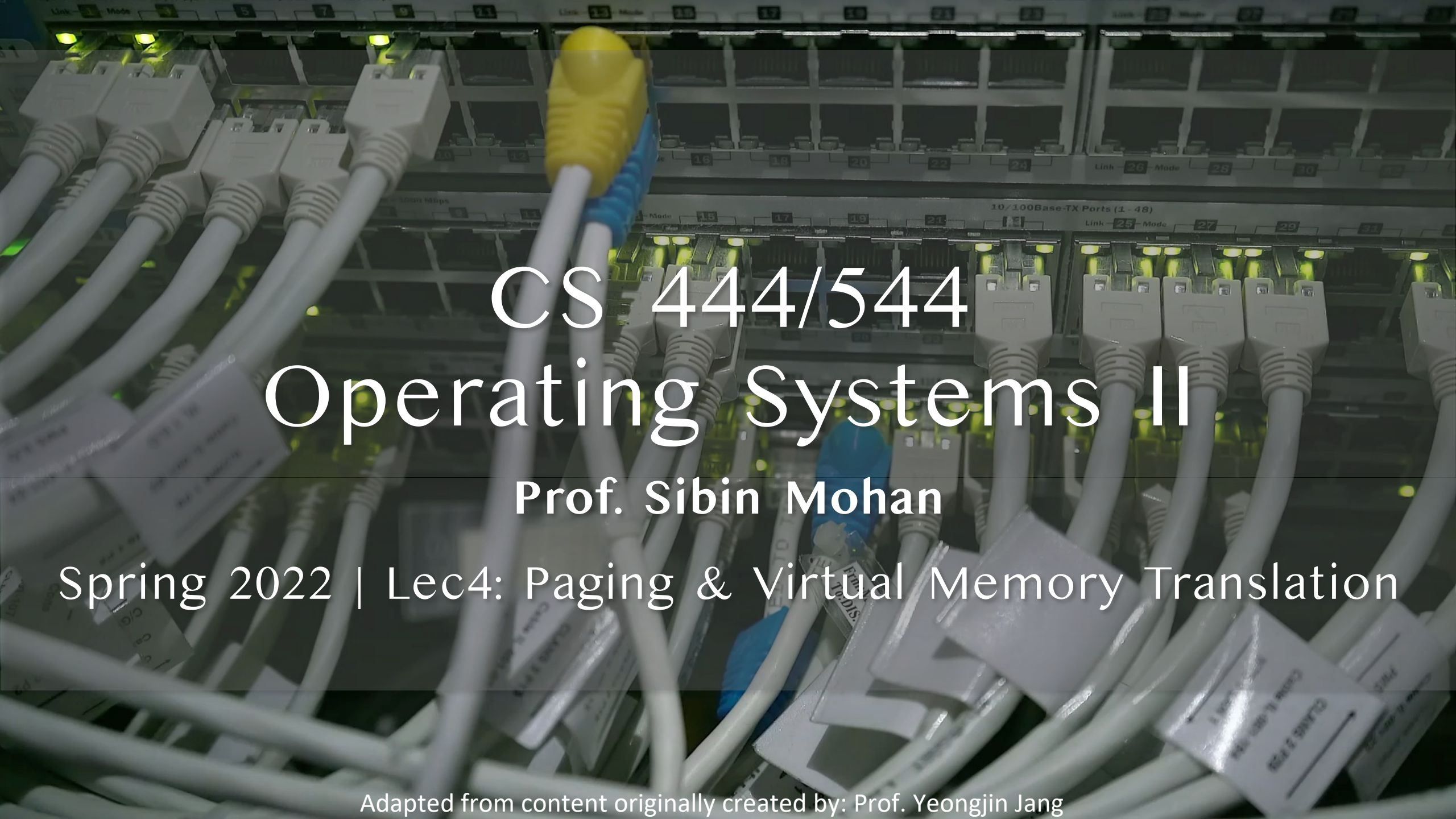


CS 444/544 Operating Systems II

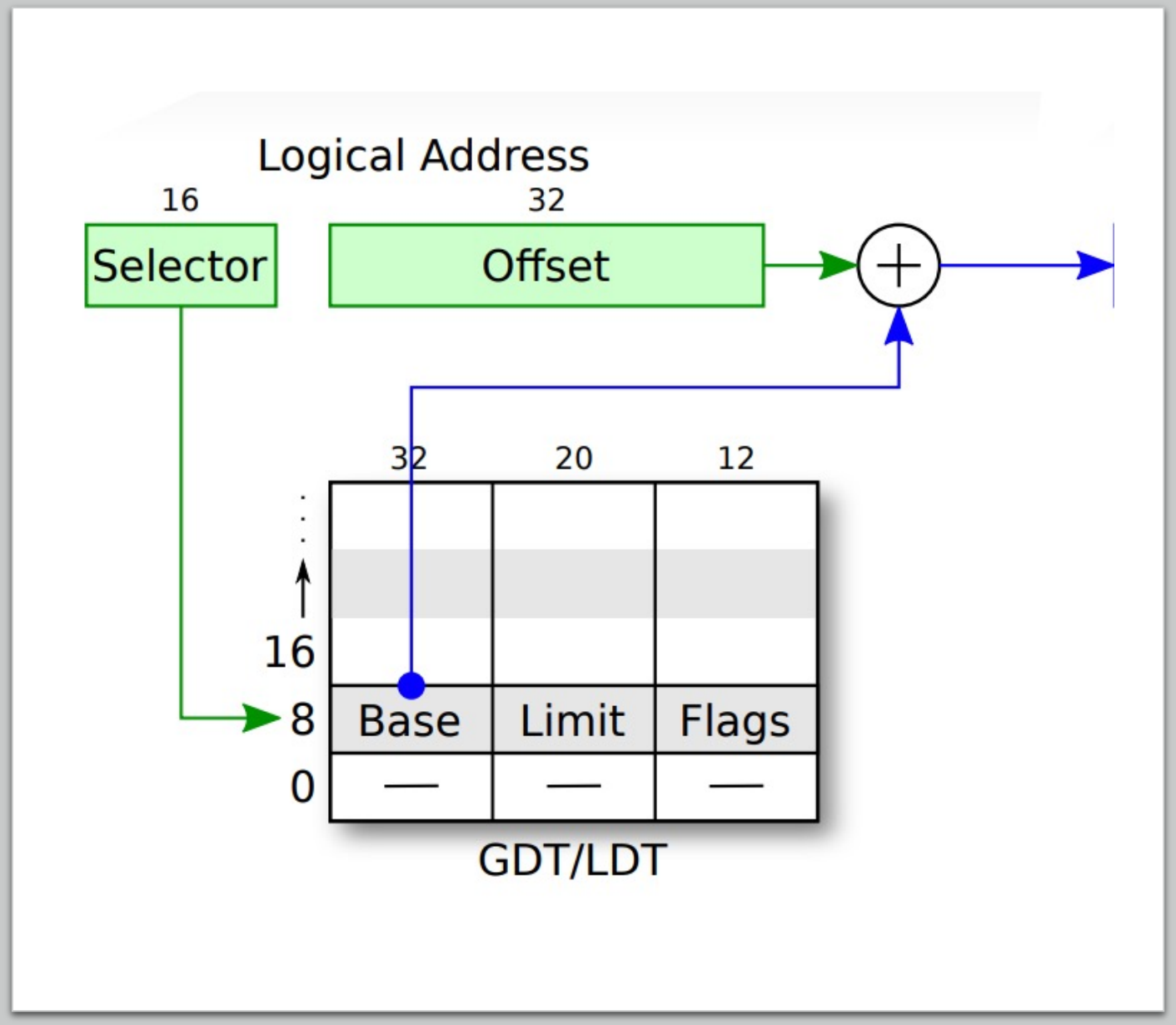
Prof. Sibin Mohan

Spring 2022 | Lec4: Paging & Virtual Memory Translation

Adapted from content originally created by: Prof. Yeongjin Jang

Protected Mode **Summary**

- Segment access via **GDT**
  - Base + Offset if Offset < Limit * 4096 if G == 1)
  - Base + Offset if Offset < Limit (if G == 0)
- Last two bits in `%cs` → CPL
  - Memory Privilege → Ring level
  - 0 for OS kernel
  - 3 for user applications
- Changing CR0 to enable protected mode
  - `CR0_PE_ON` == 1, set via eax
- Changing CPL?
  - `ljmp %cs:xxxxx`
  - set the last 2 bits of %cs as 0 for kernel, 3 for user

