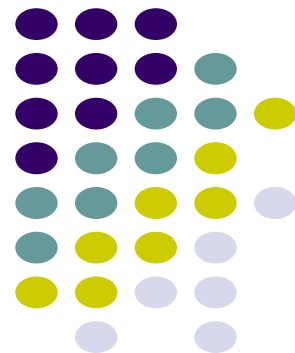


Paging and Virtual Memory Translation

ECE 469, Jan 18

Aravind Machiry



Virtual Memory



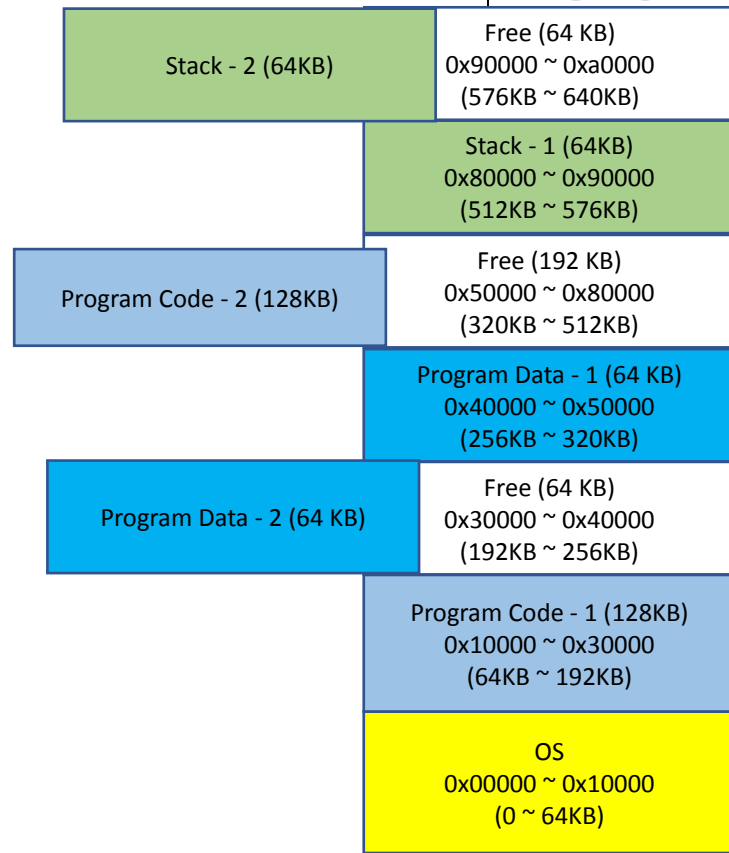
- Three goals
 - Transparency: does not need to know system's internal state
 - Program A is loaded at `0x8048000`. Can Program B be loaded at `0x8048000`?
 - Efficiency: do not waste memory; manage memory fragmentation
 - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its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

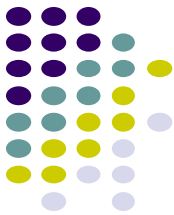
Multi-programming Environment



- 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..

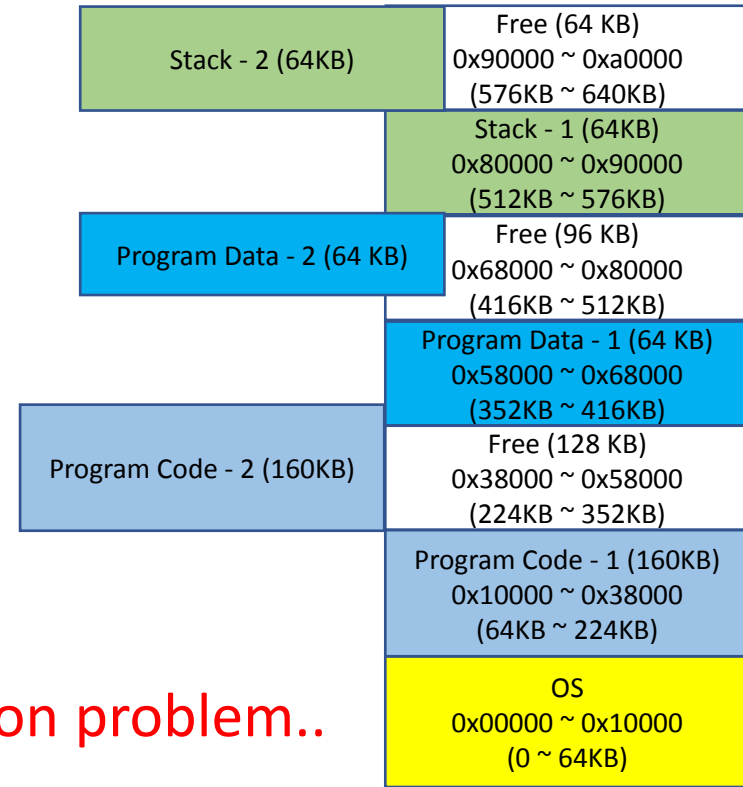
No
Transparency...





Multi-programming Environment

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



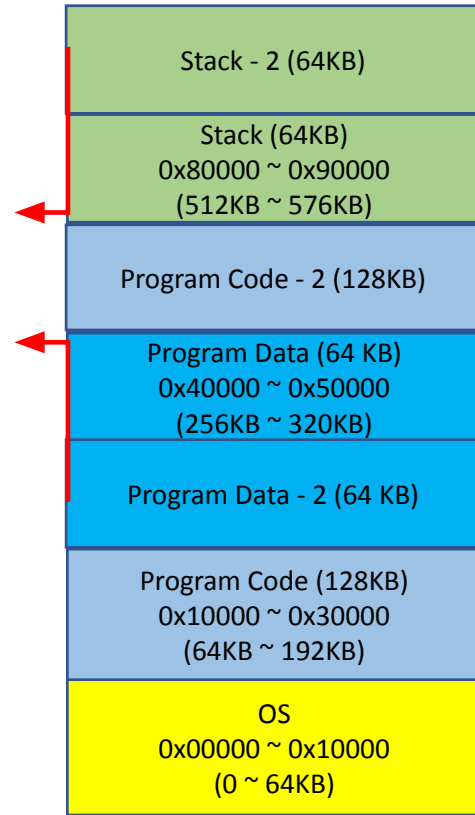
Not efficient.. Suffers memory fragmentation problem..



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.



Paging!

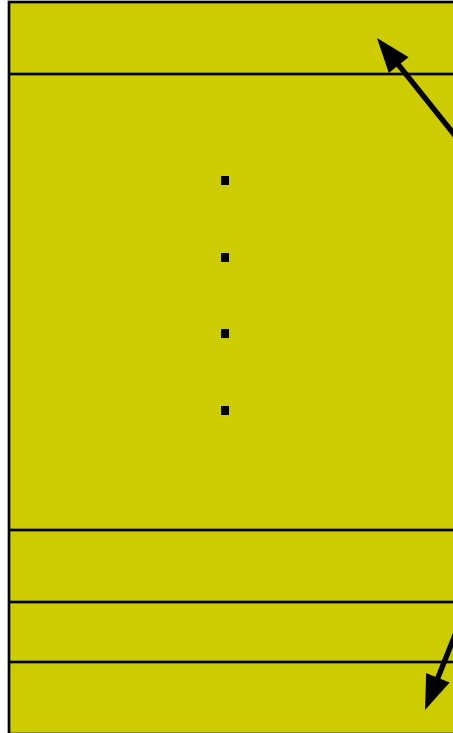


- Idea: Make all chunks of memory the same size, called **pages**
 - Both virtual and physical memory divided into same size chunks.
- For each process, a **page table** defines the base address of each of that process' pages along with existence and read/write bits

Paging!

