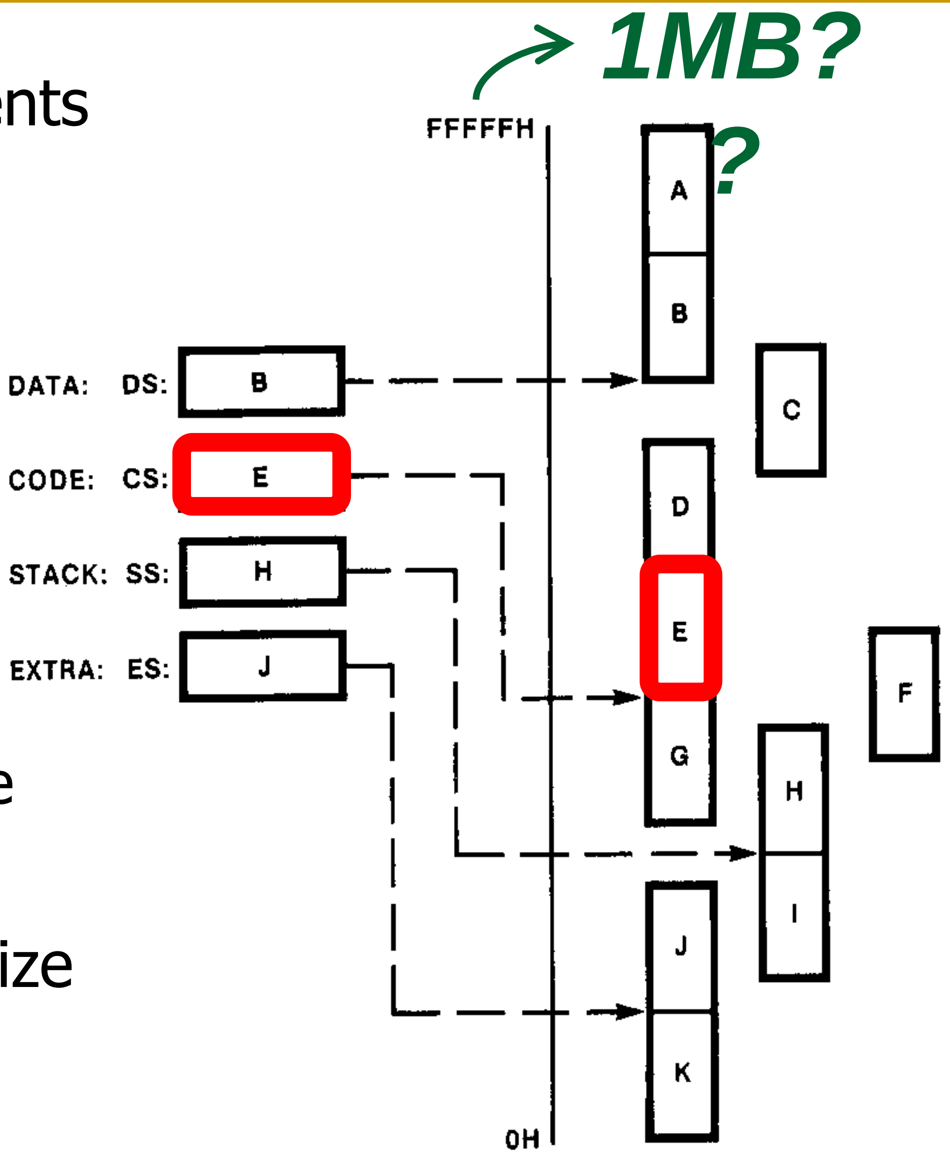


Figure 2-8. Segment Registers

Intel 8086: Specifying a Segment

- There can be many segments
- But only 4 of them are addressable at once
- Which 4 depends on the 4 segment registers
  - The programmer sets the segment register value
- Each segment is 64KB in size
  - Because 8086 is 16-bit



Intel 8086: Translation

- 8086 is a 16-bit processor ...
  - How can it address up to `0xFFFFF` (1MB)?