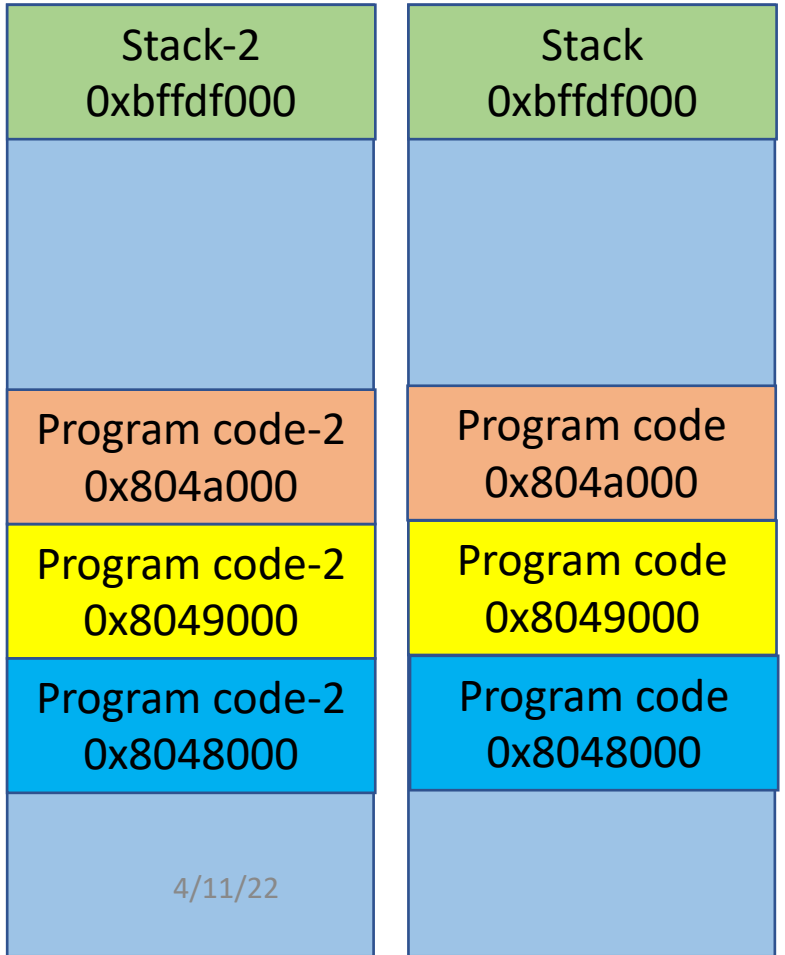


Three Goals | Details

- **Transparency**: programs shouldn't need to know system's **internal state**
  - Program A is loaded at **0x8048000**. Can Program B be loaded at **0x8048000**?
- **Efficiency**: do not waste memory; avoid **memory fragmentation**
  - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its allocation?
- **Protection**: **isolate** program's execution environment
  - Can we prevent an overflow from Program A from overwriting Program B's data?

Paging: Virtual Memory

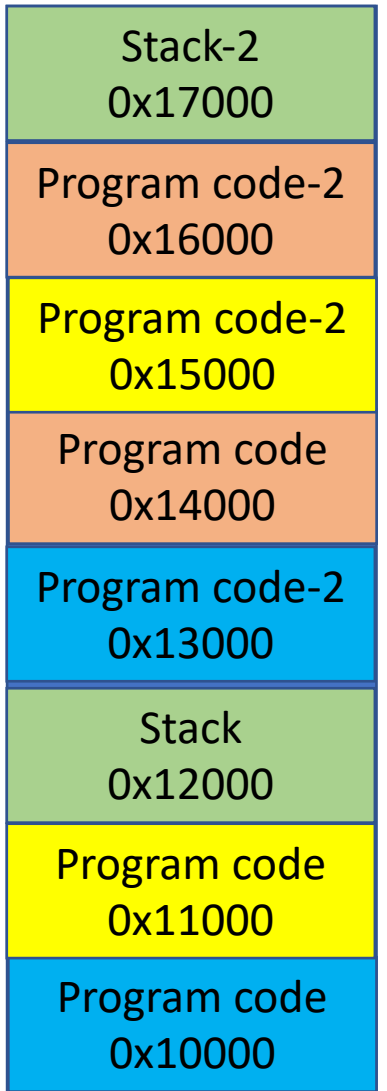
- Uses an **indirect table** that **maps virt-addr → phys-addr**



Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

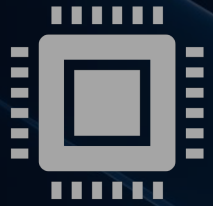
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

Physical Memory



Paging

A method to implement virtual memory



Split memory into multiple blocks [E.g. 4096 bytes, 12-bi])

Last 3 digits of page address are ZERO [hex]

E.g., 0x0, 0x1000, 0x2000, …, 0x8048000, 0x804a000, …, 0x7fffe000, etc.

An indirect map between virtual page and physical page

Set arbitrary virtual address for a page

e.g., 0x81815000

Set physical address to that page as a map

e.g., 0x32000

0x81815000 ~ 0x81815fff → 0x32000 ~ 0x32fff

Page Table

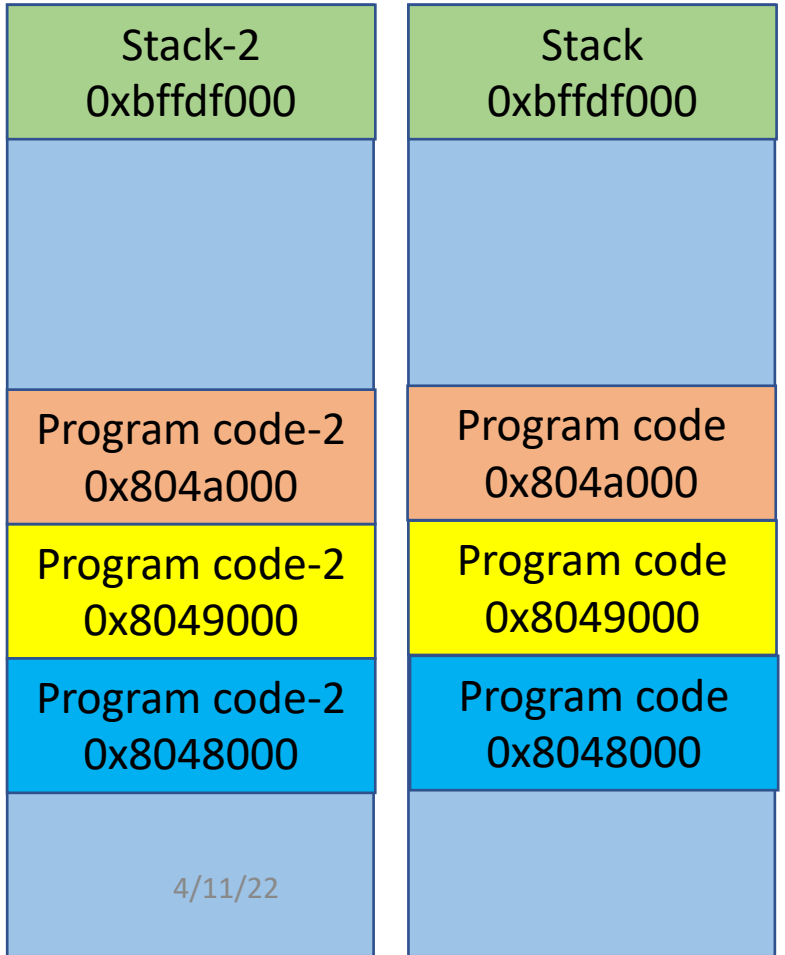
- A table that **stores virtual address to physical address mapping**
- Created for **each process**

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

Paging: Virtual Memory

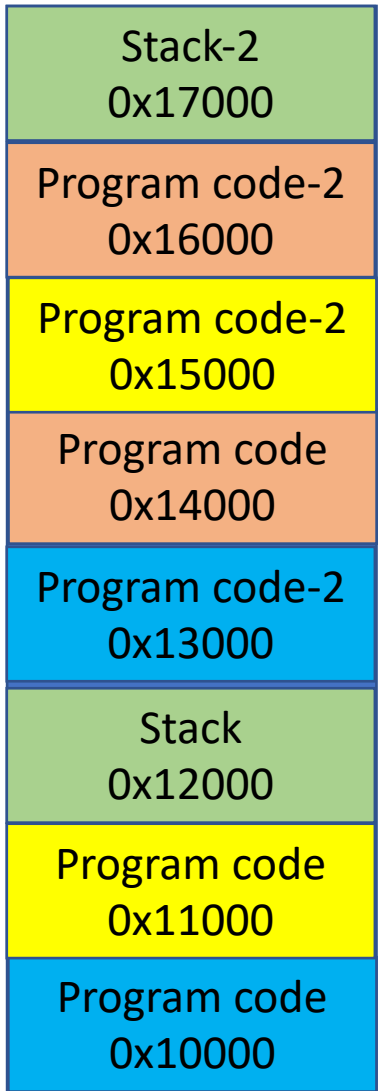
- Uses an **indirect table** that **maps virt-addr → phys-addr**



