CS 444/544 Operating Systems II

Lecture 4

Paging and Virtual Memory Translation

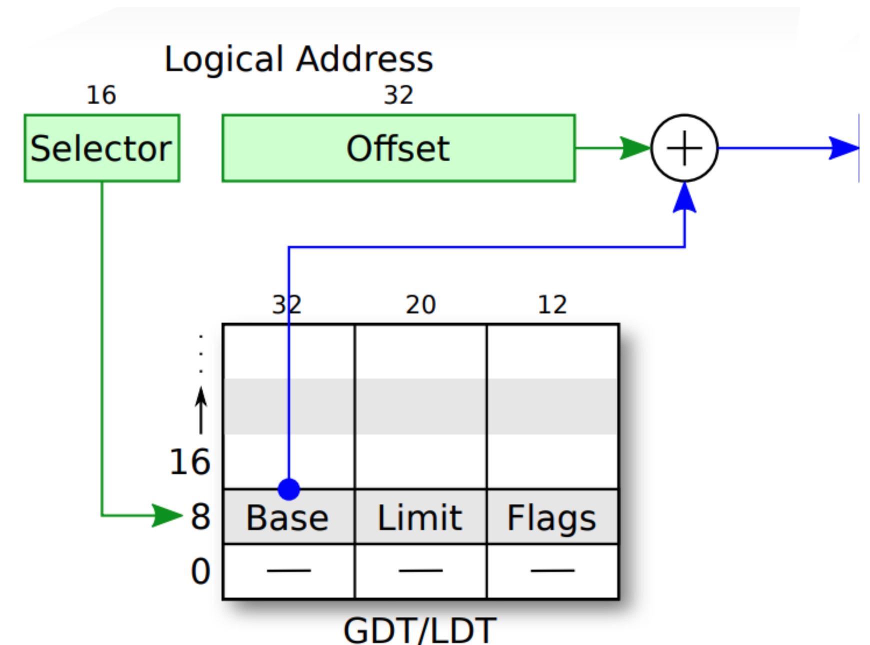
4/10/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Oregon State University

Recap: 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 (8, 16, 24 …)
  - 3 for user application (11, 19, 27, …)
- 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
  - Can get CPL down (0→0,1,2,3 but not 3→0,1,2)
  - We will rely on syscall/trap/exceptions (Week 5, Lab 3) to switch privilege from user (3) to kernel (0)

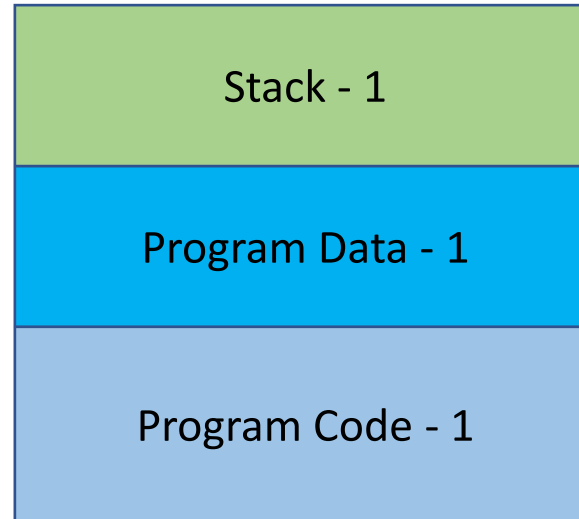


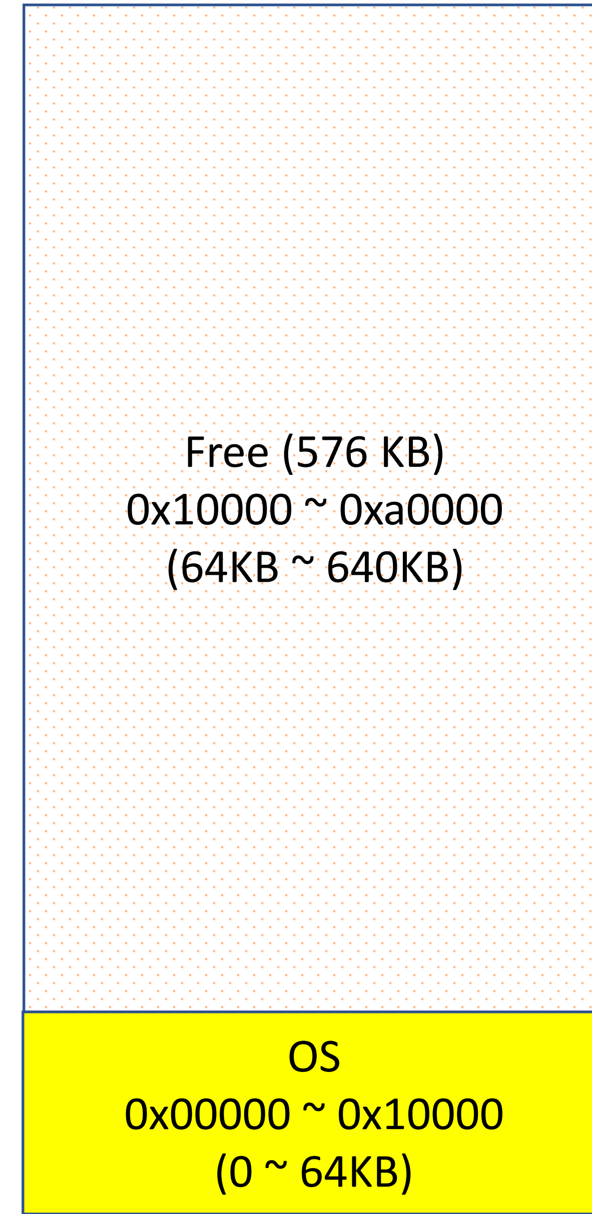
Virtual Memory

- Three goals
  - Transparency
  - Efficiency
  - Protection

Uniprogramming Environment

- Run one program
- The program can use memory space freely…

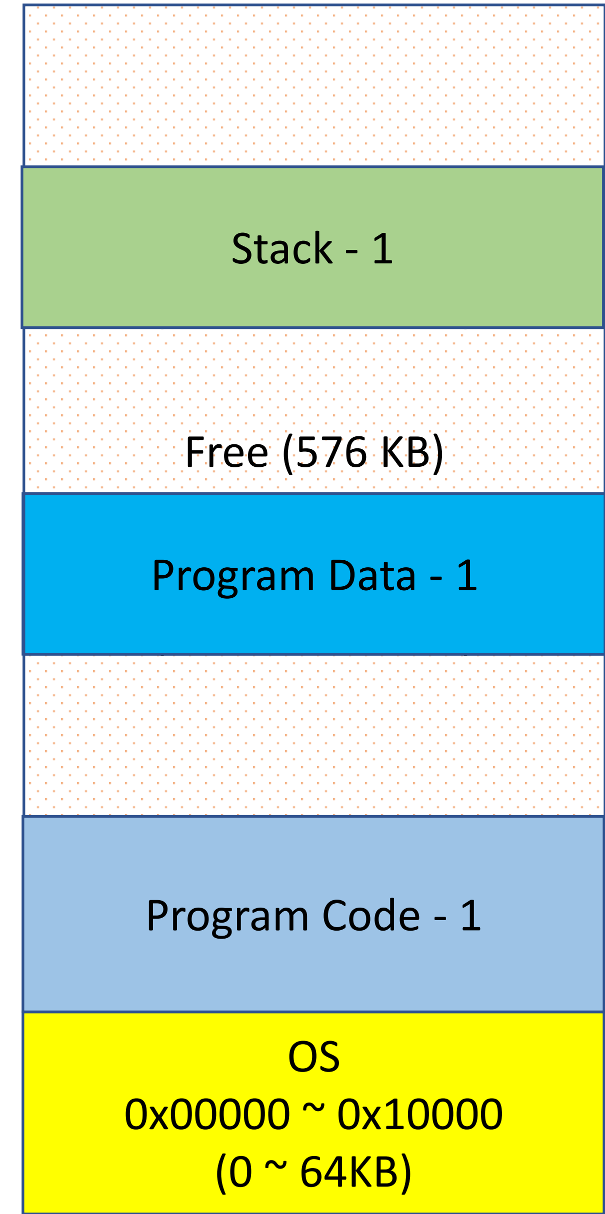




Uniprogramming Environment

- Run one program

- The program can use memory space freely…



Uniprogramming Environment

- Run one program
- The program can use memory space freely…

Free (64 KB)
0x90000 ~ 0xa0000
(576KB ~ 640KB)

Stack - 1 (64KB)
0x80000 ~ 0x90000
(512KB ~ 576KB)

Free (192 KB)
0x50000 ~ 0x80000
(320KB ~ 512KB)

Program Data - 1 (64 KB)
0x40000 ~ 0x50000
(256KB ~ 320KB)

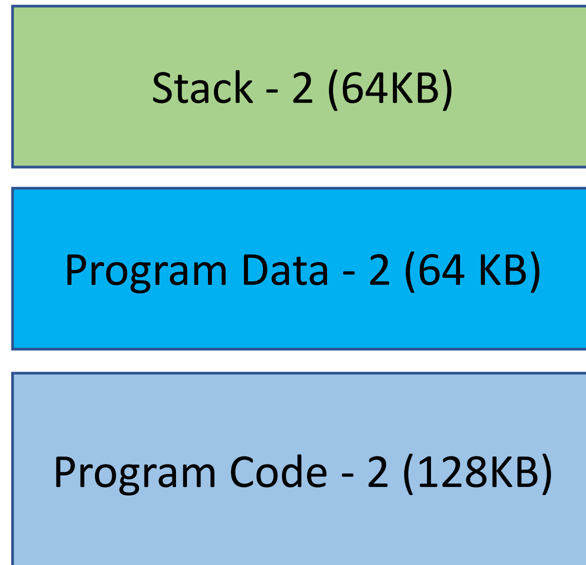
Free (64 KB)
0x30000 ~ 0x40000
(192KB ~ 256KB)