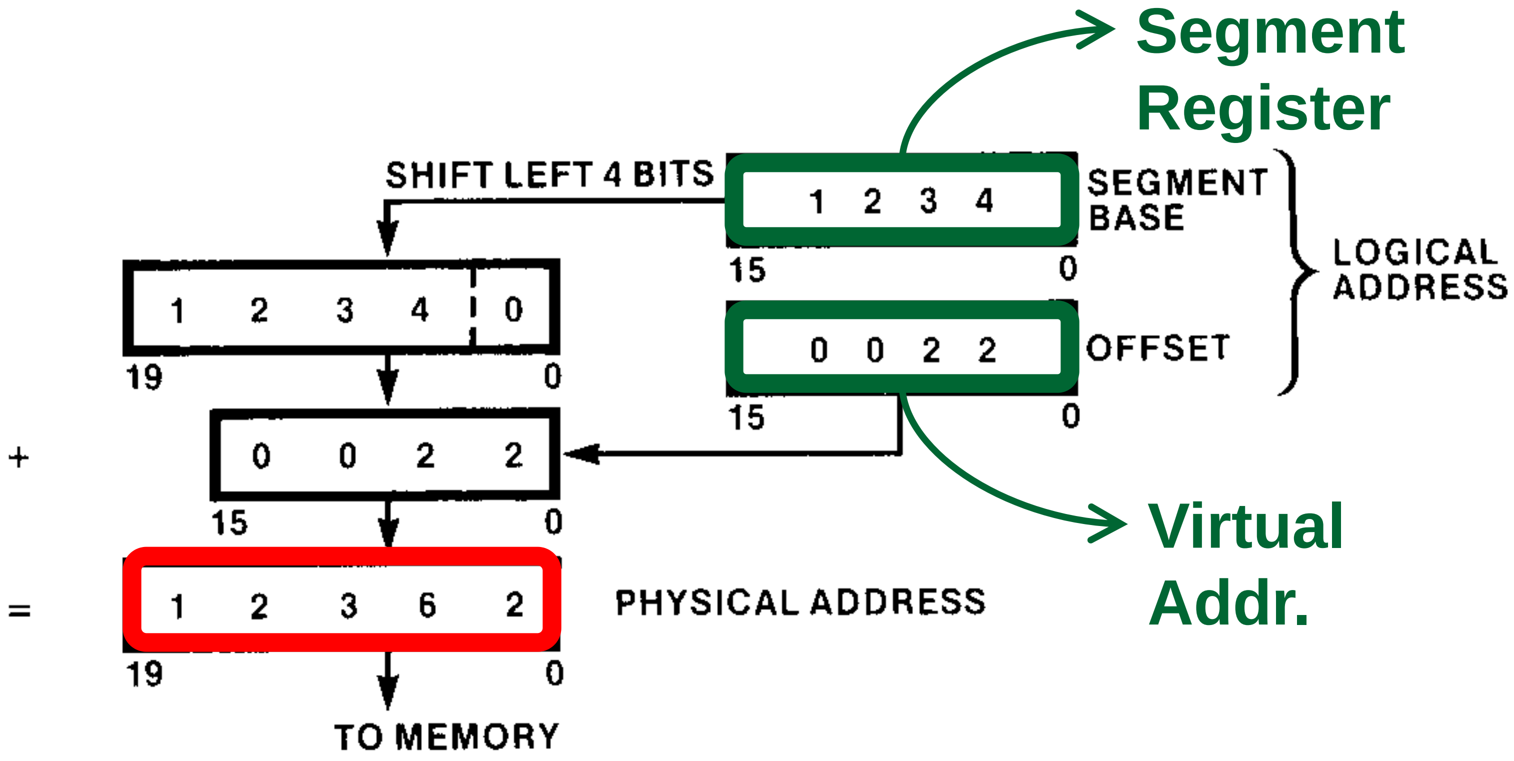


Figure 2-18. Physical Address Generation

Intel 8086: Which Segment Register?

- **Q:** For a memory access, how does the machine know which of the 4 segment register to use?
  - **A:** Depends on the *type of memory access*