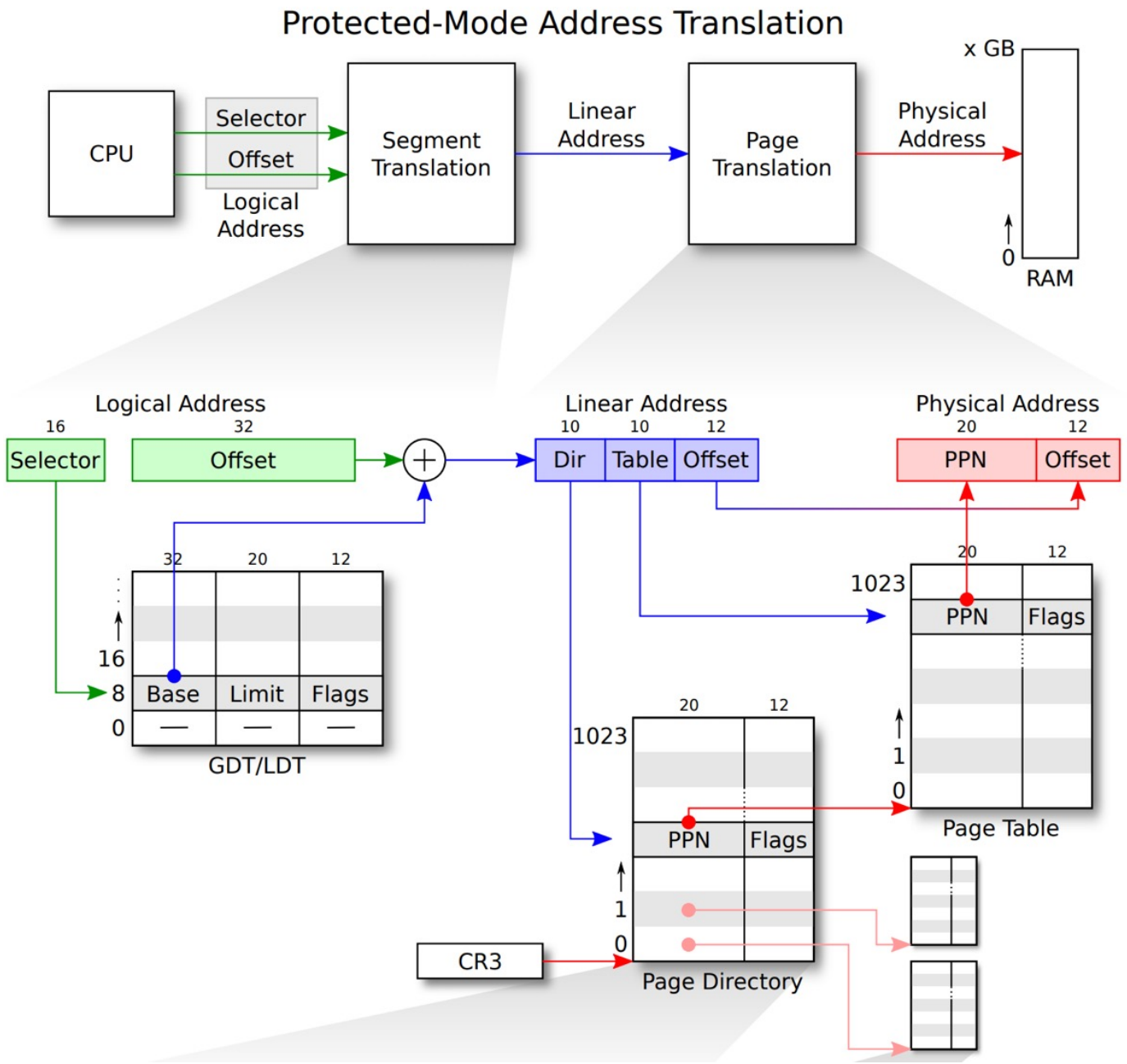
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

Physical Memory

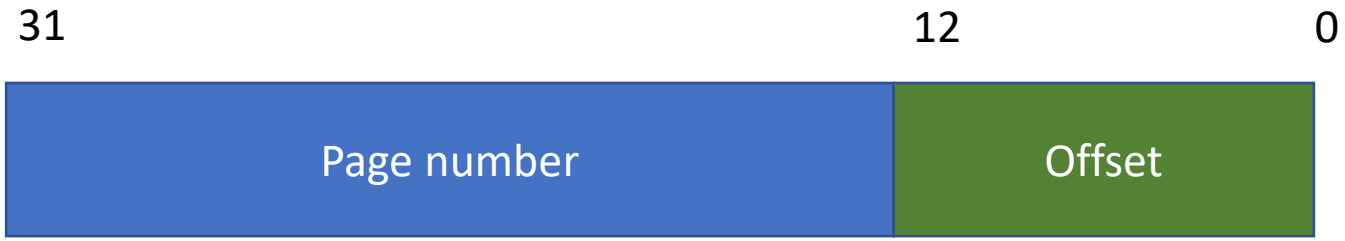


x86 Memory Access



Page Directory / Table

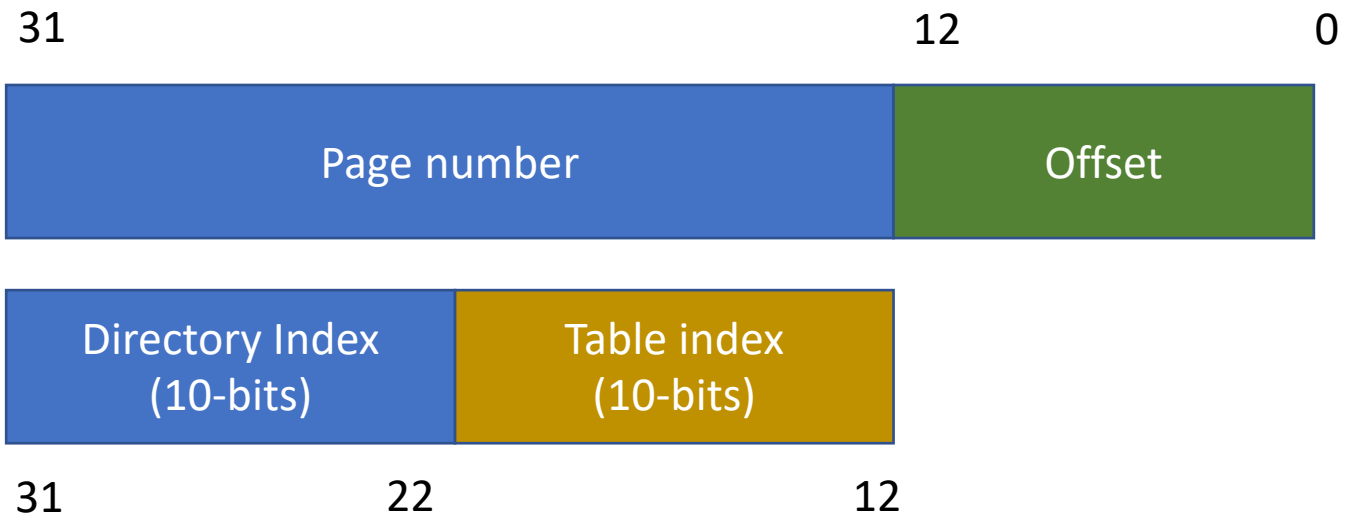
- We access page table by virtual address
- Page size: 4 KB (12bits)
- Page number: 20 bits