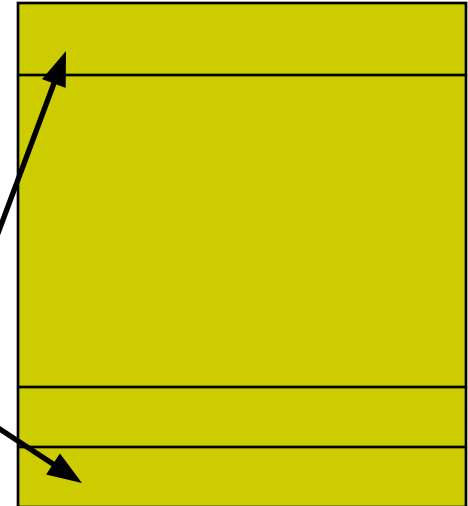


Virtual address



Virtual pages

Physical memory



physical pages

Page Table: matters of size



- Table of entries:
 - Virtual memory page to physical memory page.
 - Number of entries proportional to the size of page.
 - Each entry is of the size of a pointer = 4 bytes
- Page table size for 32-bit address space:
 - With page size 4KB = $(2^{32})/(2^{12}) = (2^{20})$ entries = $(2^{20}) * 4$ bytes = 4 MB.
 - With page size 8KB = $(2^{32})/(2^{13}) = (2^{19})$ entries = $(2^{19}) * 4$ bytes = 2MB.
 - With page size 8MB = $(2^{32})/(2^{23}) = (2^9)$ entries = $(2^9) * 4$ bytes = 2KB.

Page size / fragmentation



- 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

Design consideration:
Memory fragmentation matters!

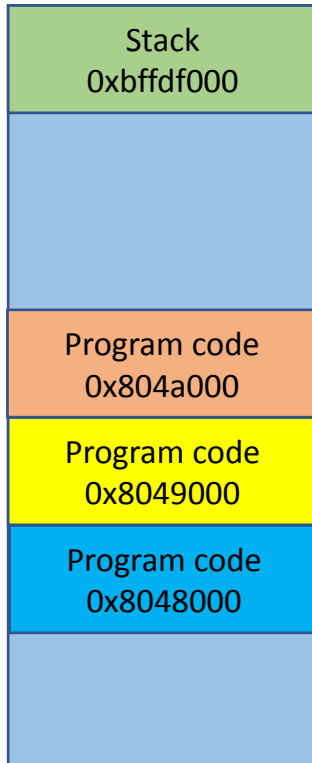
Paging



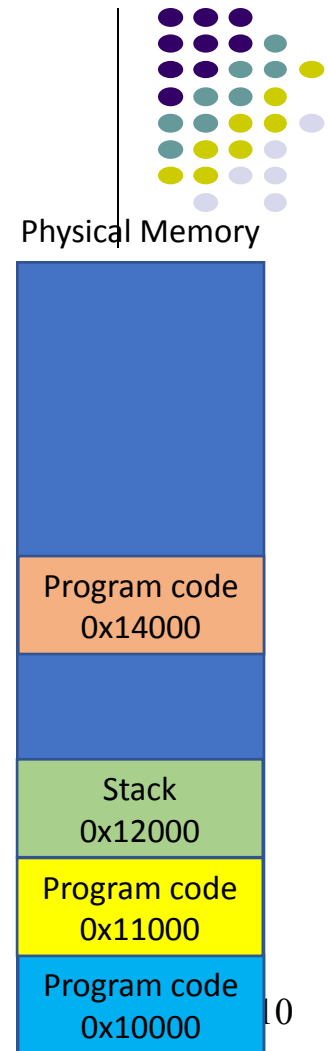
- A method of implementing virtual memory - common page size: 4KB
- Split memory into multiple 4,096 byte blocks (12-bit offset)
 - Page start address: Last 3 digits of page address are ZEROs (in hexadecimal)
 - E.g., 0x0, 0x1**000**, 0x2**000**, ..., 0x8048**000**, 0x804a**000**, ..., 0x7fffe**000**, 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

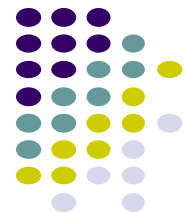


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
...	...



Virtual Memory - Paging

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



Physical Memory

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
...	...

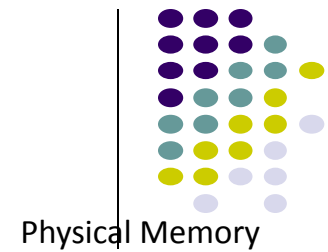
Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

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

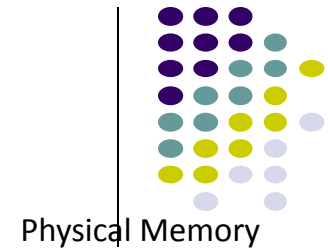
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
...	...

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

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

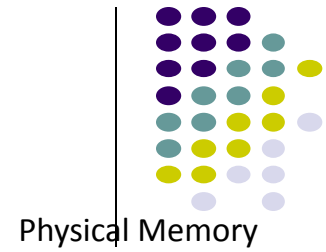
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
...	...

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

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**?

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
...	...

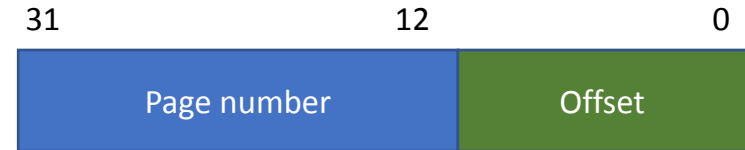
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
...	...

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

Page Table



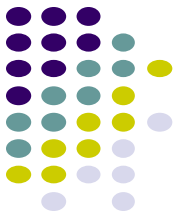
- We access page table by virtual address
- Page size: 4 KB (12bits)
- Page number: 20 bits
- What is the page number and offset of
 - **0x8048000**
 - **0xb7ff3100**