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

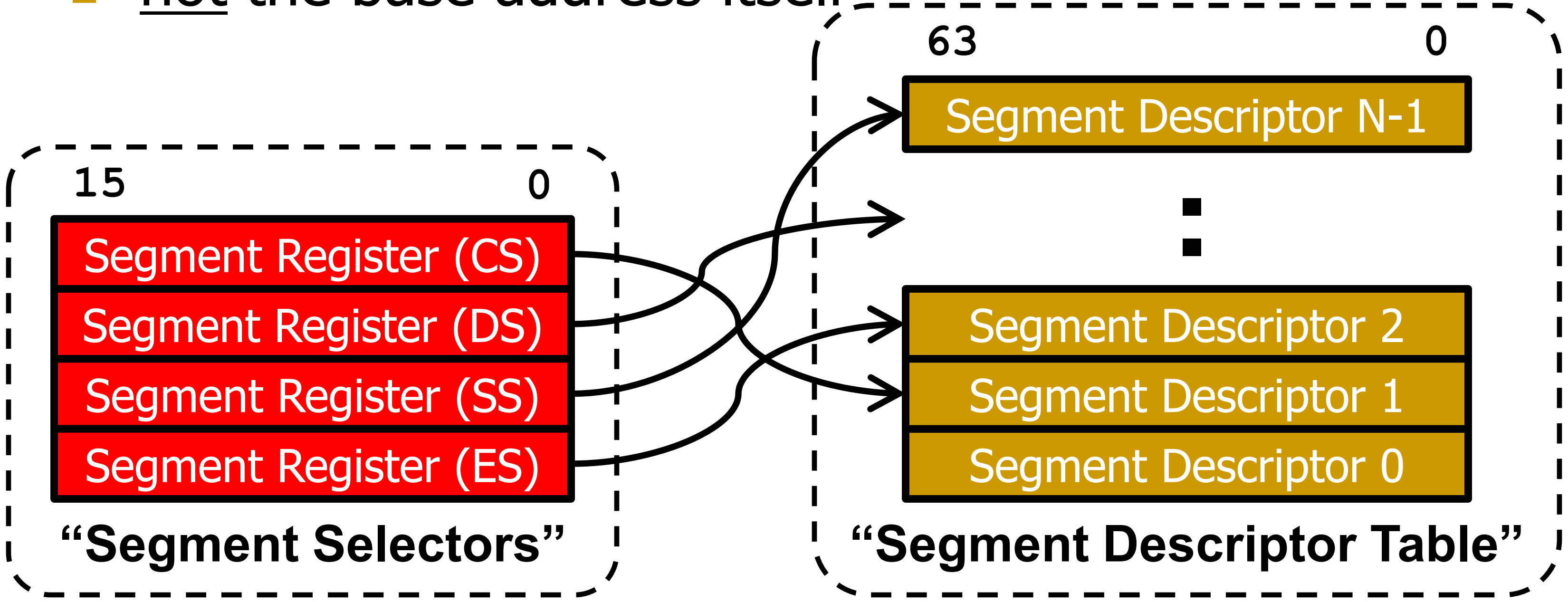
  - Can be overridden: `mov %AX,(%ES:0x1234)`

x86 Instruction

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

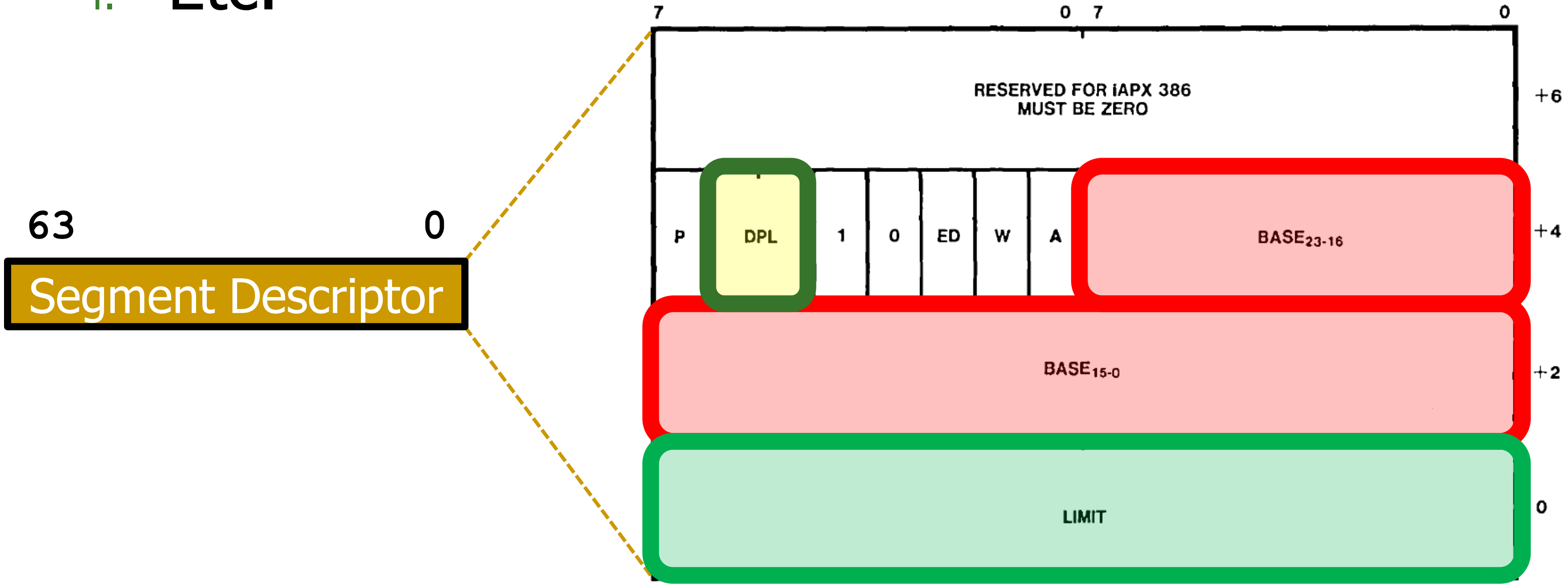
Segmentation in Intel 80286

- ***Intel 80286*** (Early 80s)
  - Still a 16-bit processor
  - Still has 4 segment registers that …
    - stores the ***index*** into a table of base addresses
    - <u>not</u> the base address itself