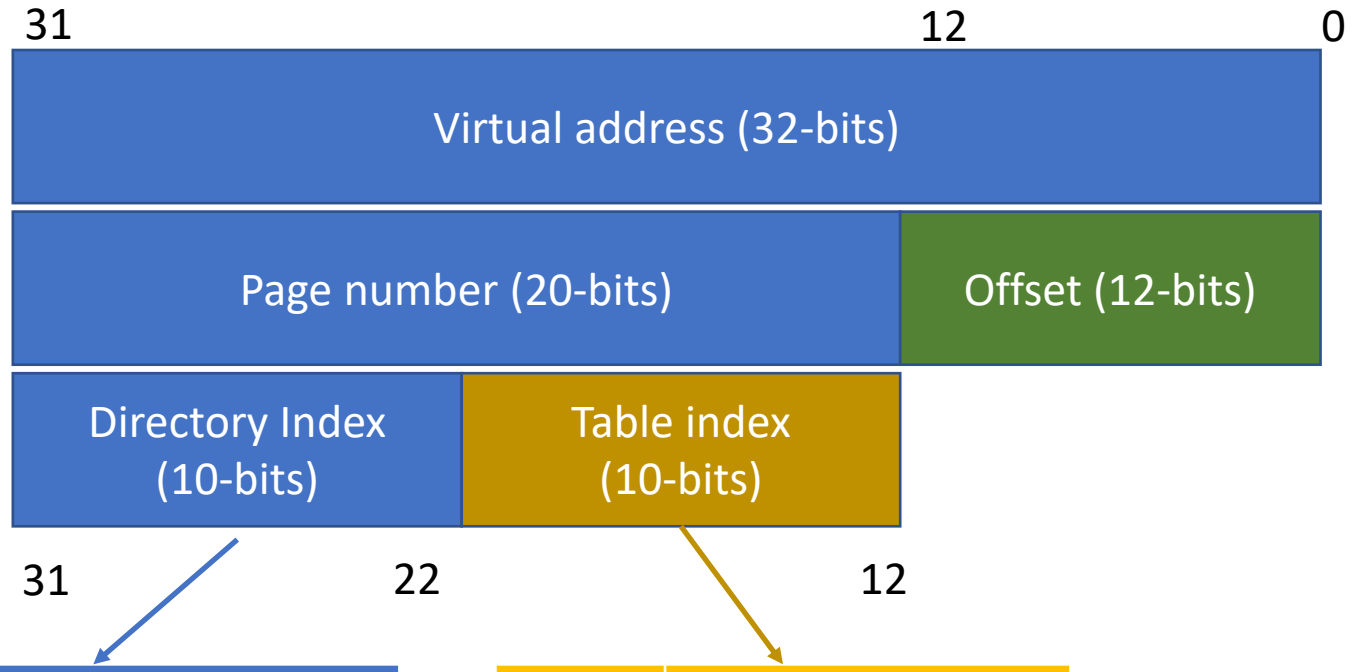
- What is the page number and offset of
  - 0x**8048000**
  - 0x**b7ff3100**

Page Directory / Table

- In x86 (32-bit), CPU uses 2-level page table
- 10-bit directory index
- 10-bit page table index
- 12-bit offset



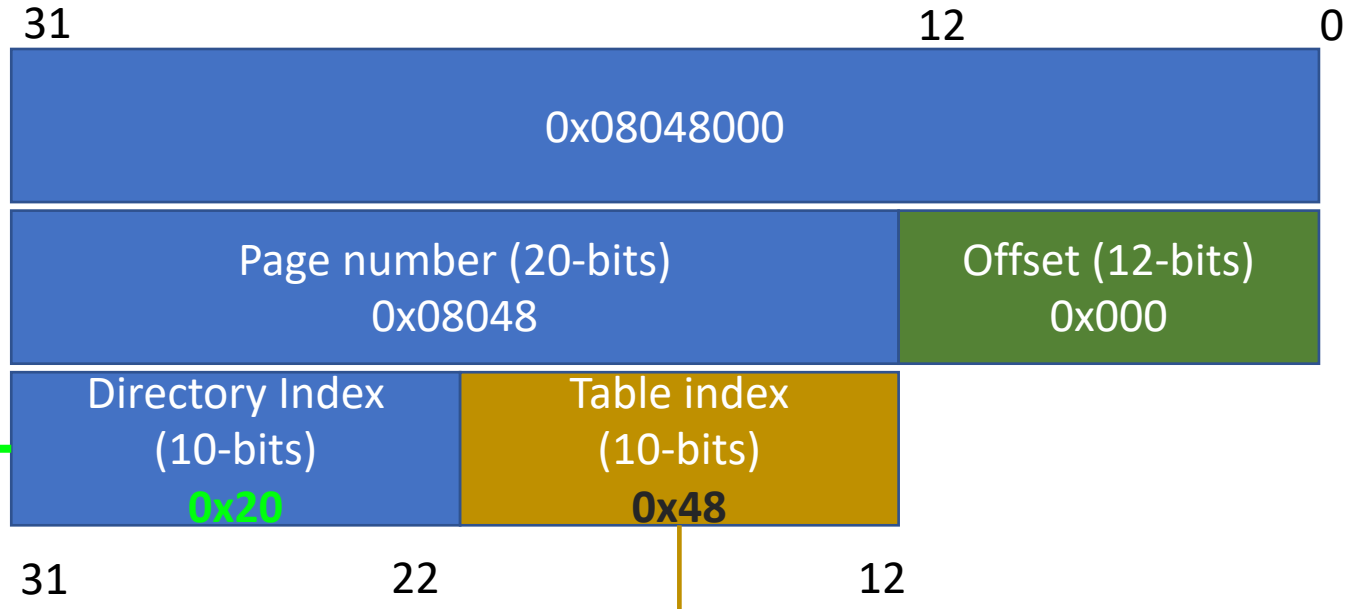
Address Translation



	Page Directory Entry
0	Addr PT
1	Addr PT
2	Addr PT
1024	Addr PT

	Page Table Entry
0	Addr page
1	Addr page
2	Addr page
1024	Addr page

Address Translation Example



	Page Directory Entry
0	Addr PT1
..	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Physical access

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Page Table Entry

- Address translation is from virtual page to physical page
  - E.g., 0x8048000 -> 0x11000
- We do not need to translate lower 12 bits
  - E.g., 0x8048001 -> 0x8048000 + 0x001 -> 0x11000 + 0x001

- So page table entry only uses higher 20 bits to store physical page address
  - **Lower 12 bits** are remaining as the same
  - We **do not** translate the lower 12 bits!

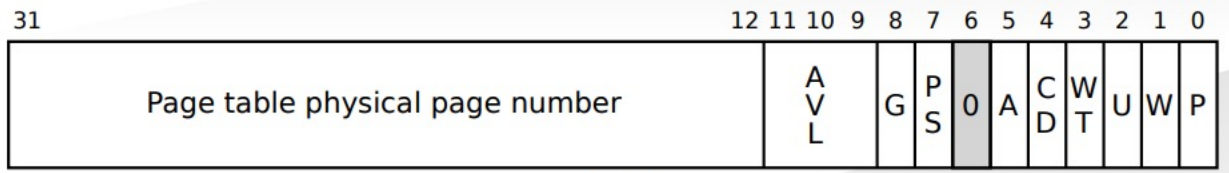
Address Translation Examples

- Virtual addresss 0x8048338
  - Virtual page number: 0x8048
  - Offset: 0x338
  - Physical page number: 0x10
  - Physical address: 0x10 (000) + 0x338 = 0x10338
- Virtual address 0x443325af
  - Virtual page number: 0x44332
  - Offset: 0x5af
  - Physical page number: 0x33885
  - Physical address: 0x33885000 + 0x5af = 0x338855af

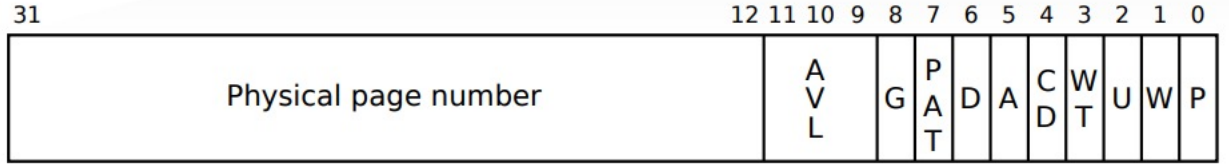
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
0x44332000	0x33885000

Virtual	Physical
0x8048	0x10
0x8049	0x11
0x804a	0x14
0xbffdf	0x12
0x44332	0x33885