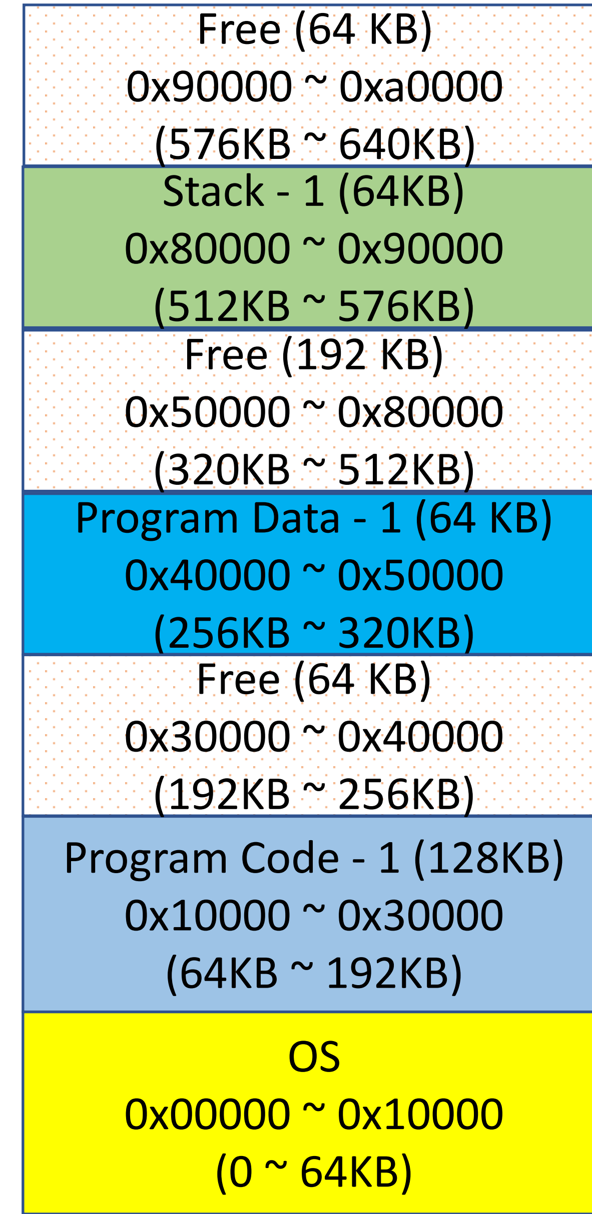
Program Code - 1 (128KB)
0x10000 ~ 0x30000
(64KB ~ 192KB)

OS
0x00000 ~ 0x10000
(0 ~ 64KB)

Multi-programming Environment

- Run two programs

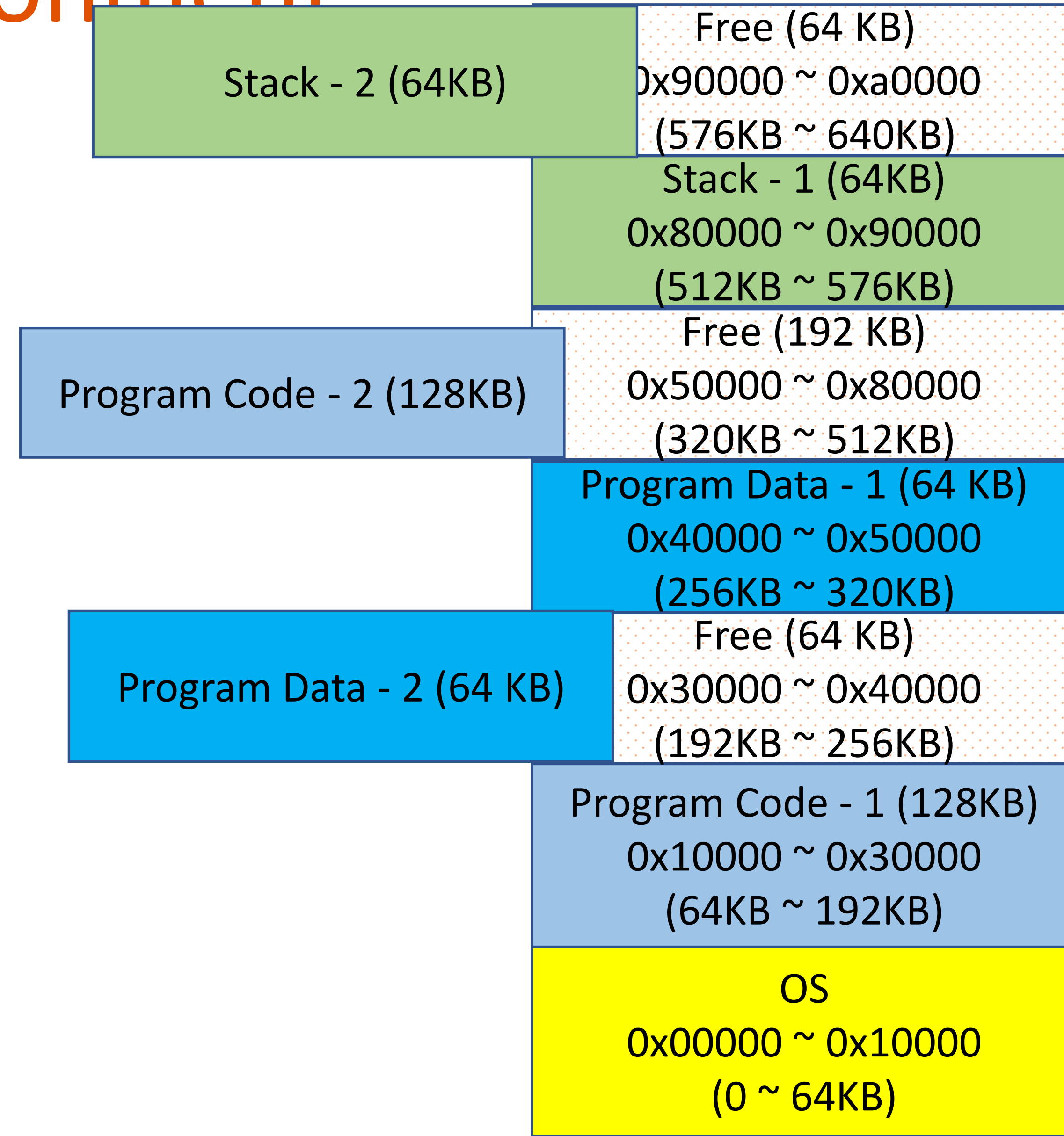




Multi-programming Environment

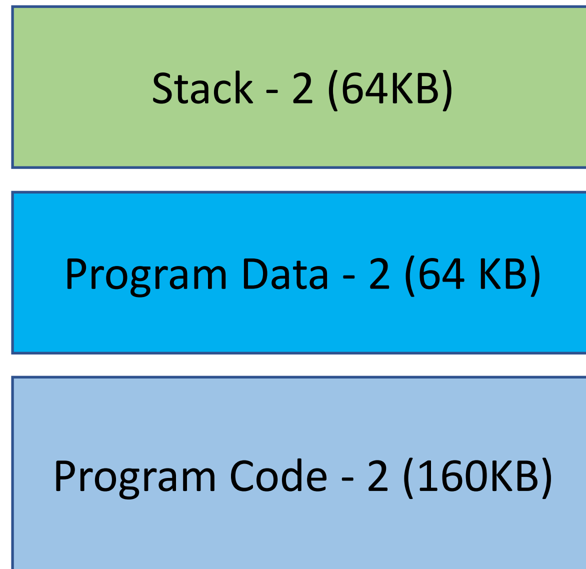
- Run two programs
- System's memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can't determine… system does..