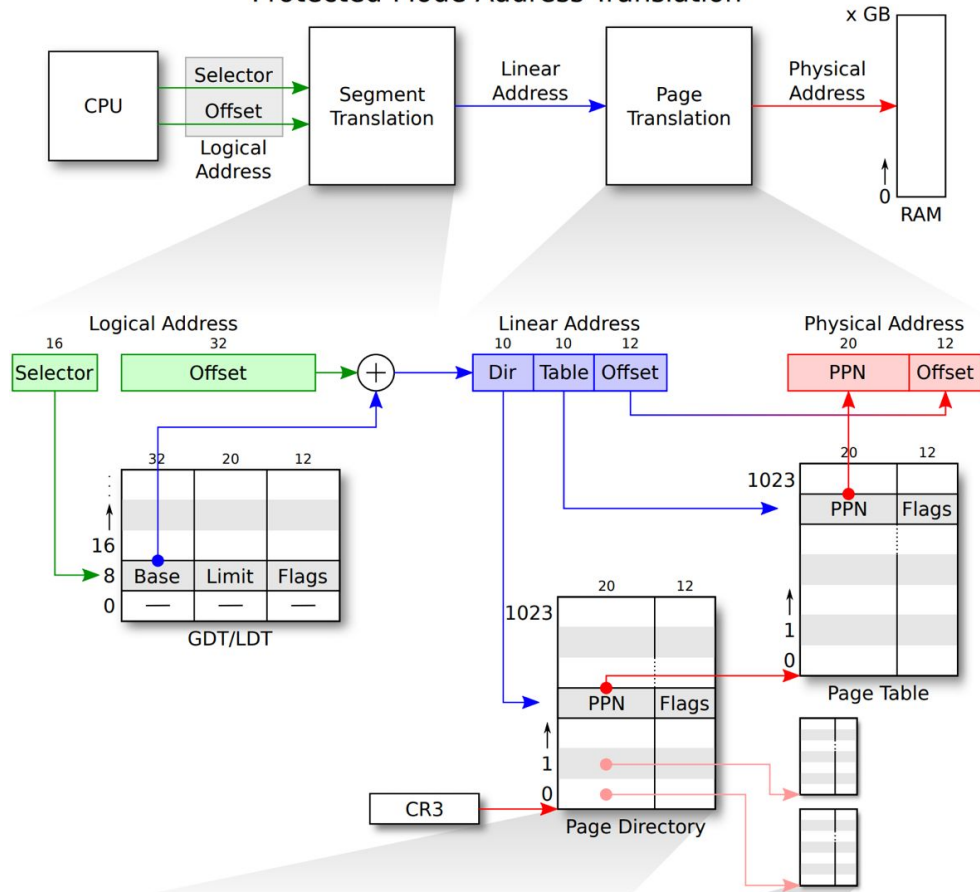
Balancing Page Size v/s Page Table Size



- Page size: 4 KB (12 bits)
 - Page Table size: $(2^{20}) * 4$ bytes = **4MB (of contiguous physical memory)**.
- If a program needs 1KB of memory, it still needs 4MB for its page table??!!!
- How do we balance this?
 - Add another level of indirection: Page directories and Page Tables.



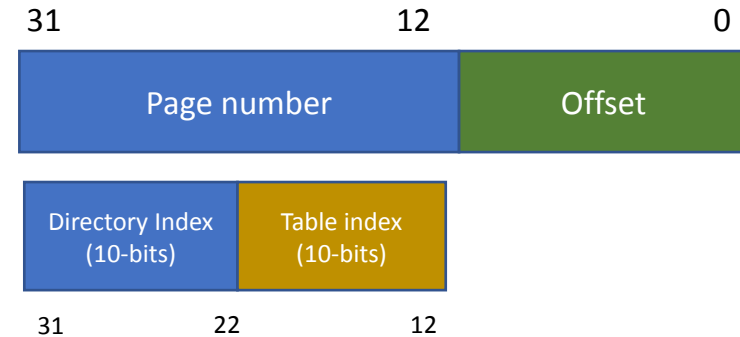
Protected-Mode Address Translation



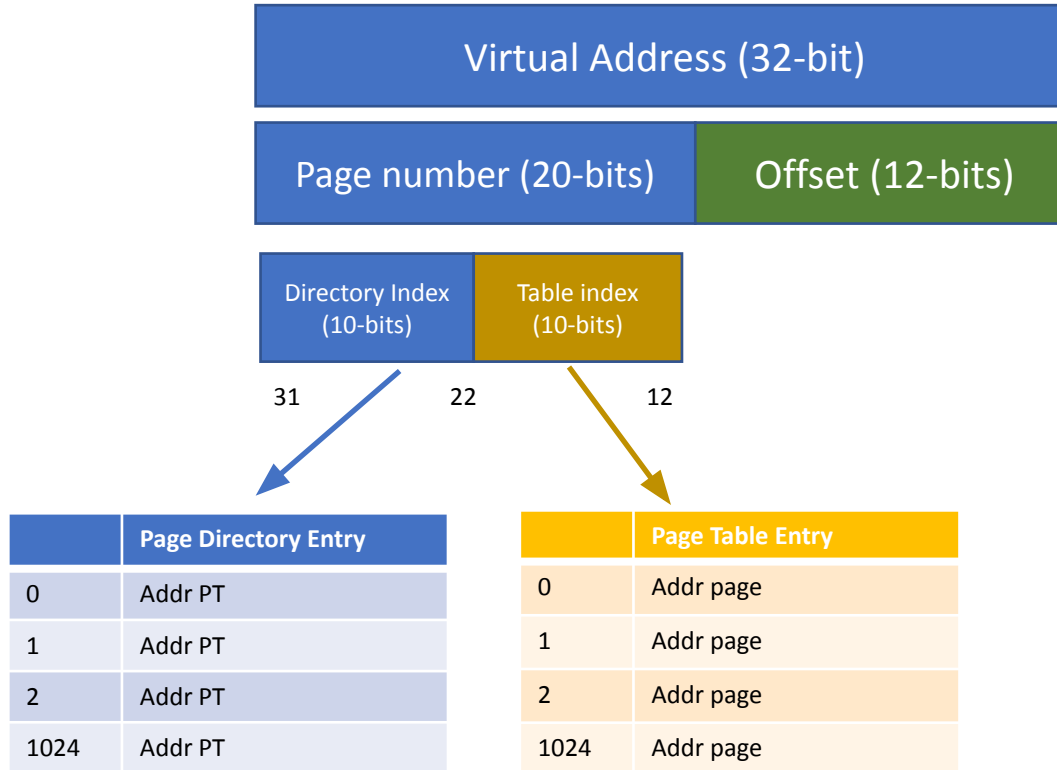
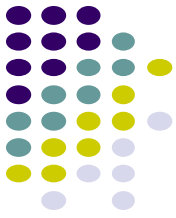
Page Directory / Table



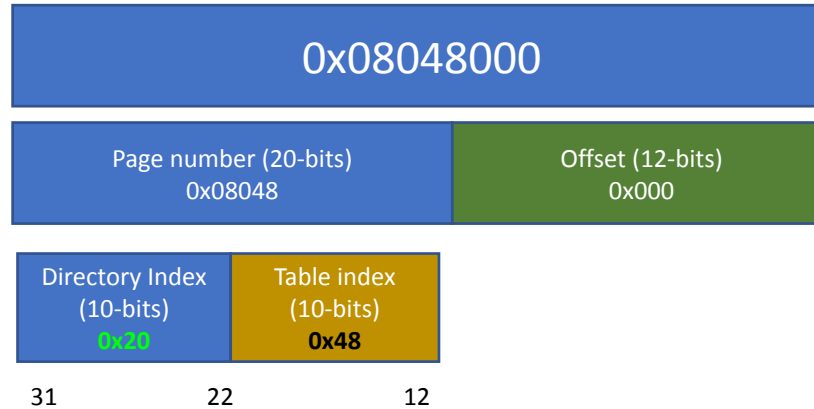
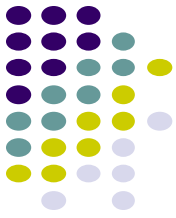
- In x86 (32-bit), CPU uses 2-level page table
- 10-bit directory index
- 10-bit page table index
- 12-bit offset