PDE, PTE



PDE



PTE

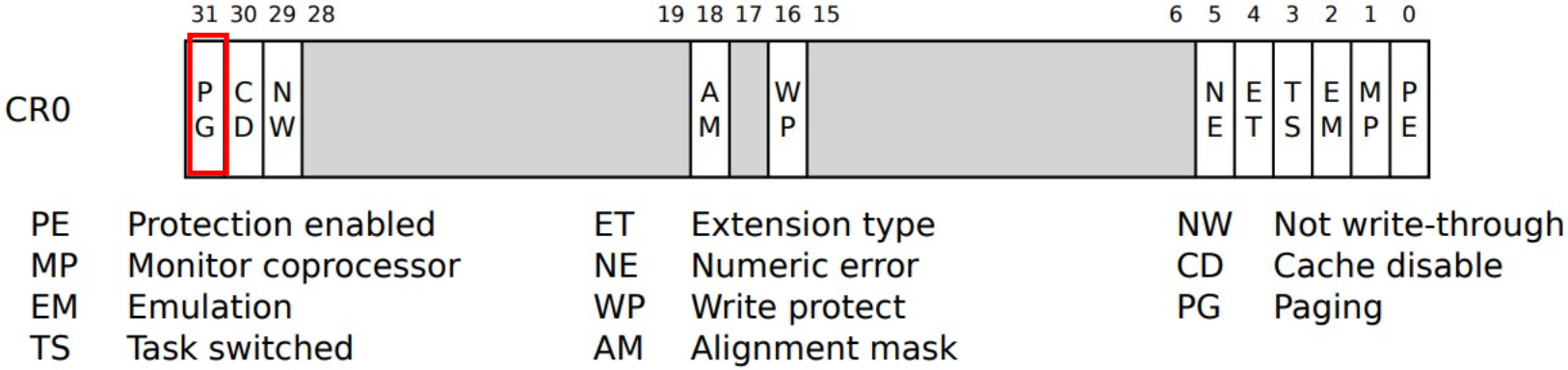
P	Present
W	Writable
U	User **0: kernel, 1: user**
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

	Page Directory Entry
0	Addr PT1
..	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

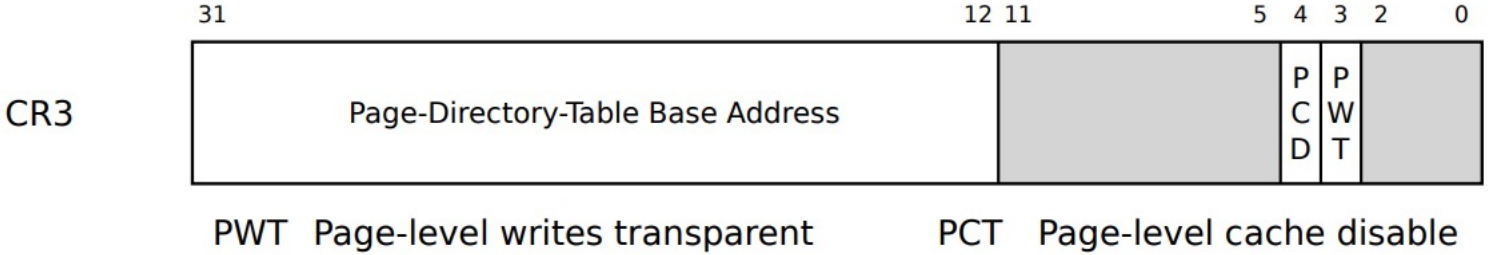
PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Paging in x86

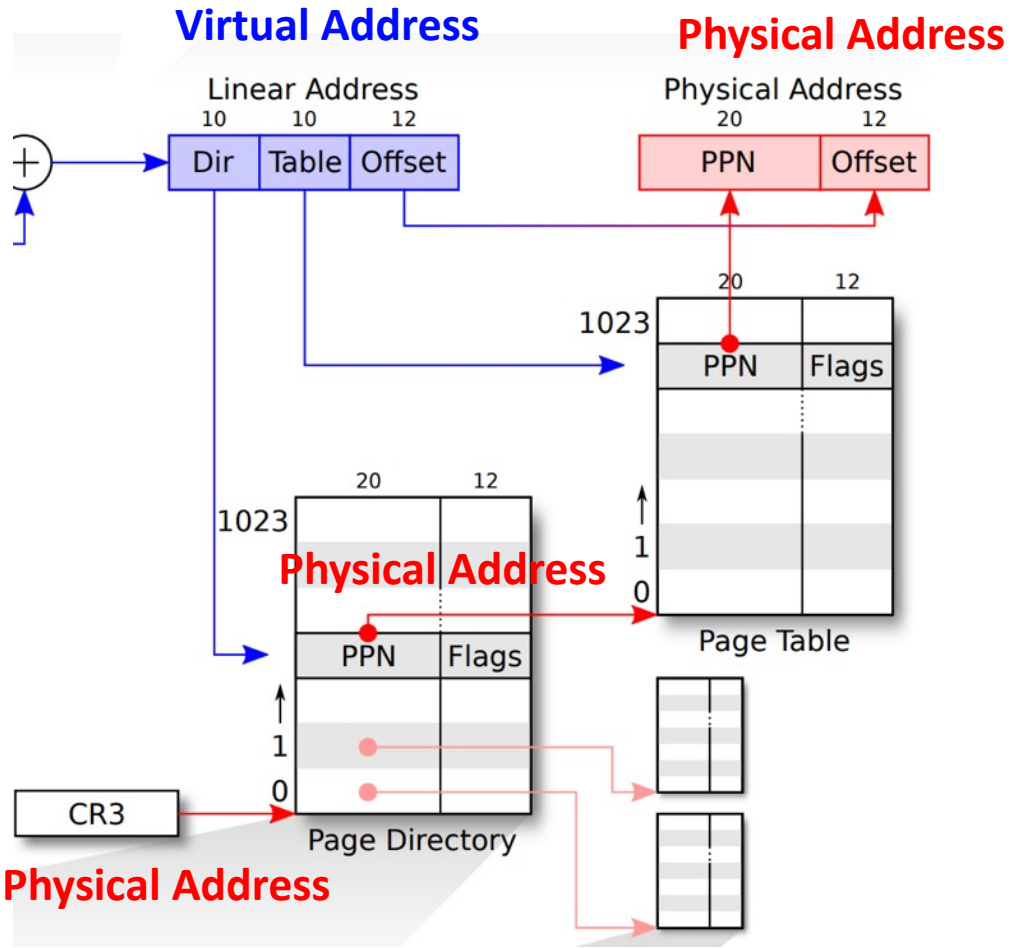
- Can be enabled via CR0



- Page table address (physical) need to be stored at CR3



Paging in x86



Paging in x86

- In JOS (kern/entry.S)

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

Set cr3 to point the address of page directory

Turn on CR0_PG!

- After CR0_PG is up (paging is turned on)
  - All address access regarded as virtual address

How can we access physical address

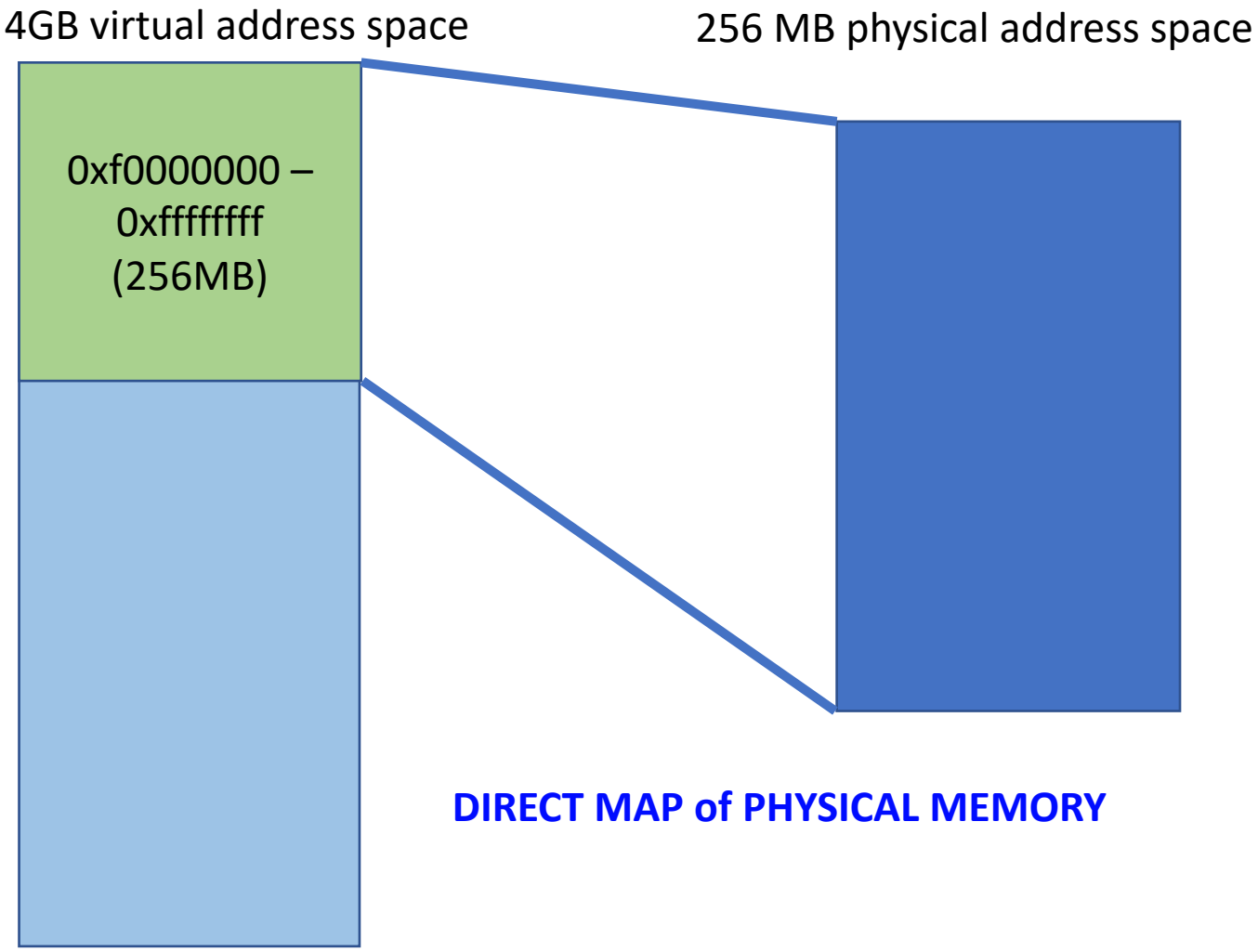
- After CR0_PG is up (paging is turned on)
  - All normal address access regarded as virtual address
    - Abnormals? CR3 will regard the address as physical
- What if we would like to access the physical address 0x100010?
  - Our kernel is at physical address 0x100000 ~ 0x110000
- What's the virtual addresses of those area?