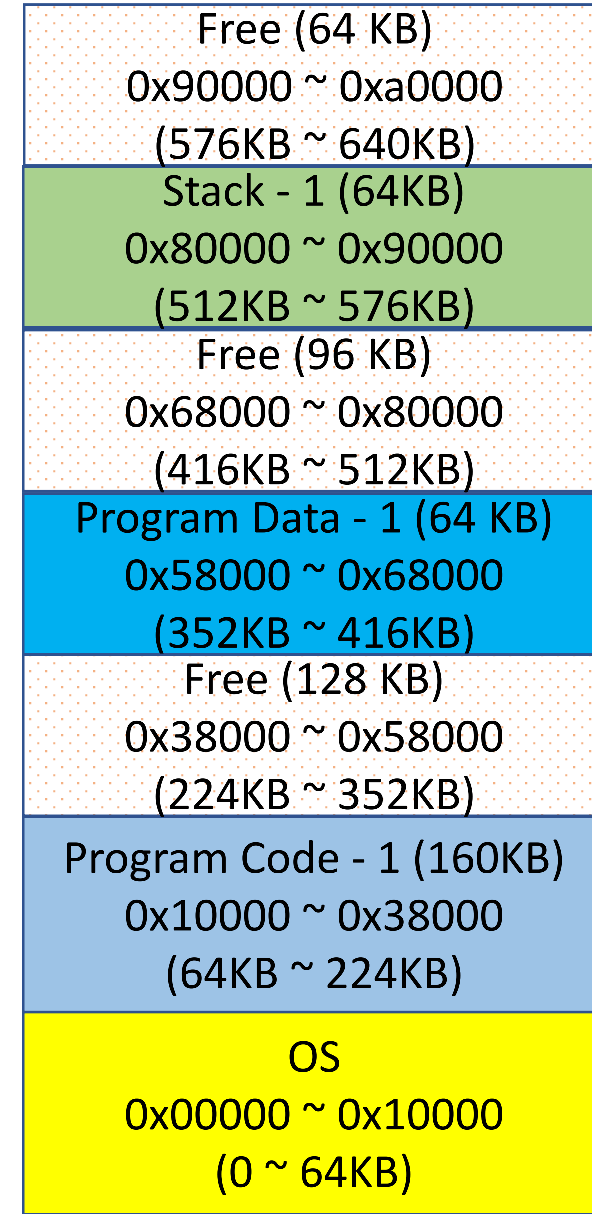
No Transparency…



Multi-programming Environment

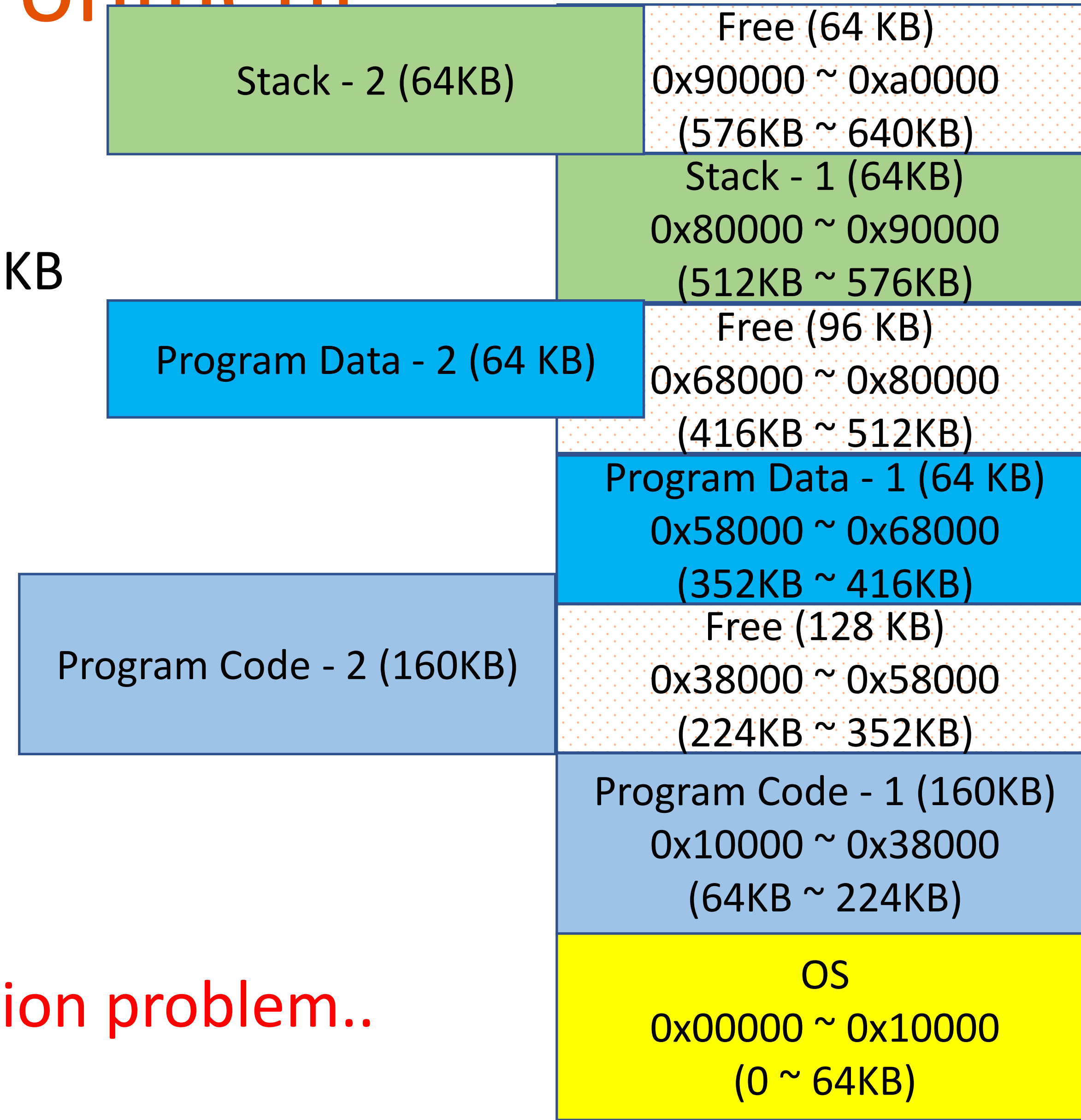
- Run two programs





Multi-programming Environment

- Run two programs
  - Program size: 64KB + 64KB + 160K = 288KB
- Free mem
  - 64 + 96 + 128 = 288KB
- Cannot run Program – 2
  - Can’t fit…

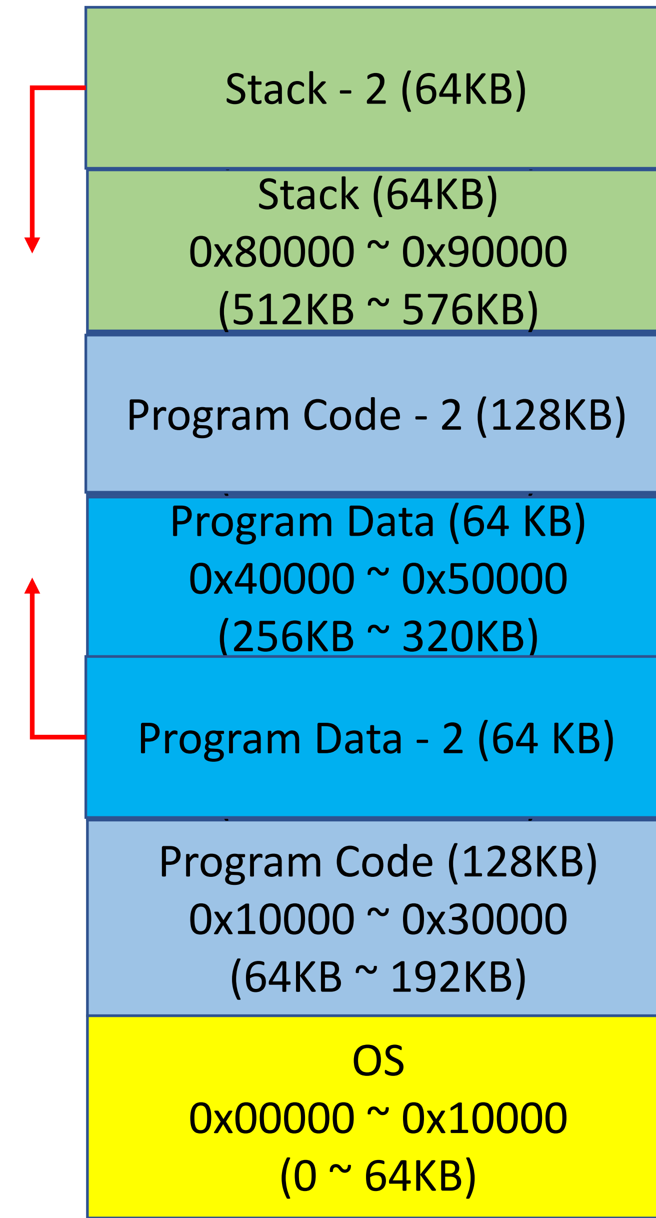


Not efficient.. Suffers memory fragmentation problem..

Multi-programming Environment

- Run two programs
- What if Program-2's stack underflows?
- What if Program-2's data overflows?
- Without virtual memory
  - Programs can affect to the other's execution

No isolation. Security problem.



Virtual Memory

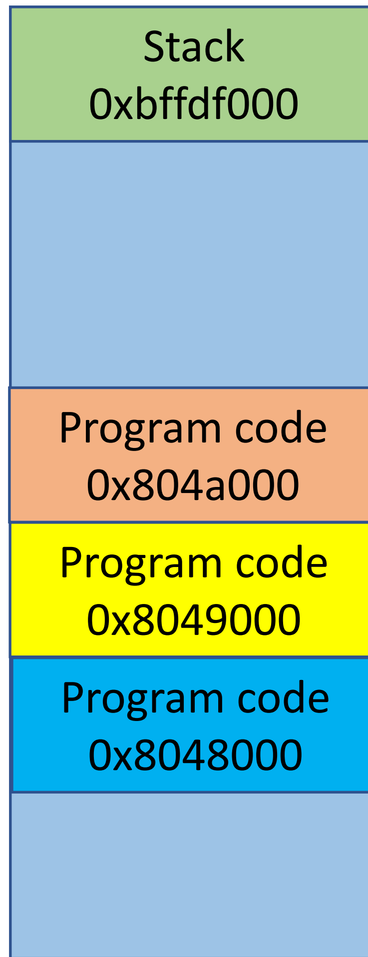
- Three goals
  - **Transparency**: does not need to know system's internal state
    - Program A is loaded at 0x8048000. Can Program B be loaded at 0x8048000?
  - **Efficiency**: do not waste memory; manage memory fragmentation
    - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless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

Paging

- A method of implementing virtual memory
- Split memory into multiple 4,096 byte blocks (12-bit)
  - Last 3 digits of page address are ZERO (in hexadecimal)
  - E.g., 0x0, 0x1000, 0x2000, …, 0x8048000, 0x804a000, …, 0x7fffe000, etc.
- Having an indirect map between virtual page and physical page
  - Set an arbitrary virtual address for a page, e.g., 0x81815000
  - Set a physical address to that page as a map, e.g., 0x32000
  - 0x81815000 ~ 0x81815fff will be translated into
  - 0x32000 ~ 0x32fff

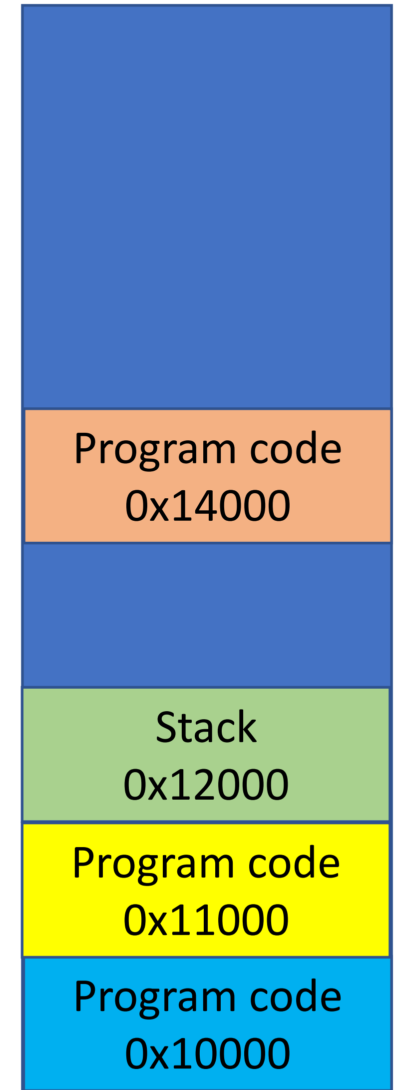
Virtual Memory - Paging

- Having an indirect table that maps virt-addr to phys-addr



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

Physical Memory



Paging: Virtual Memory

- Having an indirect table that maps virt-addr to phys-addr

Stack-2
0xbffdf000

Program code-2
0x804a000

Program code-2
0x8049000

Program code-2
0x8048000

Stack
0xbffdf000

Program code
0x804a000

Program code
0x8049000

Program code
0x8048000

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

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

Physical Memory

Stack-2
0x17000

Program code-2
0x16000

Program code-2
0x15000

Program code
0x14000

Program code-2
0x13000

Stack
0x12000

Program code
0x11000

Program code
0x10000

Transparency: does not need to know system's internal state
Program A is loaded at 0x8048000.
Can Program B be loaded at 0x8048000?

- Having an indirect table that maps virt-addr to phys-addr

Stack-2 0xbffdf000
Program code-2 0x804a000
Program code-2 0x8049000
Program code-2 0x8048000

Stack 0xbffdf000
Program code 0x804a000
Program code 0x8049000
Program code 0x8048000

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

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

Physical Memory

Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

Efficiency: do not waste memory
Can Program B (288KB) be loaded if
only 288 KB of memory is free, regardless of its allocation?