Address Translation



Address Translation Example

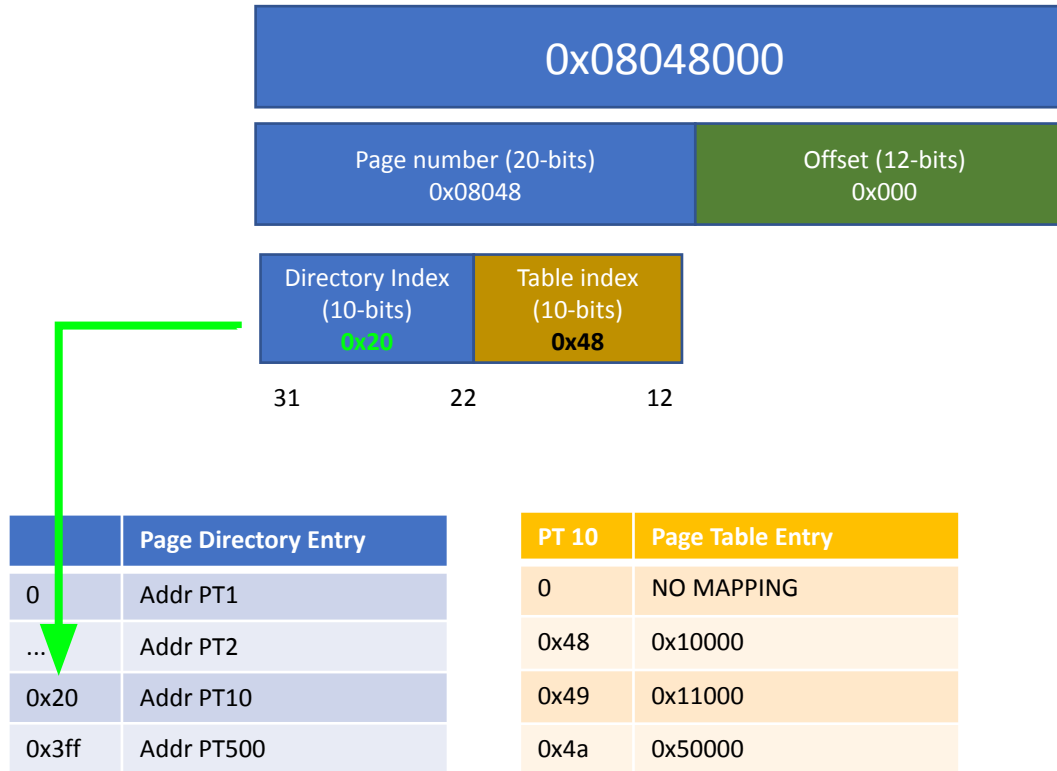


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

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

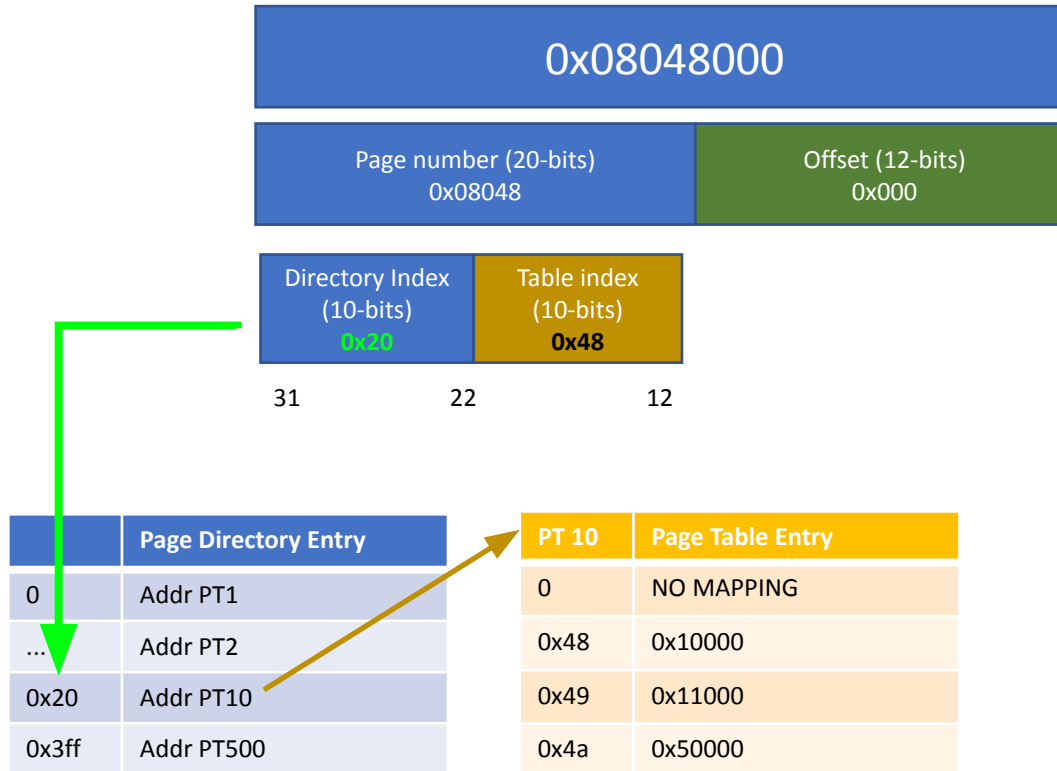
PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Address Translation Example



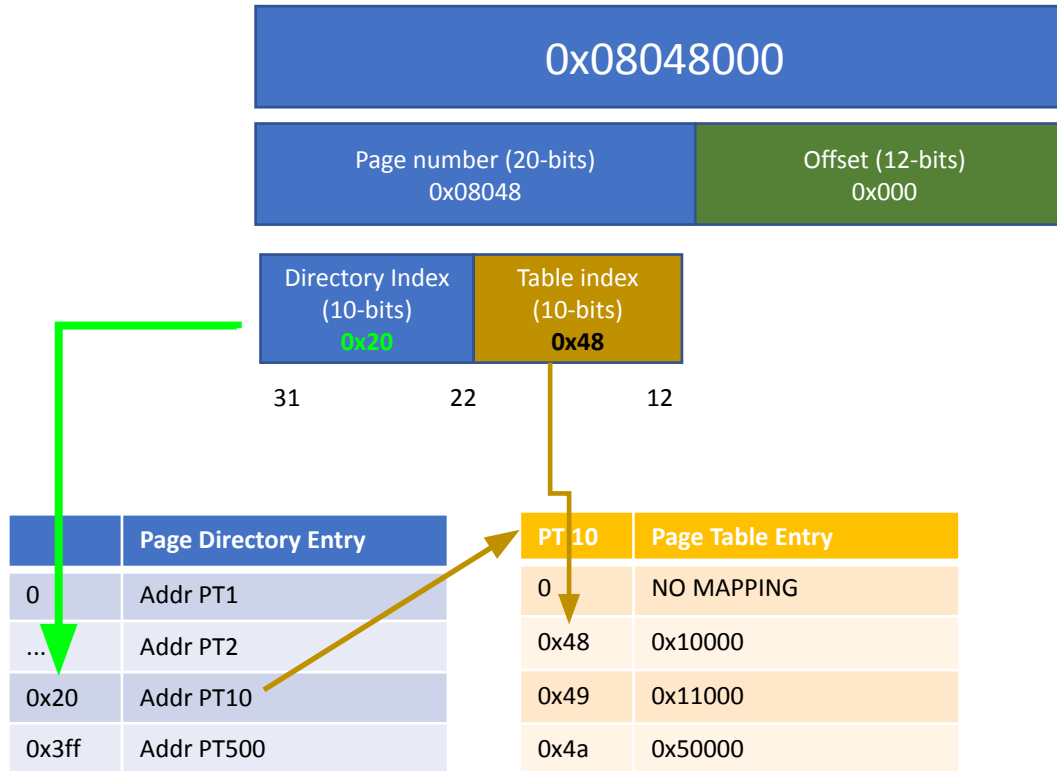
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Address Translation Example