How can we access physical address

- In JOS, we will map 0xf0000000 ~ 0xffffffff to
  - Physical address
  - 0x00000000 ~ 0x0fffffff
  - 256 MB Region

- 0xf0100010 -> 0x00100010

- 0xf1333358 -> 0x01333358



In kern/entrypgdir.c

```
__attribute__((__aligned__(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
    [KERNBASE>>PDXSHIFT]
        = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
};
```

Entry_pgdir only contains two entries;
0 ~ 0x400000 to 0 ~ 0x400000 (not writable)
0xf0000000 ~ 0xf0400000 to 0 ~ 0x400000 (writable)

In kern/entrypgdir.c

0xf0001000 will consult entry_pgtable[1]

-> 0x001000 | PTE_P | PTE_W

Phyaddr: 0x1000

```
__attribute__((__aligned__(PGSIZE)))
pte_t entry_pgtable[NPTENTRIES] = {
    0x000000 | PTE_P | PTE_W,
    0x001000 | PTE_P | PTE_W,
    0x002000 | PTE_P | PTE_W,
    0x003000 | PTE_P | PTE_W,
    0x004000 | PTE_P | PTE_W,
    0x005000 | PTE_P | PTE_W,
    0x006000 | PTE_P | PTE_W,
    0x007000 | PTE_P | PTE_W,
    0x008000 | PTE_P | PTE_W,
    0x009000 | PTE_P | PTE_W,
    0x00a000 | PTE_P | PTE_W,
    0x00b000 | PTE_P | PTE_W,
    0x00c000 | PTE_P | PTE_W,
    0x00d000 | PTE_P | PTE_W,
    0x00e000 | PTE_P | PTE_W,
    0x00f000 | PTE_P | PTE_W,
    0x010000 | PTE_P | PTE_W,
    0x011000 | PTE_P | PTE_W,
```

Page

- A page
  - A 4,096 (4 KB) block of memory
- Why having a block?
- How large does the page table should be to map 4GB (32-bit space)?
  - If a block is 1GB – 4 entries * pointer (32-bit, 4byte) = 16 byte long
  - If a block is 4MB – 1024 entries * pointer = 4,096 bytes, 4KB long
  - If a block is 4KB – 1048576 entries * pointer = 4MB long
  - If a block is 1 byte – 4294967296 entries = 4GB… NONO…

Design consideration:
Size of page table matters!

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Page (cont'd)

- A page
  - A 4,096 (4 KB) block of memory
- Why 4KB in Intel x86?
- How much memory do you need to allocate 1 byte?
  - 1GB page – 1GB
  - 4MB page – 4MB
  - 4KB page – 4KB
  - 1B page – 1B

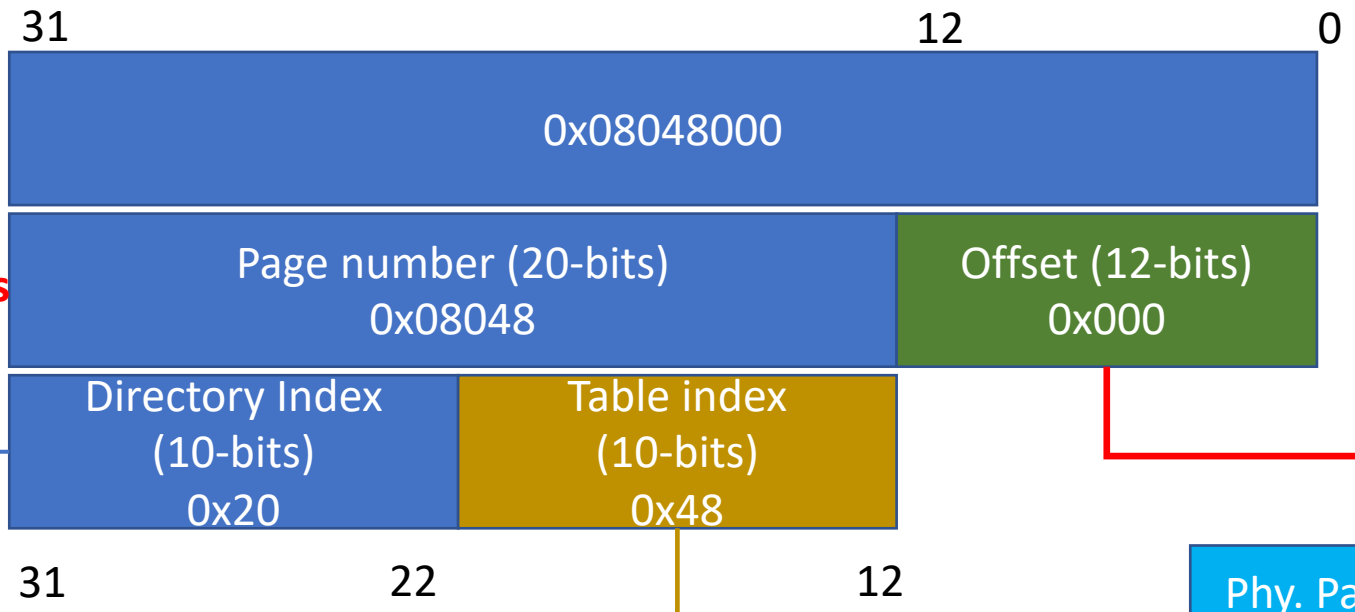
Design consideration:
Memory fragmentation matters!

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Page Size / Page Table Size

- If a page size is too small, it requires a big page table
  - 1B, 4GB
  - 4KB, 4MB
  - 4MB, 4KB
  - 1G, 16B
- If a page size is too big, unused memory in a page will be wasted
  - 1B - 1B (no waste)
  - 4KB – 1B
  - 4MB – 1B
  - 1G – 1B

Recap – Page Table & Addr Translation



Mem access #1

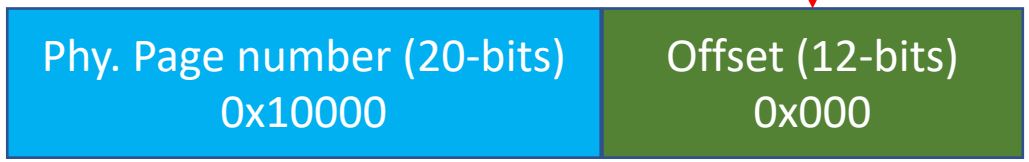
CR3[0x20]

Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

PDE[0x48]
Mem access #2

	Page Table Entry
0	Addr PT
0x48	0x10000
0x49	0x11000
0x4a	0x50000



Mem access #3 (required)

Page Table Lookup - Caching

- Access example
  - movl 0x12345678, %eax
- 3 memory access per each memory access – SLOW!
  - 2 additional accesses due to page table lookup
  - mov instruction – takes 1 cycle
  - memory access – takes ~200 cycles…
  - 200 (access the address) + 1 (mov) + 200 * 2 (page table…) = 601 cycles..
- CPU caches Address Translation
  - Translation Lookaside Buffer (TLB)
  - Reduce these additional accesses -> **Speed up!**

Translation Lookaside Buffer (TLB)