- Having an indirect table that maps virt-addr to phys-addr

Stack-2 0xbffdf000	Stack 0xbffdf000
Program code-2 0x804a000	Program code 0x804a000
Program code-2 0x8049000	Program code 0x8049000
Program code-2 0x8048000	Program code 0x8048000

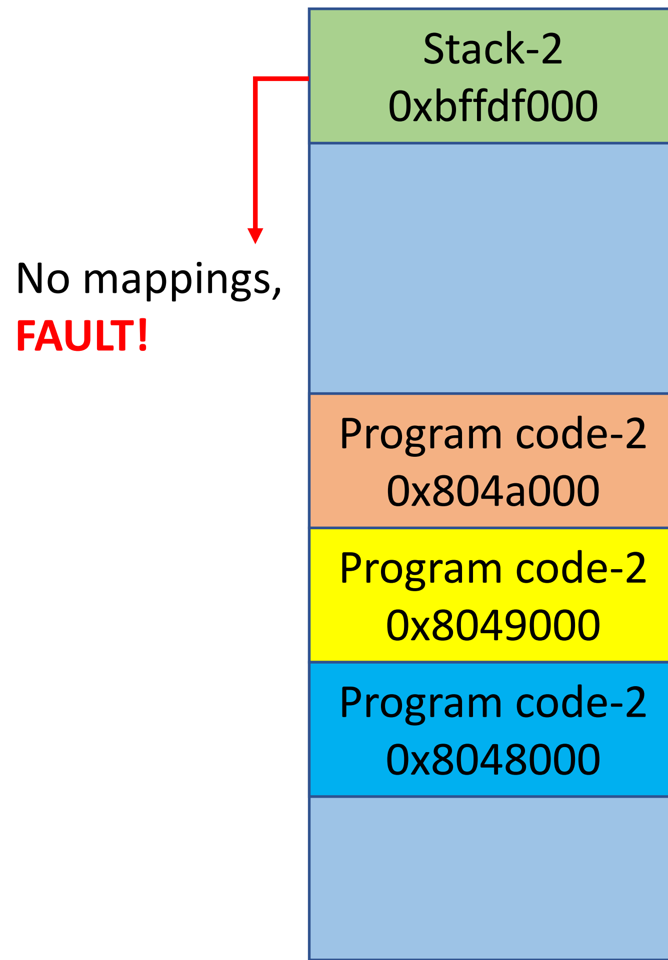
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

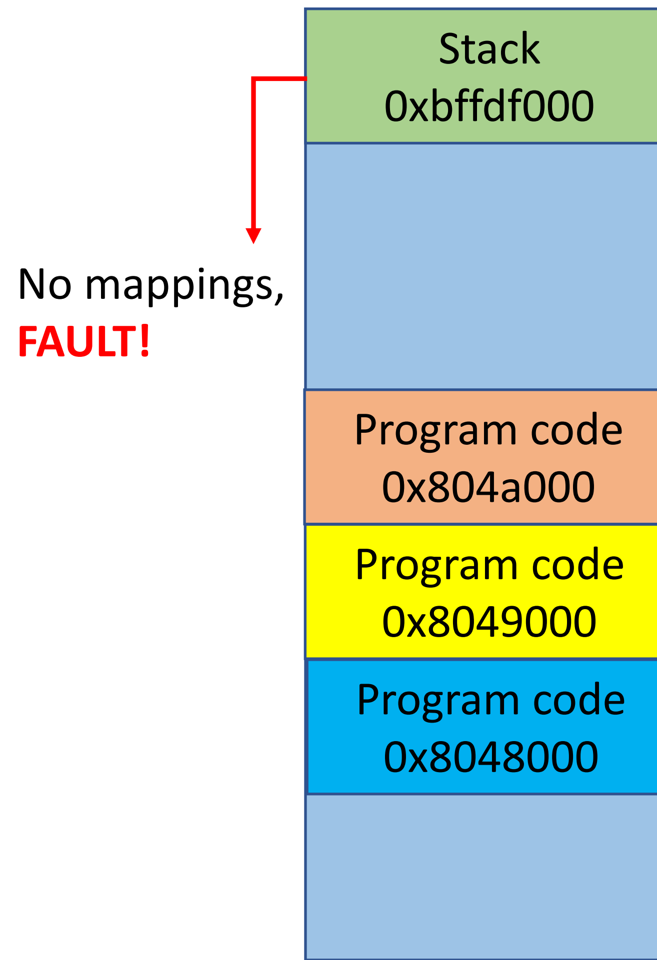
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

Physical Memory

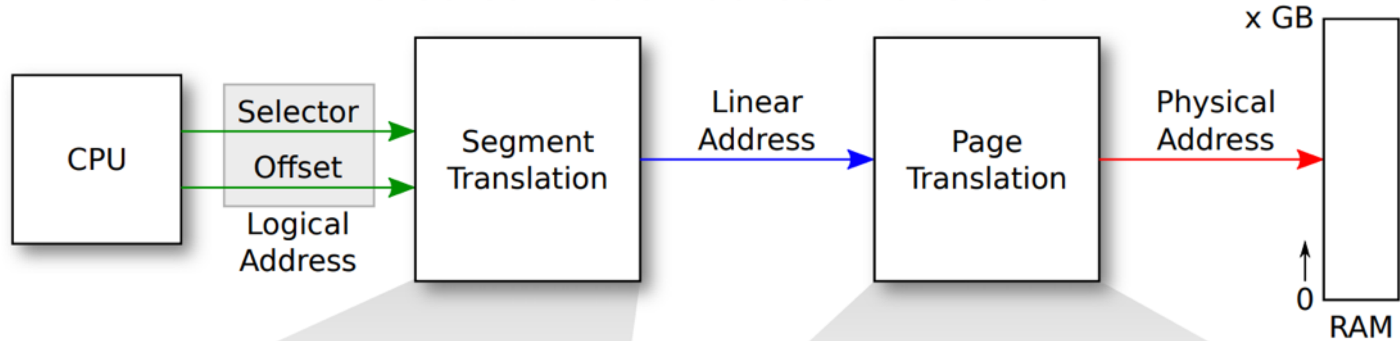
Physical Memory
Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

Protection: isolate program's execution environment
Can we prevent an overflow from Program A from overwriting Program B's data?





Protected-Mode Address Translation



Page Directory / Table

- CPU translates virtual address to physical address by
  - Translating the page number (not FULL ADDRESS)

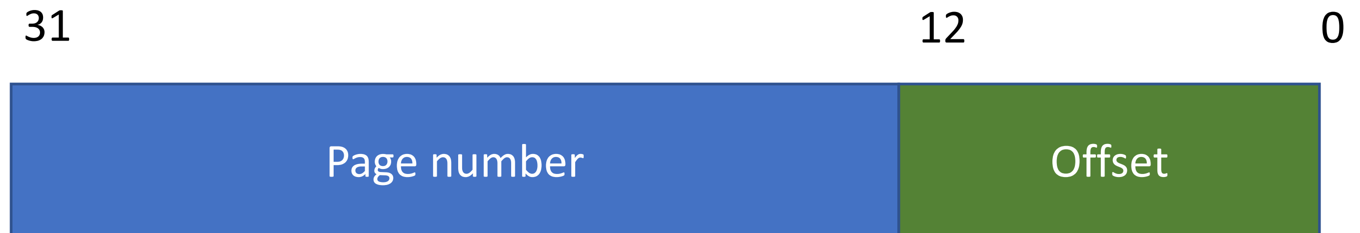
- Page size: 4 KB (12bits)
  - Last 3 digits are 0

- Page number: 20 bits

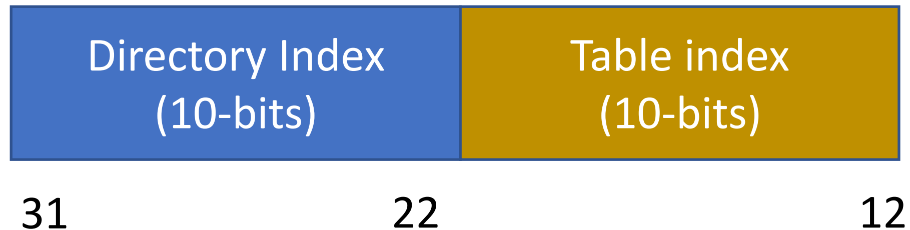
- What is the page number and offset of
  - 0x**08048000** → **0x01234000**
  - 0x**b7ff3100** → **0x34567100**

Page Directory / Table

- In x86 (32-bit), CPU uses 2-level page table
  - Each level stores 1024 entries
  - Why 1024? 1 page = 4096 bytes; each entry is 4 bytes; 4096 / 4 = 1024