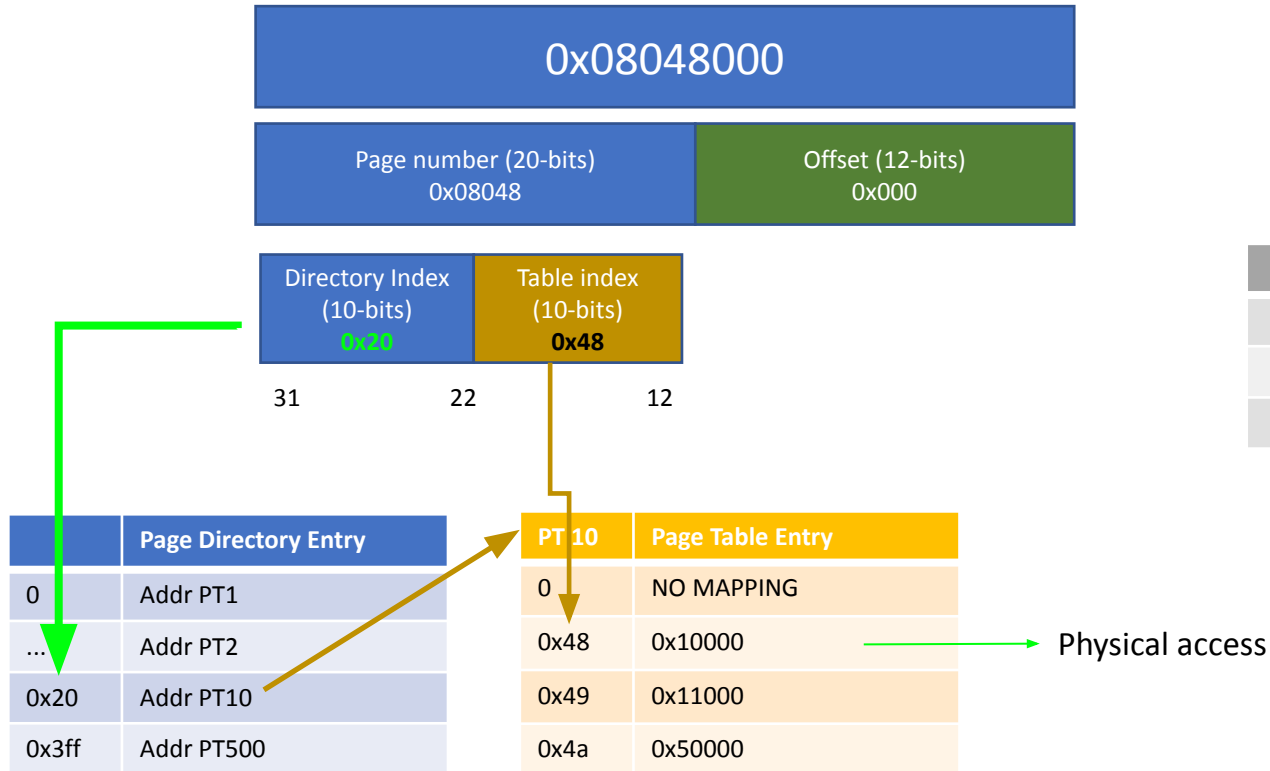


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

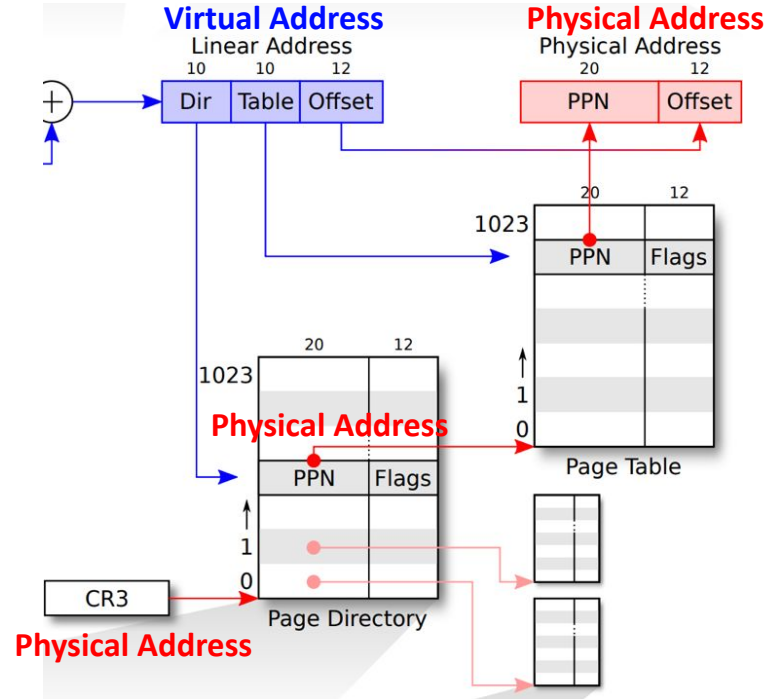
Address Translation Example



Address Translation Example



Addresses in use!



Page Directory and Page Table sizes



- Program wants to 1 KB of memory (0x8048000 -> 0x10000):

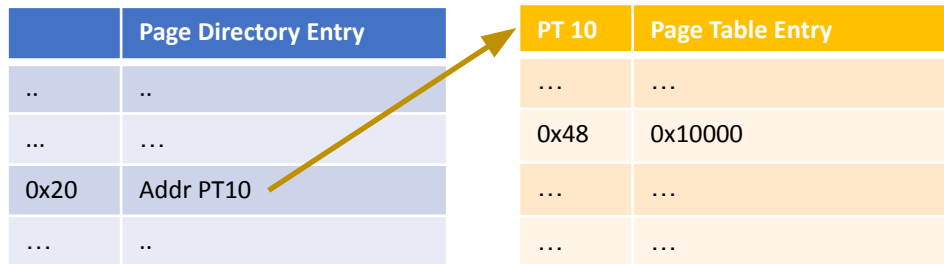
- Traditional page table:

- Page Table: $(2^{20}) * 4 \text{ bytes} = 4 \text{ MB}$.
- 1 page : 4 KB.
- **Overhead: 4 MB + 3 KB**

Virtual	Physical
0x8048000	0x10000

- Page directories:

- Page directory = $(2^{10}) * 4 \text{ bytes} = 4 \text{ KB}$.
- 1 Page table chunk = $(2^{10}) * 4 \text{ bytes} = 4 \text{ KB}$.
- 1 page : 4KB.
- **Overhead: 4 KB + 4 KB + 3 KB**



4KB

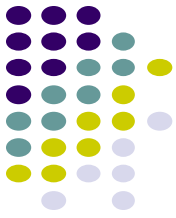
4KB

Notable features of x86 paging!