Intel 80286: Segment Descriptor

- A ***segment descriptor*** describes a segment:
  1. **BASE**: Base address
  2. **LIMIT**: The size of the segment
  3. **DPL**: Descriptor Privilege Level (!!)
  4. Etc.



Issues with Segmentation

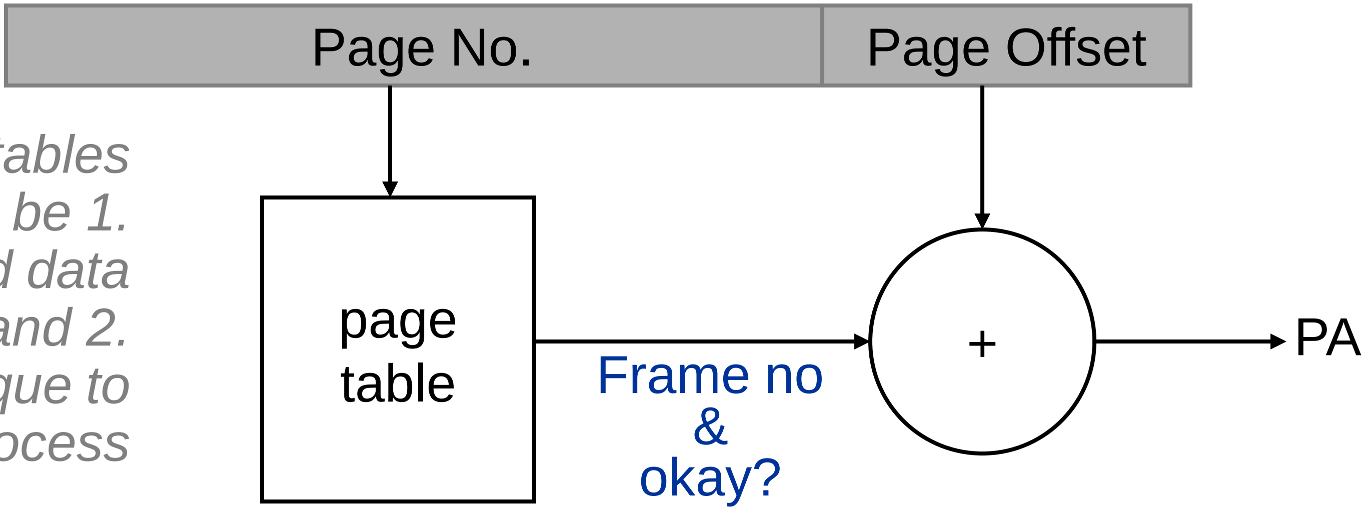
- Segmented addressing creates fragmentation problems:
  - a system may have plenty of unallocated memory locations
  - they are useless if they do not form a contiguous region of a sufficient size

- Page-based virtual memory solves these issues
  - By ensuring the address space is divided into fixed size "pages"
  - And virtual address space of each process is contiguous
  - The key is the use of indirection to give each process the illusion of a contiguous address space

Page-based Address Space

- In a Paged Memory System:
- PA space is divided into fixed size "segments" (e.g., 4kbyte),
  more commonly known as "page frames"
- VA is interpreted as page number and page offset

page tables must be 1. privileged data structures and 2. private/unique to each process



Fast Forward to Today (2014)

- Modern x86 Machines
  - 32-bit x86: Segmentation is similar to 80286
  - 64-bit x86: Segmentation is not supported *per se*
    - Forces the `BASE=0x0000000000000000`
    - Forces the `LIMIT=0xFFFFFFFFFFFFFFFF`
    - But `DPL` is still supported
- Side Note: Linux & 32-bit x86
  - Linux does not use segmentation *per se*
    - For all segments, Linux sets `BASE=0x00000000`
    - For all segments, Linux sets `LIMIT=0xFFFFFFFF`
  - Instead, Linux uses segments for privilege levels
    - For segments used by the kernel, Linux sets DPL = 0
    - For segments used by the applications, Linux sets DPL = 3

Other Issues

- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data