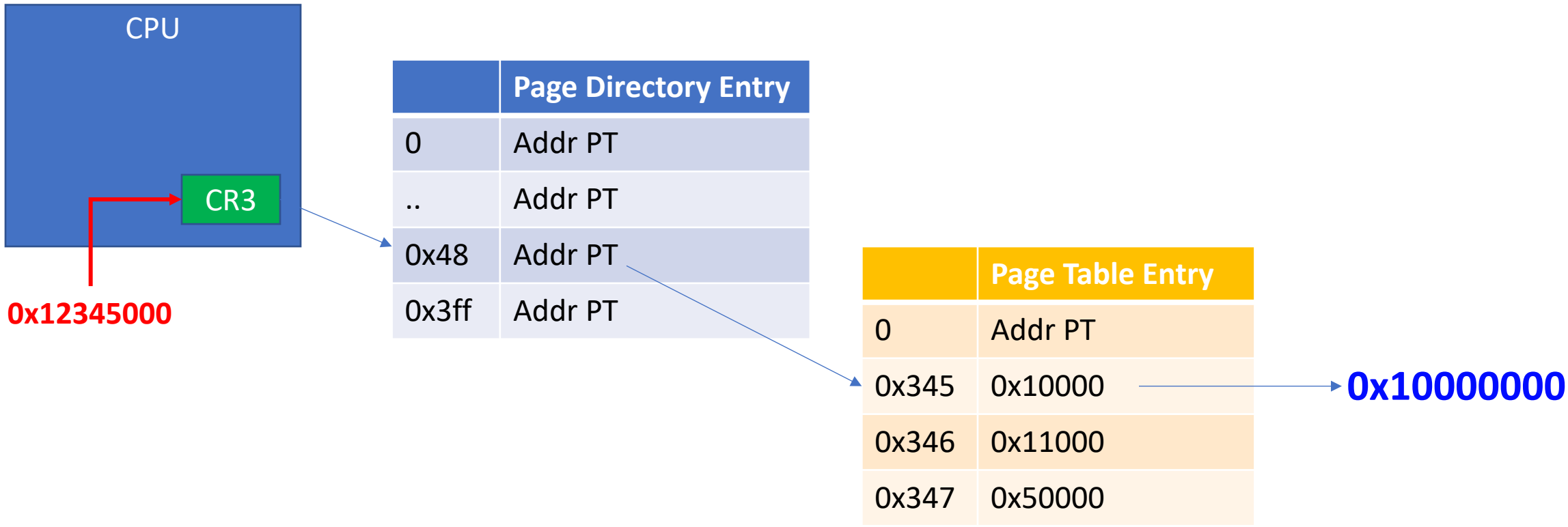
- Stores VA-PA mappings and cache them!

VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

0x12345678 -> 0x678
0x12346678 -> 0x5678
0x12347678 -> 0xff678
0x12348678 -> 0xfff678

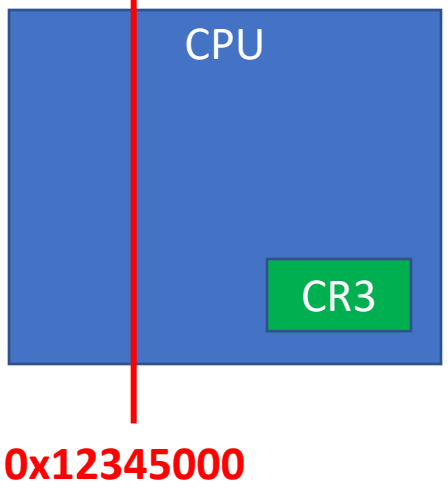
- Benefits
  - Do not have to walk down page tables for cached entries

CPU that does not have TLB



CPU with TLB

VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1



0x12345000

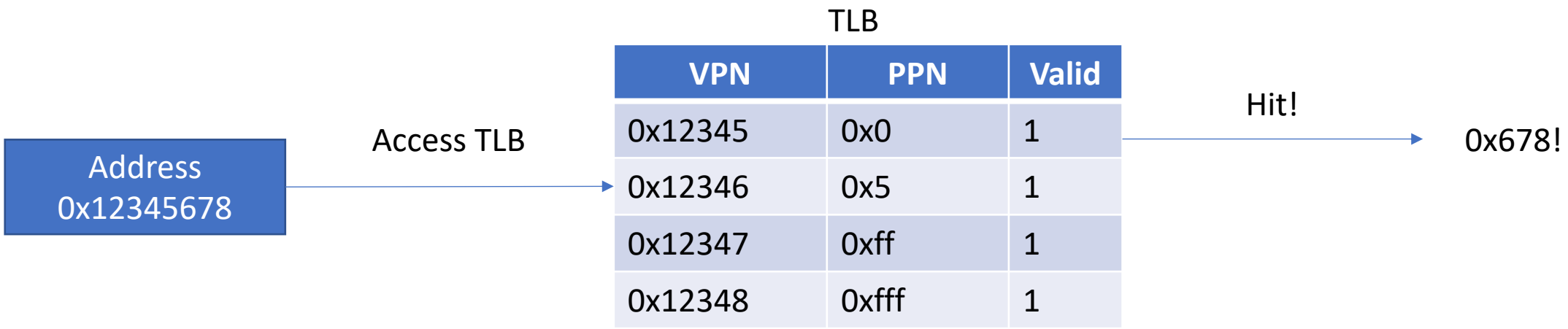
	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x48	0x10000
0x49	0x11000
0x4a	0x50000

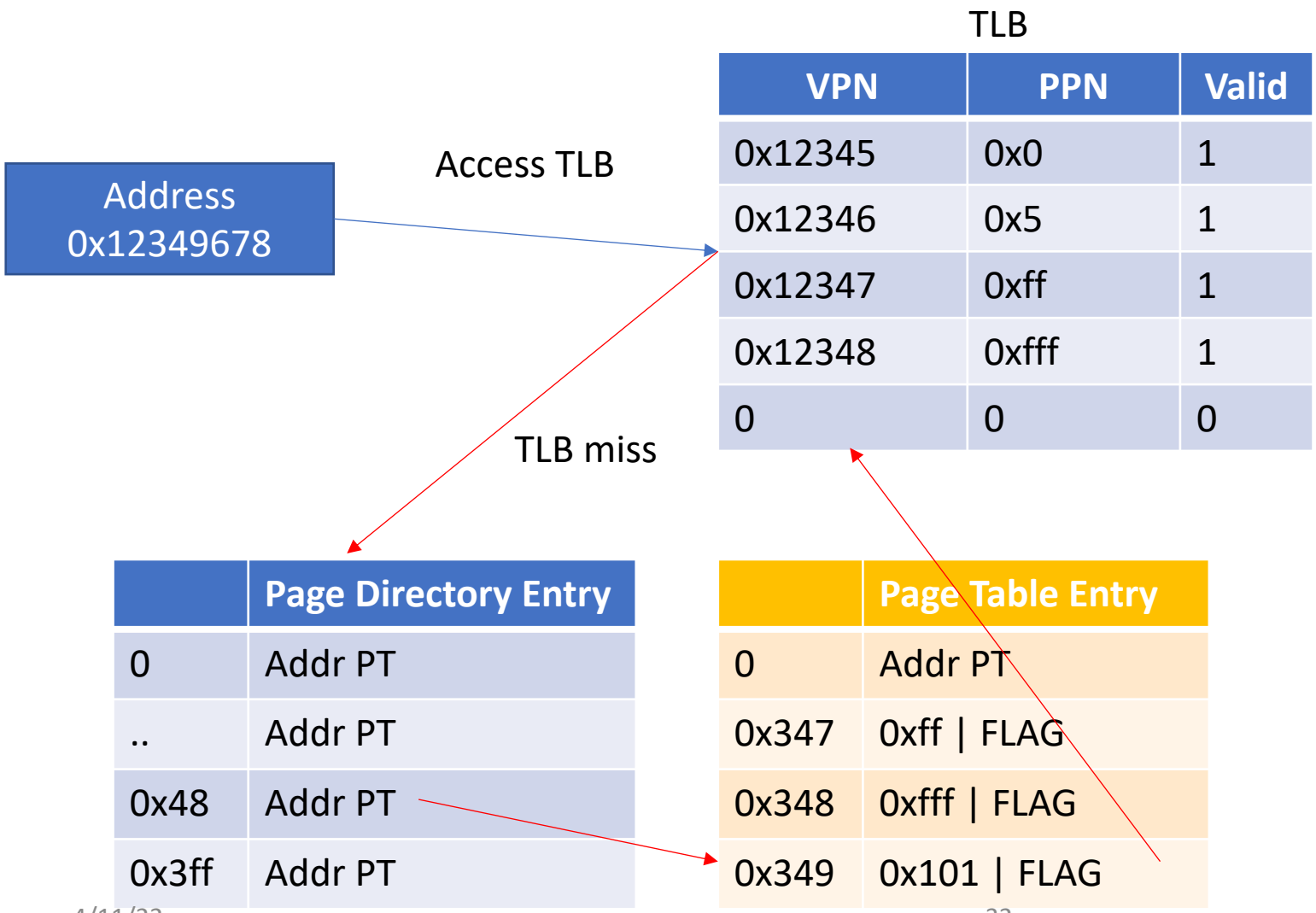
0x10000000

No page table access..