- We only ever want 4KB of contiguous memory:
 - Page directory: 4KB.
 - Page table chunk or Second level page table: 4KB.
 - Page size: 4KB.
- All metadata is of size 4KB!!

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



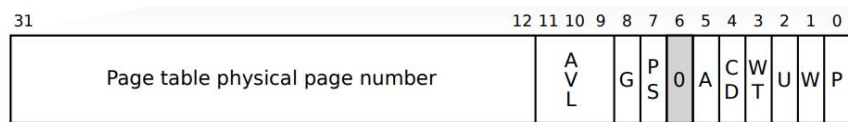
- Virtual address 0x8048338
 - Virtual page number: 0x8048
 - Offset: 0x338
 - Physical page number: 0x10
 - Physical address: $0x10\ 000 + 0x338 = 0x10338$

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

- Virtual address 0x443325af
 - Virtual page number: 0x44332
 - Offset: 0x5af
 - Physical page number: 0x33885
 - Physical address: $0x33885000 + 0x5af = 0x338855af$

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

PDE, PTE



PDE



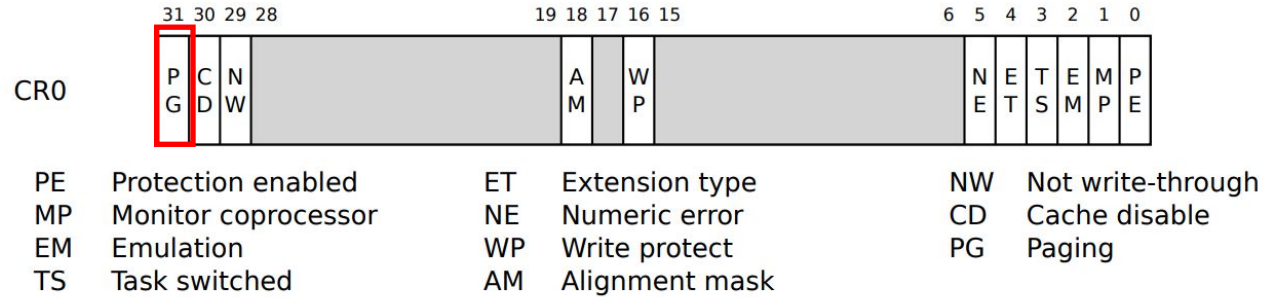
PTE

- 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

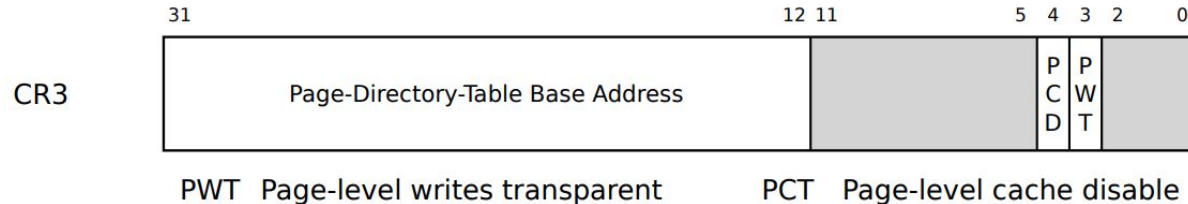
	Page Directory Entry
0	Addr PT1
...	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

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

- Can be enabled by CR0



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



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    Set cr3 to point the address of page directory
# Turn on paging.
movl    %cr0, %eax
orl    $(CR0_PE|CR0_PG|CR0_WP), %eax    Turn on CR0_PG!
movl    %eax, %cr0
```

- After CR0_PG is set -> Paging is in effect!

Accessing physical address

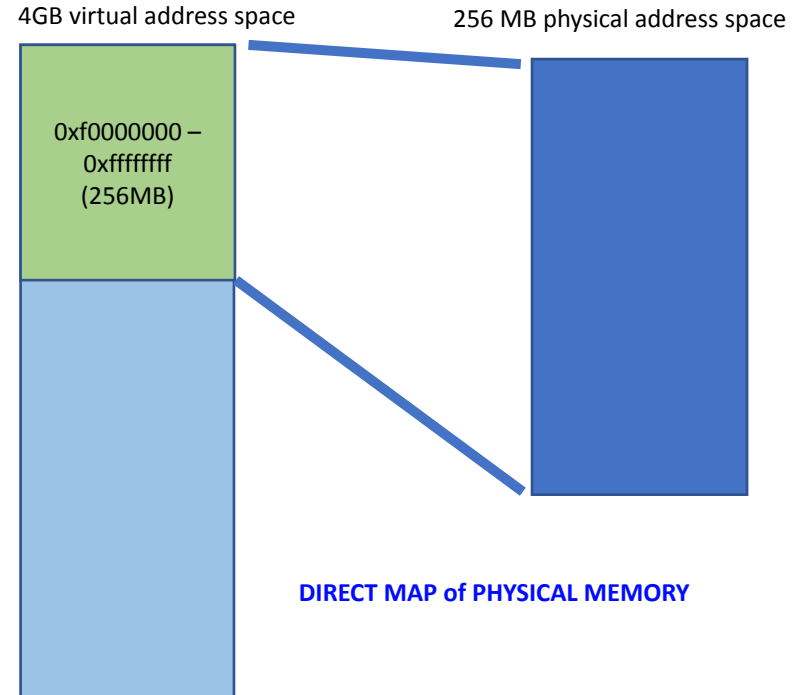


- After CR0_PG is up (paging is turned on)
 - All normal address access regarded as virtual address
 - Special addresses. E.g., CR3 will be considered as physical address
- What if we would like to access the physical address 0x100010?
 - Our kernel is at physical address 0x100000 ~ 0x110000
- What's the virtual addresses of that area?

Accessing physical address



- In JOS, we will map 0xf0000000 ~ 0xffffffff to
 - Physical address
 - 0x00000000 ~ 0x0ffffff
 - 256 MB Region
- 0xf0100010 -> 0x00100010
- 0xf1333358 -> 0x01333358



Setting up page directory in JOS



- In JOS (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)

Page table in JOS

- In JOS (kern/entrypgdir.c)

0xf0001000 will consult entry_pgtable[1]
-> 0x001000 | PTE_P | PTE_W

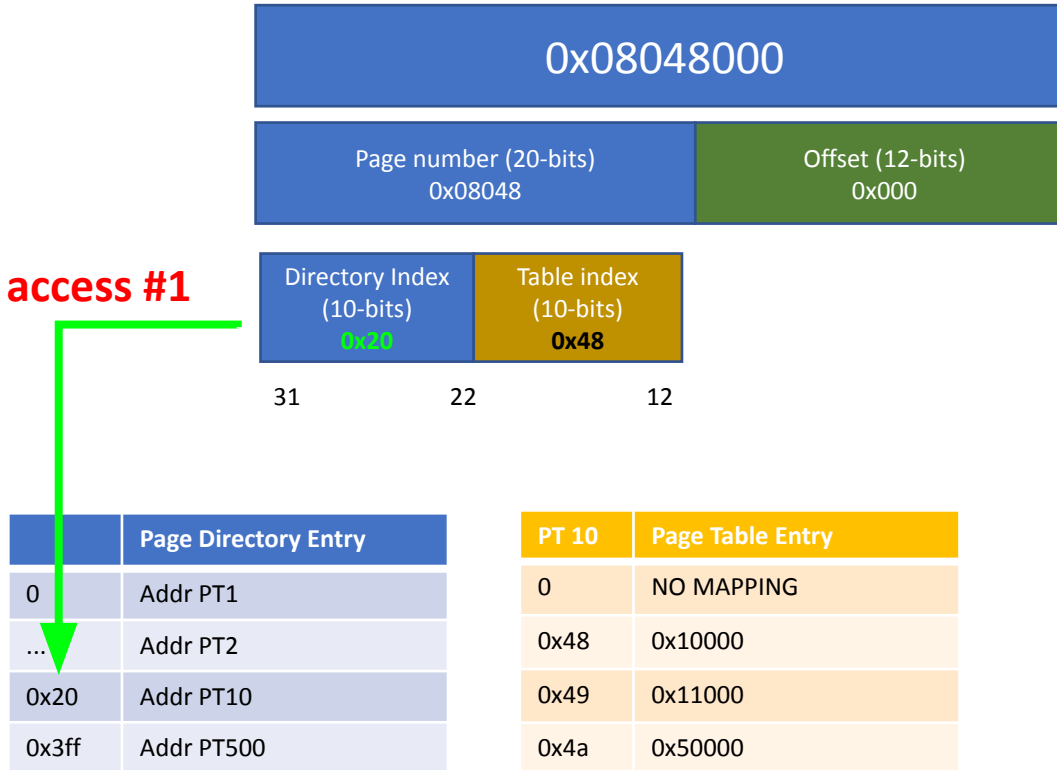
Phyaddr: 0x1000