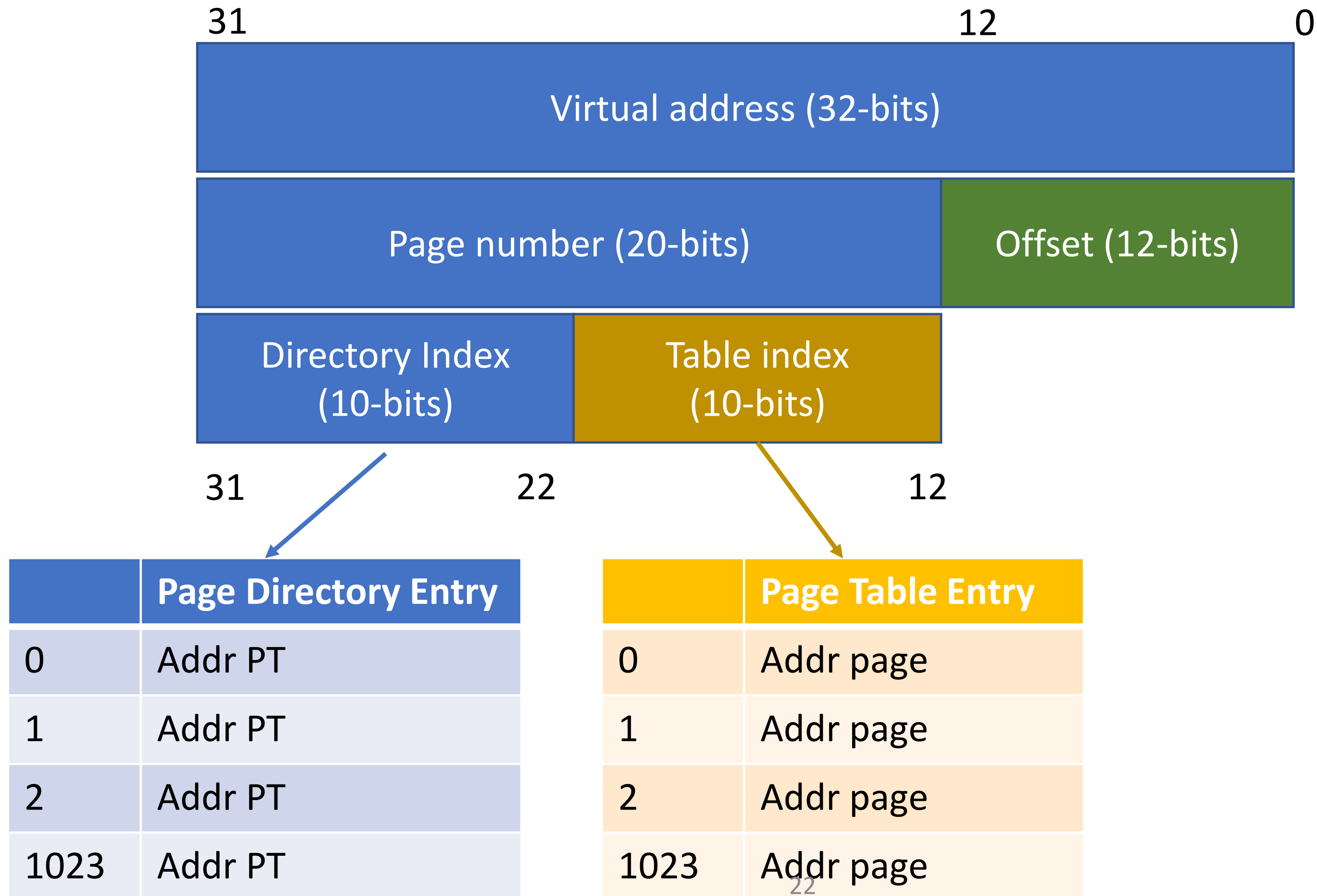


- 10-bit directory index



- 10-bit page table index
- 12-bit offset

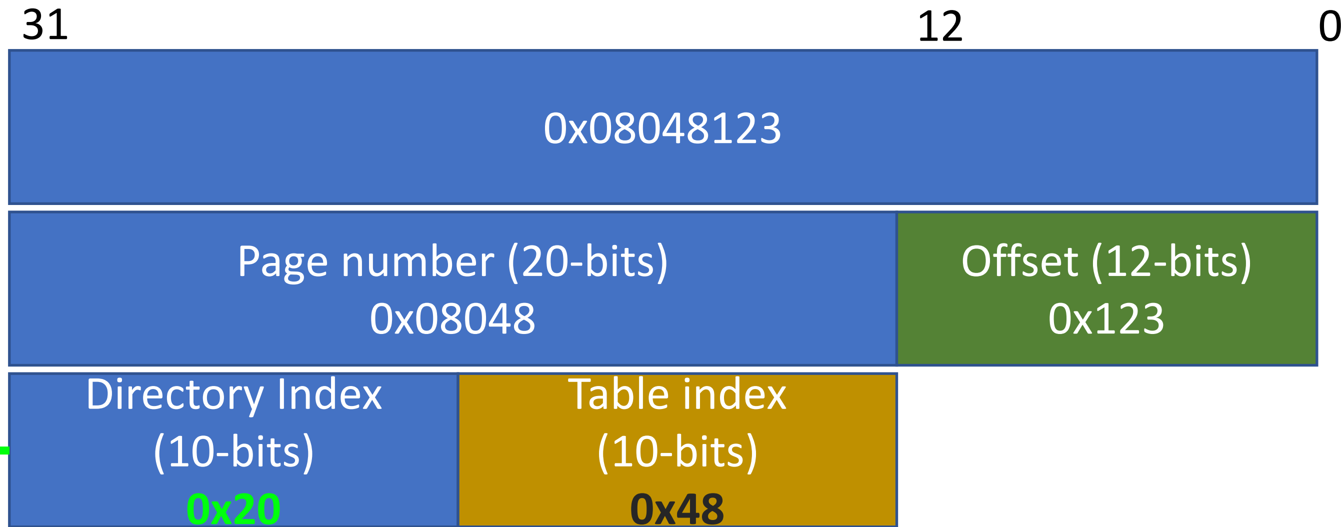
Address Translation



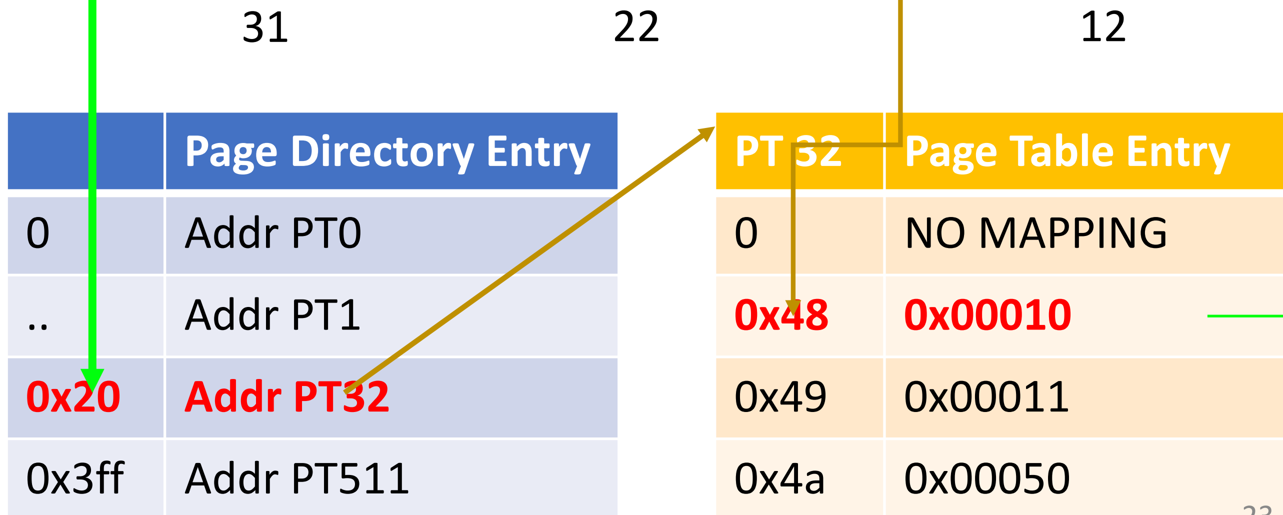
	Page Directory Entry
0	Addr PT
1	Addr PT
2	Addr PT
1023	Addr PT

	Page Table Entry
0	Addr page
1	Addr page
2	Addr page
1023	Addr page

Address Translation Example




```
>>> bin(0x8048)
'0b1000000001001000'
>>> bin(0x8048 >> 10)
'0b100000'
>>> hex(0x8048 >> 10)
'0x20'
>>>
```



Physical access
0x00010123

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

Address Translation Examples

- Virtual address 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

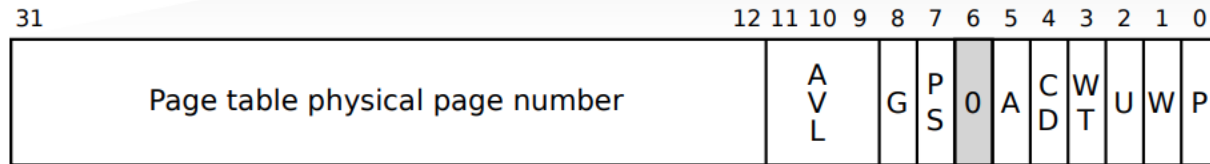
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
0x44332000	0x33885000

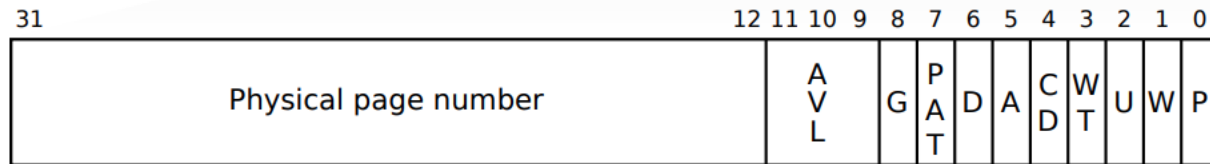
Virtual	Physical
0x8048	0x10
0x8049	0x11
0x804a	0x14
0xbffdf	0x12
0x44332	0x33885

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 remaining as the same
  - We do not translate the lower 12 bits!

PDE, PTE





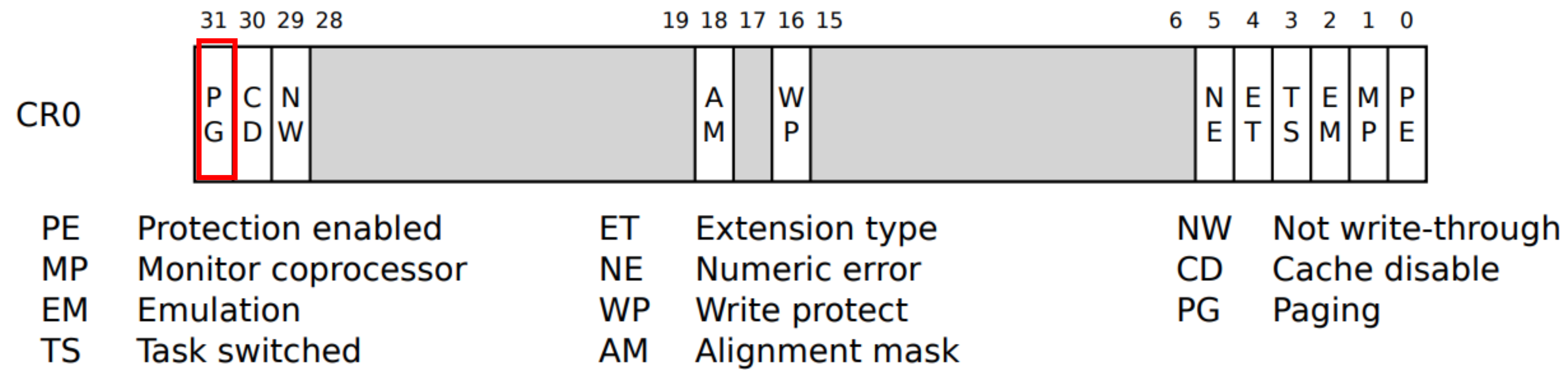
P	Present
W	Writable
U	User **0: kernel, 1: user**
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