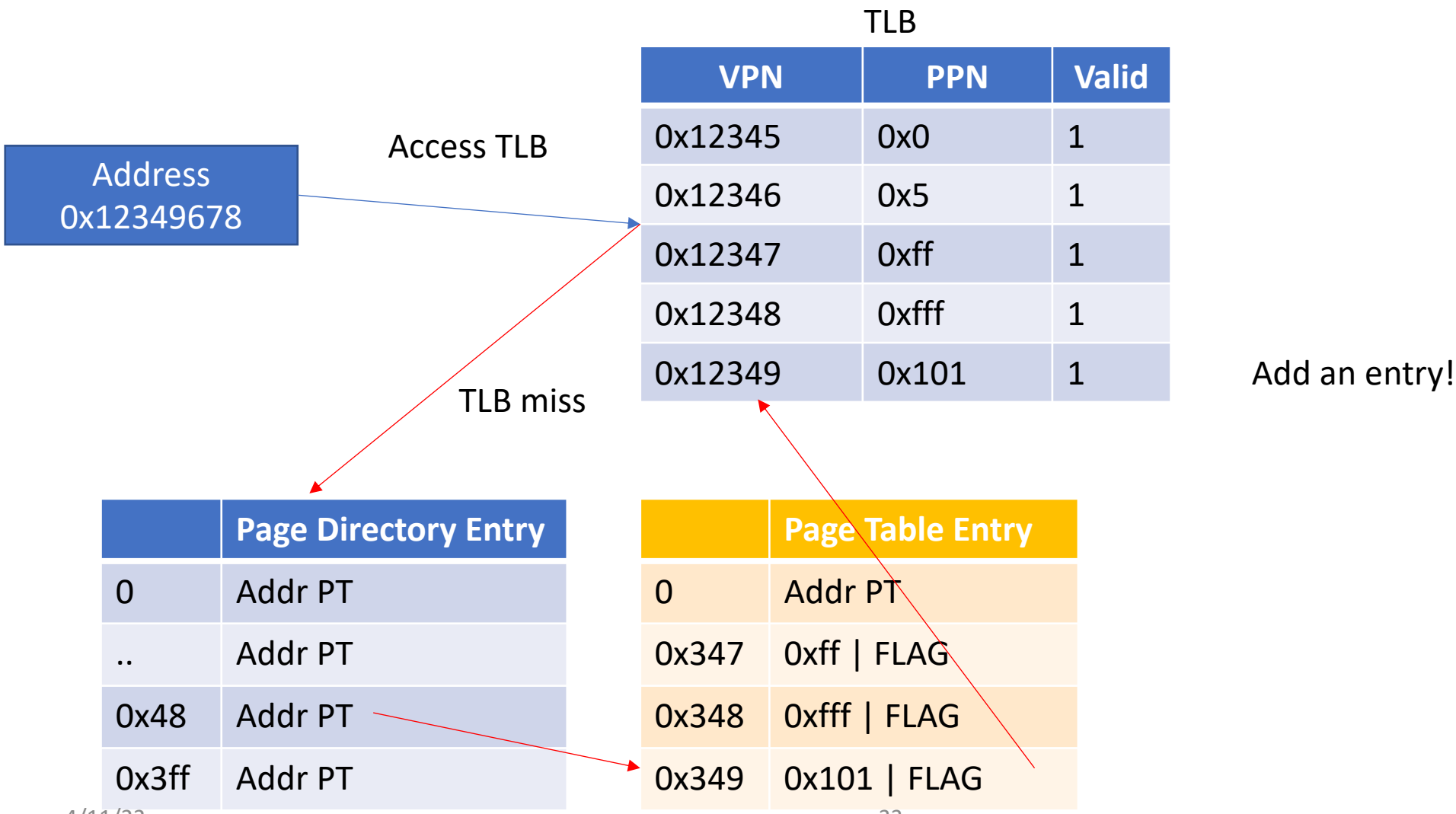
Address Translation with TLB (hit)



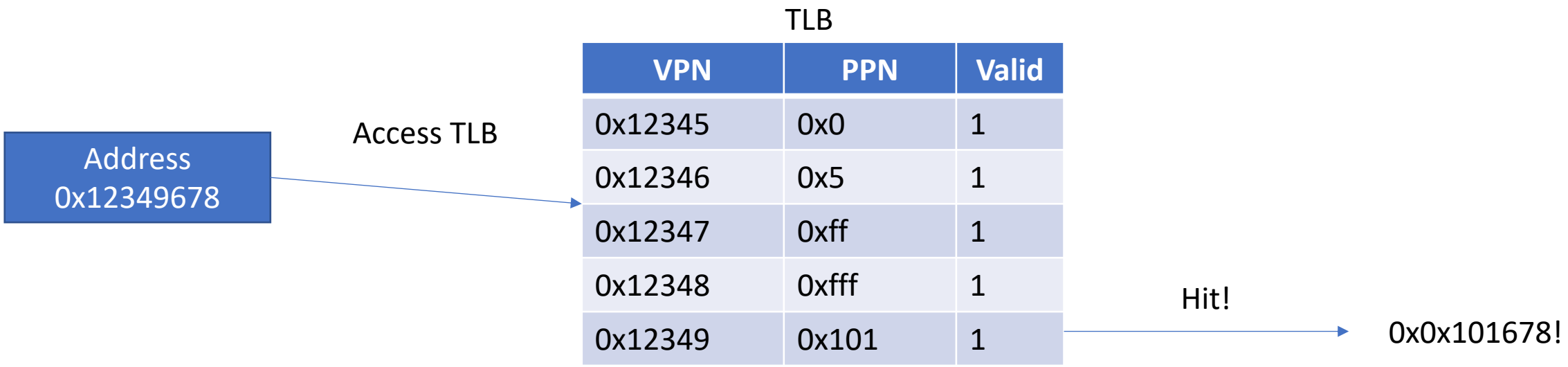
Address Translation with TLB (miss)



Address Translation with TLB (miss)



Address Translation with TLB (hit)



	Page Directory Entry
0	Addr PT
..	Addr PT
0x48	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x347	0xff | FLAG
0x348	0xfff | FLAG
0x349	0x101 | FLAG

Why do we have TLB?

- Core i7-6700K (Skylake, 4.00 GHz)
  - Data TLB L1: 64 items. 4-way. Miss penalty = 9 cycles. Parallel miss: 1 cycle per access
  - Data TLB L2 (STLB): 1536 items. 12-way. Miss penalty = 17 ? cycles. Parallel miss: 14 cycles per access
  - PDE cache = ? items. Miss penalty = ? cycles.

```
Size        Latency       Increase    Description

32 K        4
```

- TLB hit requires 4 cycles, 1ns!

Why do we have TLB?

- TLB hit requires 4 cycles, 1ns!

- Page table walk requires 2 memory access for translation
  - Uncached: 9 cycles + (42 cycles + 51ns) * 2
  - [TLB miss] [RAM latency]
    2ns + (10ns + 51ns) * 2 = 124ns (124 times slower…)
  - Cached: 9 + 4 * 2 = 17 cycles if all blocks cached in L1 (4 ns, 4 times slower!)

- Data TLB L1: 64 items. 4-way. Miss penalty = 9 cycles. Parallel miss: 1 cycle per access
- Data TLB L2 (STLB): 1536 items. 12-way. Miss penalty = 17 ? cycles. Parallel miss: 14 cycles per access
- PDE cache = ? items. Miss penalty = ? cycles.
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 42 cycles (core 0) (i7-6700 Skylake 4.0 GHz)
- L3 Cache Latency = 38 cycles (i7-7700K 4 GHz, Kaby Lake)
- RAM Latency = 42 cycles + 51 ns (i7-6700 Skylake)

We have limited entries in TLB

- Core i7-6700K (Skylake, 4.00 GHz)
  - Data TLB L1: 64 items. 4-way. Miss penalty = 9 cycles. Parallel miss: 1 cycle per access
  - Data TLB L2 (STLB): 1536 items. 12-way. Miss penalty = 17 ? cycles. Parallel miss: 14 cycles per access
  - PDE cache = ? items. Miss penalty = ? cycles.
- 64 items for L1 d-TLB, 1536 items for L2 d-TLB
- CPU need to schedule this cache
  - Based on Temporal/Spatial locality…