```
__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,  
    ...  
};
```

Number of accesses for a translation!



Mem access #1

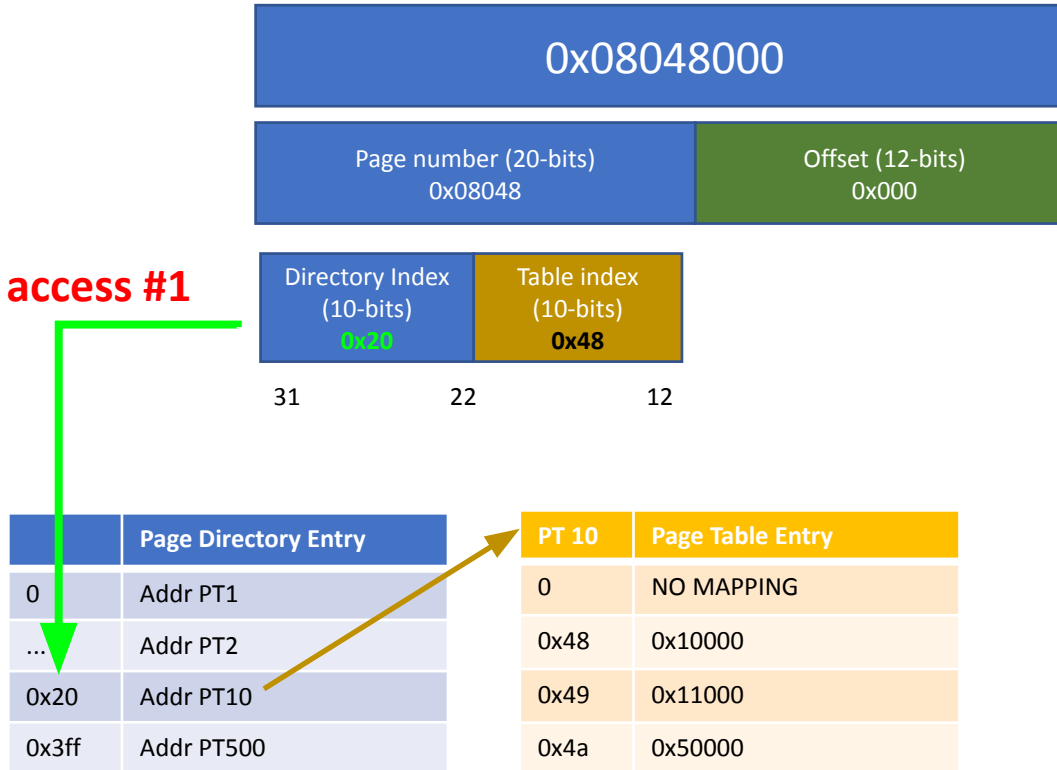


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Number of accesses for a translation!

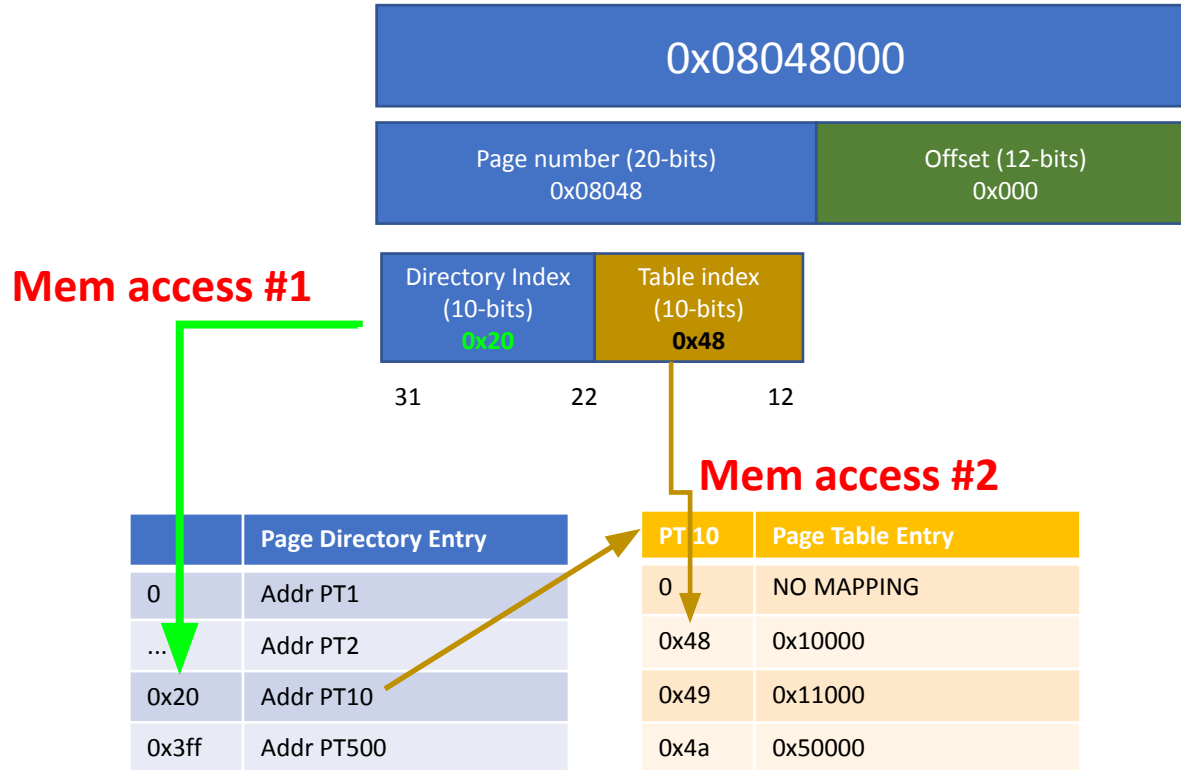


Mem access #1



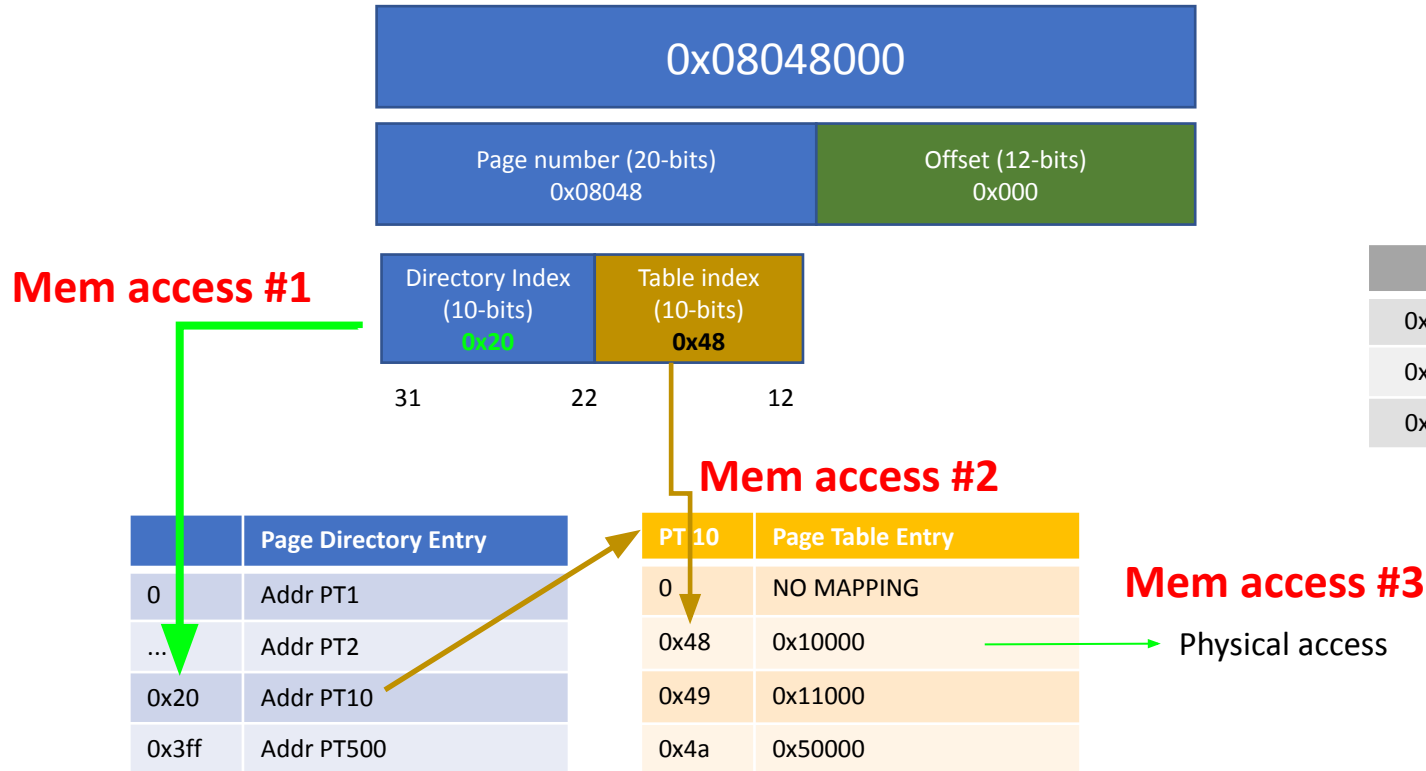
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Number of accesses for a translation!