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

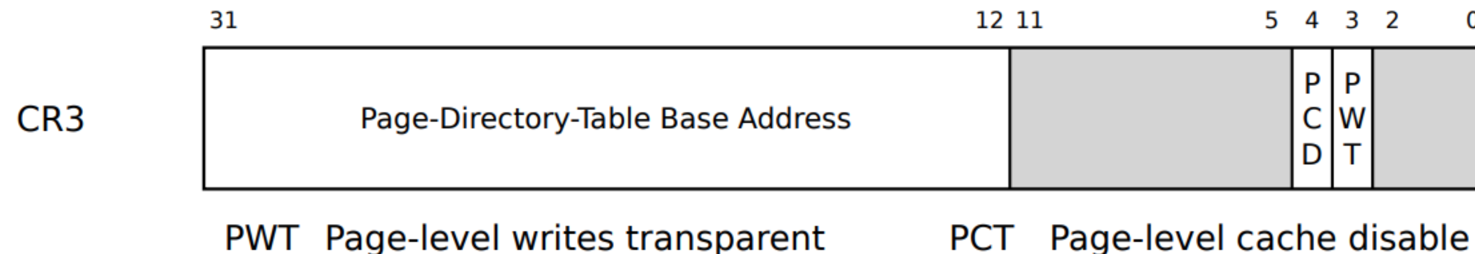
PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Paging in x86

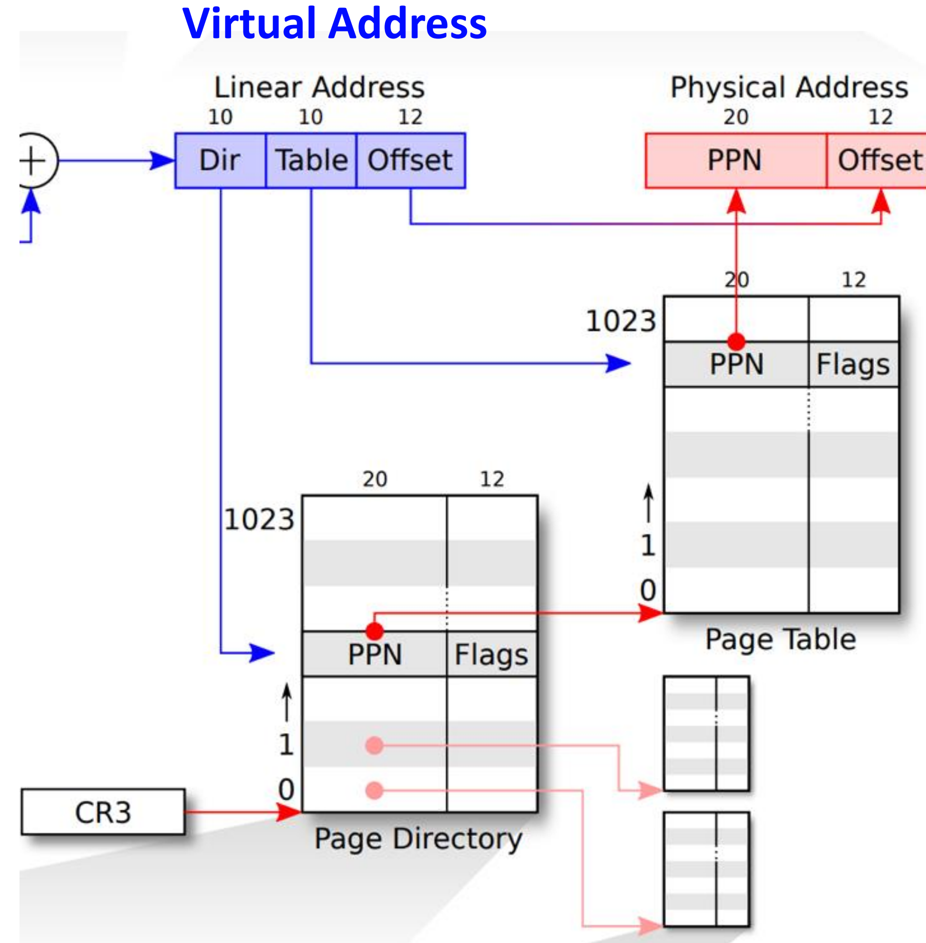
- Can be enabled via CR0



- Page directory 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 address access regarded as virtual address

- What if we would like to access the physical address 0x100010?
  - Our kernel is at physical address 0x100000 ~ 0x110000
  - We need to map virtual 0x100000 to physical 0x100000
    - JOS maps only few pages of kernel for this purpose

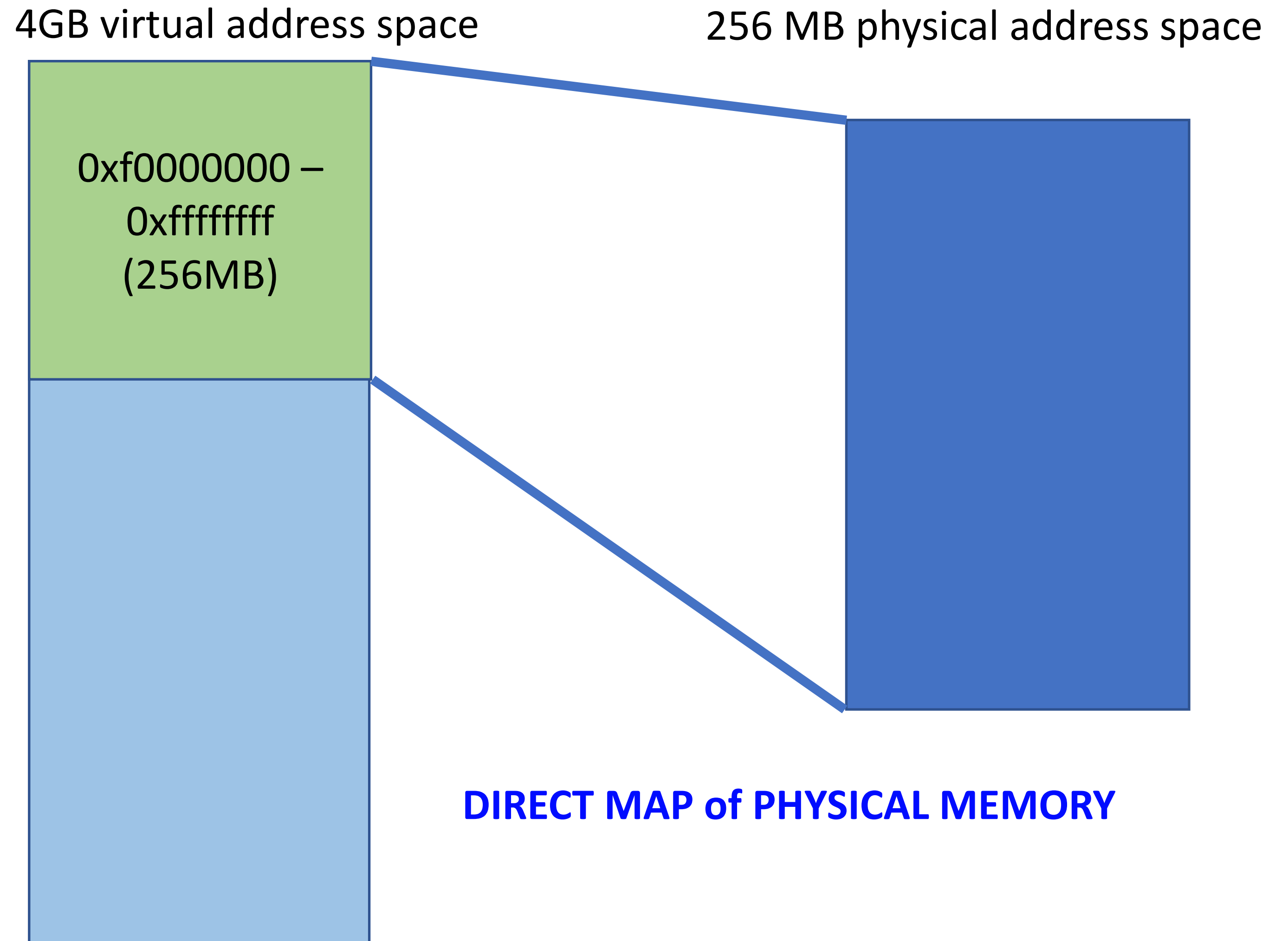
- What about other addresses? E.g., 0x1333358 as physical?

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

- 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 * pointer = 16GB… 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

- How much memory do you need to allocate 1 byte?
  - 1GB page – 1GB
  - 4MB page – 4MB
  - 4KB page – 4KB
  - 1B page – 1B

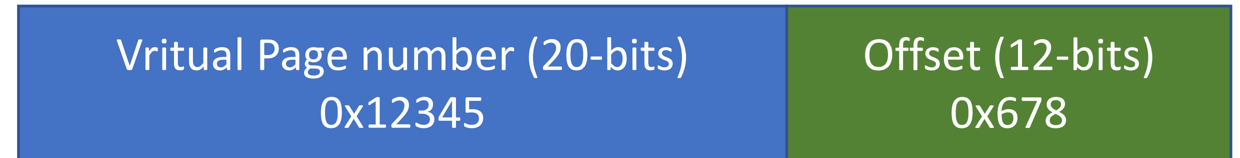
Design consideration:
Memory fragmentation matters!

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

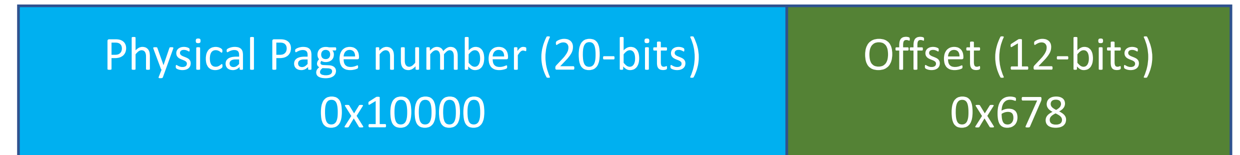
Page Size / Page Table Size

- 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

Page Table Lookup

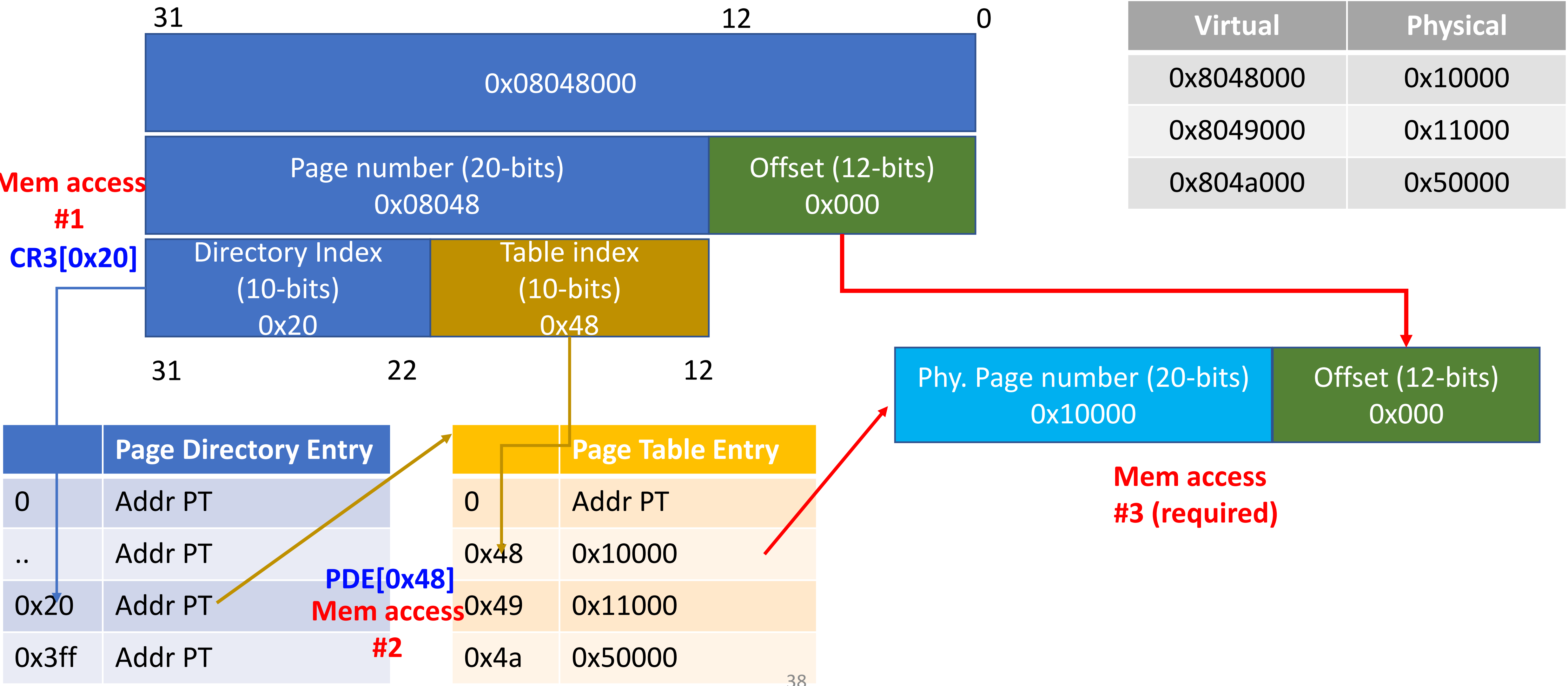


- Access example
  - `movl 0x12345678, %eax`
  - (load `4-byte data` from the address `0x12345678` to `%eax`)



- Memory access sequence
  - Virtual page number: top 20bits, **PGNUM** `= (0x12345678 >> 12) =` **0x12345**
  - Page Directory Index: top 10 bits, `PDX = (` **PGNUM** `>> 10 )& 0x3ff = 0x48`
  - Page Table Index: next 10 bits, `PTX =` **PGNUM** `& 0x3ff = 0x345`
  - Page Table Entry: `PTE = CR3[PDX][PTX]` – 2 memory dereferences
  - Physical address = **Physical Page Number** + **OFFSET**
    = `(PTE & 0xfffff000) + (0x12345678 & 0xfff)`
  - eax = PHY_ADDR[0] – 1 memory dereferences (required)

Recap – Page Table & Addr Translation



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

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

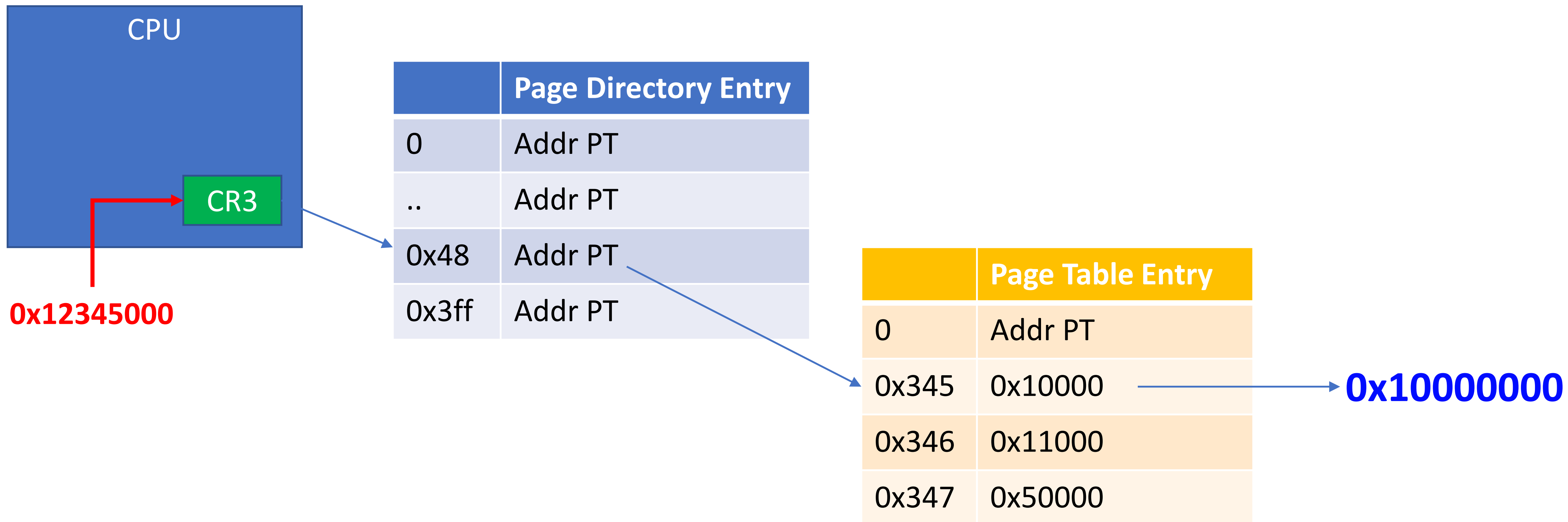
0x12346678 -> 0x5678

0x12347678 -> 0xff678

0x12348678 -> 0xfff678

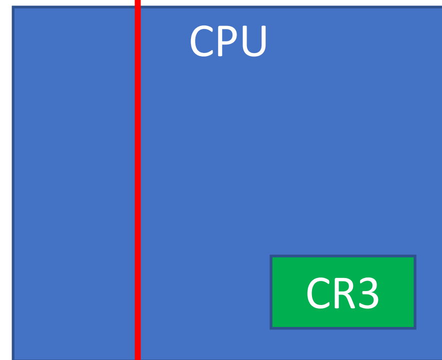
- Benefits
  - Do not have to walk down page tables for cached entries

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

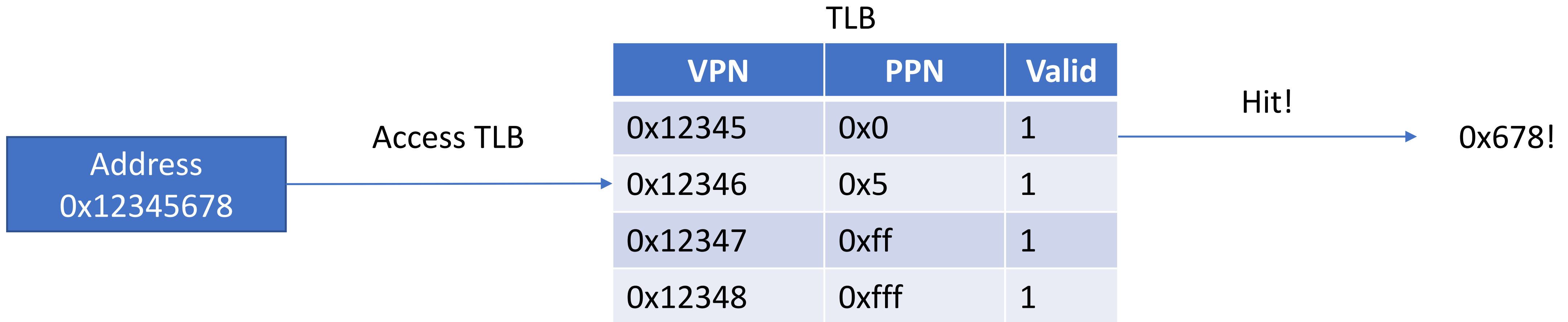
	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

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

0x10000000

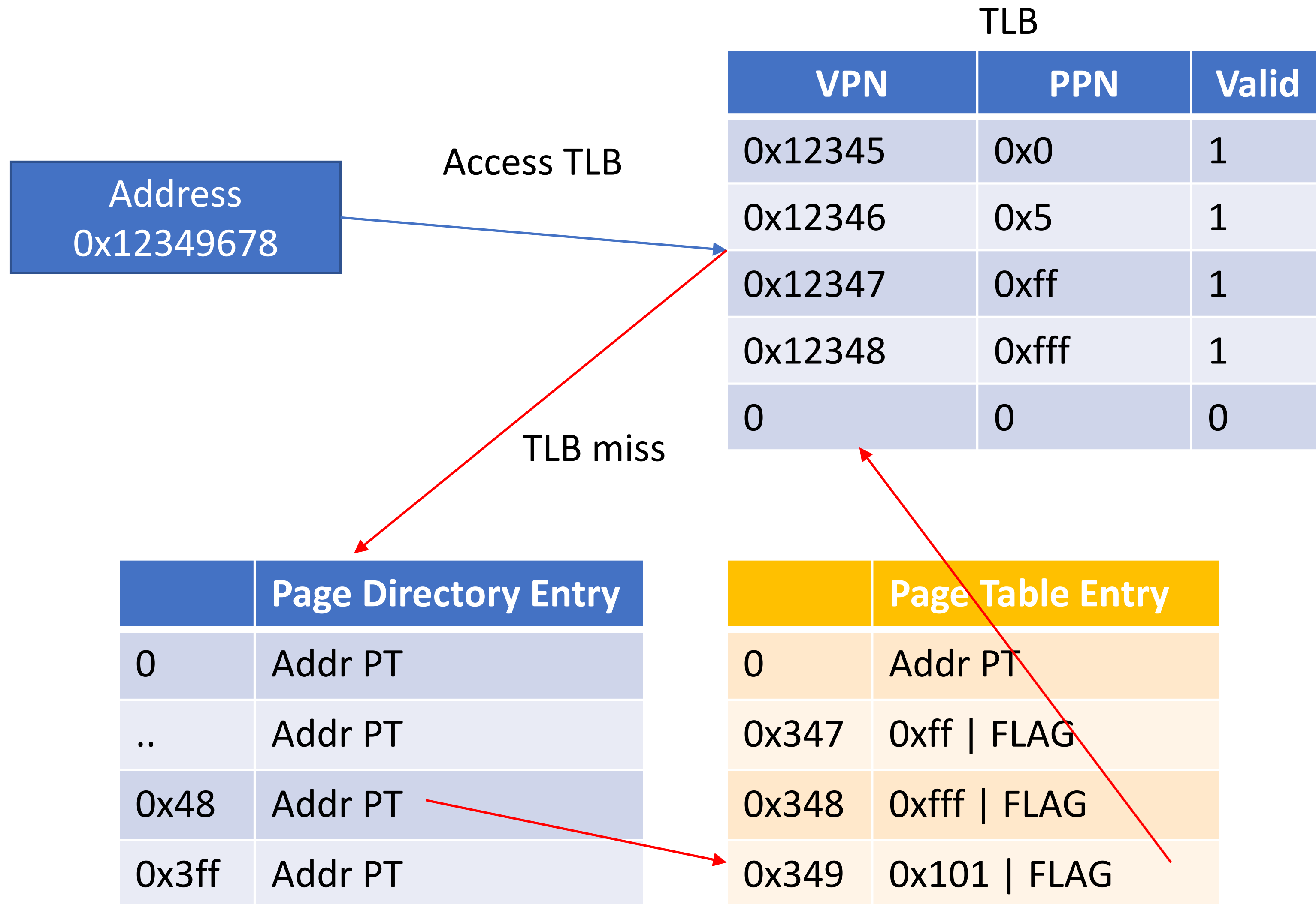
No page table access..

Address Translation with TLB (hit)



VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

Address Translation with TLB (miss)



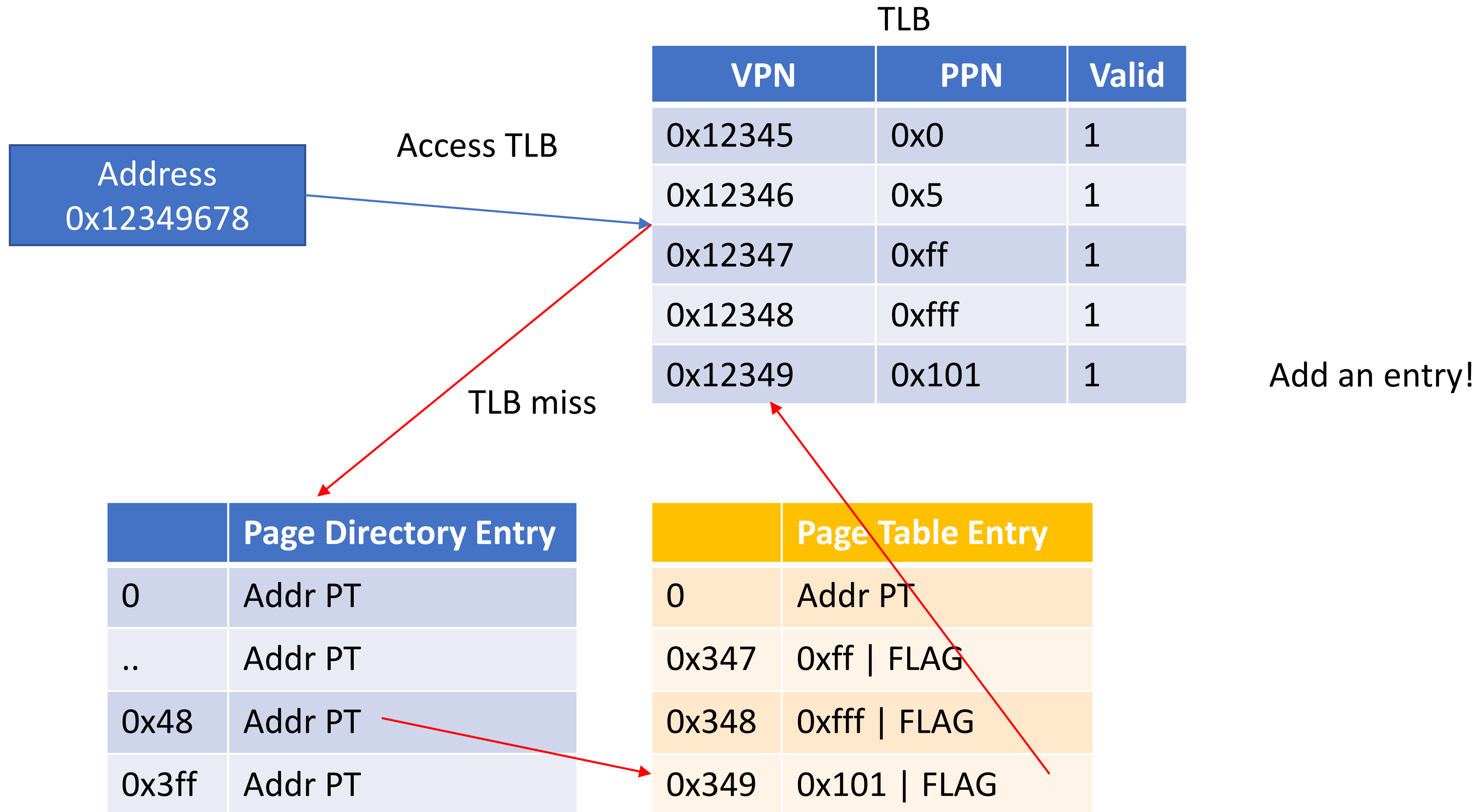
TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0	0	0

	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

Address Translation with TLB (miss)

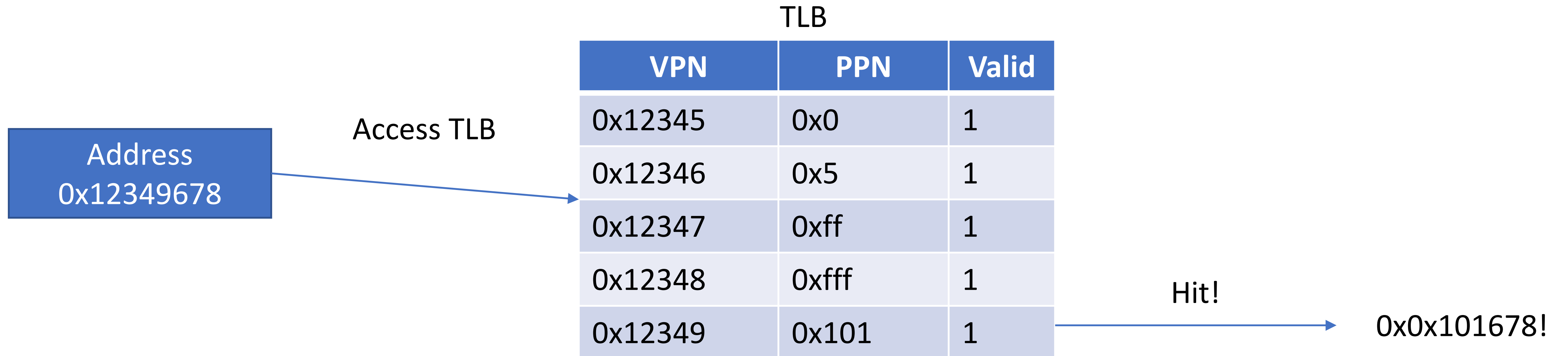


VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

	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

Address Translation with TLB (hit)



VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

	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…

TLB Replacement Policy

- Smart: LRU (Least Recently Used)
  - Mark when the block was accessed (update timestamp upon access)
  - When an eviction is required, evict the one with the oldest timestamp
- Random
  - Do not do anything when accessed
  - When an eviction is required, evict an entry randomly…
  - Sometimes, this works better than LRU..
- We will learn more about such scheduling later
  - When we learn about process scheduling…

Synchronizing TLB with Page Table

- CPU uses the TLB for caching Page Table Entries
- What will happen if content in TLB mismatches to the PTE in the page table?
  - Access wrong physical memory…
  - Does not honor the correct privilege in PTE (if we updated PTE after caching)
  - Running a new process with new CR3
    - Use old process's mapping, wrong access

TLB and Page Table Update

Address
0x12349678

TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

	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

TLB and Page Table Update

Address
0x12349678

TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

	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	**0x102** | FLAG

Update

TLB and Page Table Update

Address
0x12349678

TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	**0**

We need to invalidate this entry

	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	**0x102** | FLAG

Update

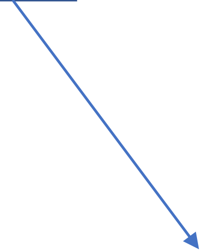
TLB and Process Context Switch

TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

Address
0x12349678

CR3



	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

TLB and Process Context Switch

TLB

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0x12349	0x101	1

Address
0x12349678

CR3

	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

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

	Page Table Entry
0	Addr PT
0x347	0x20 | FLAG
0x348	0x30 | FLAG
0x349	0x50 | FLAG

TLB and Process Context Switch

TLB

VPN	PPN	Valid
0x12345	0x0	0
0x12346	0x5	0
0x12347	0xff	0
0x12348	0xfff	0
0x12349	0x101	0

We need to invalidate all previous entries..

Address
0x12349678

CR3

	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

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

	Page Table Entry
0	Addr PT
0x347	0x20 | FLAG
0x348	0x30 | FLAG
0x349	0x50 | FLAG

Updating Page Table

- When updating a Page Table Entry
  - We must invalidate TLB for that entry
  - invlpg

INVLPG — Invalidate TLB Entries

		Op/En	64-Bit Mode	Compat/Leg Mode	Description
		M	Valid	Valid	Invalidate TLB entries for page containing *m*.

Updating Page Table

- In JOS (kern/pmap.c & inc/x86.h)

```
//
// Invalidate a TLB entry, but only if the page tables being
// edited are the ones currently in use by the processor.
//
void
tlb_invalidate(pde_t *pgdir, void *va)
{
	// Flush the entry only if we're modifying the current address space.
	// For now, there is only one address space, so always invalidate.
	invlpg(va);
}
```

```
static inline void
invlpg(void *addr)
{
	asm volatile("invlpg (%0)" : : "r" (addr) : "memory");
}
```

Summary

- Address translation
  - Segmentation: base+offset -> linear address
  - Paging: virtual -> physical
    - Virtual Page Number -> Physical Page Number (top 20 bits)
    - Page directory index (10 bits) + Page table index (10 bits)

- TLB
  - Benefits?
  - Scheduling?
  - Invalidating?