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

Number of accesses for a translation!



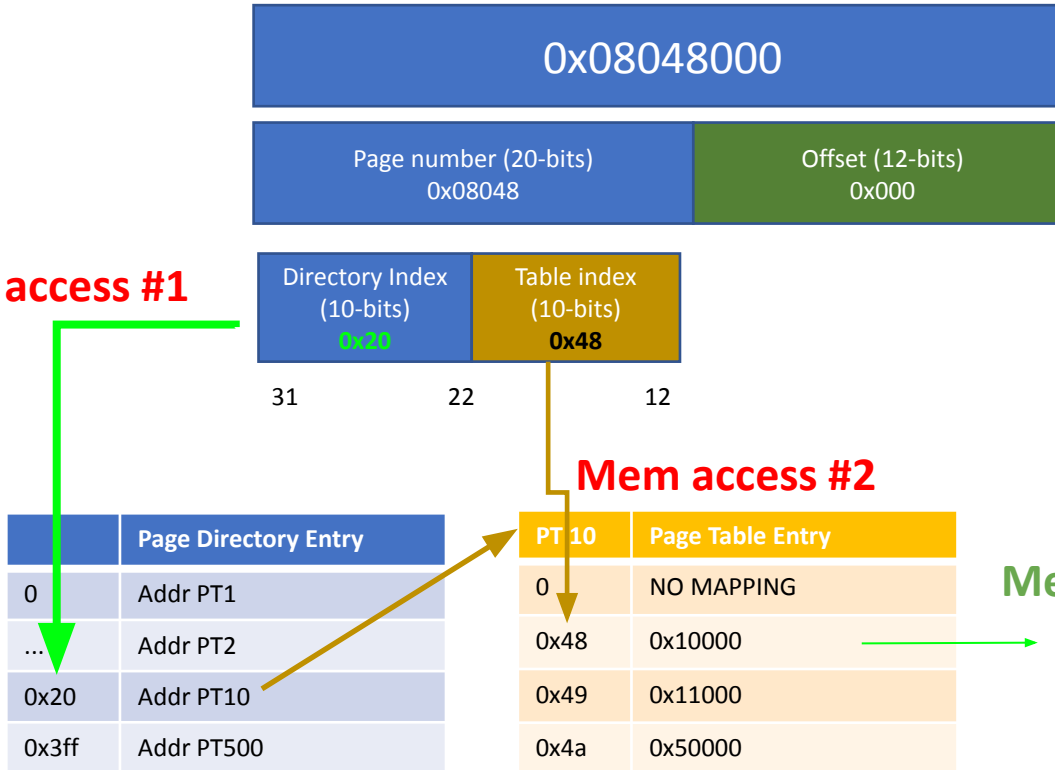
Number of accesses for a translation!



Expected: 1

Reality: 3 (2 additional)

Mem access #1



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

Access overhead!



- 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...
 - $200 \text{ (access the address)} + 1 \text{ (mov)} + 200 * 2 \text{ (page table...)} = 601 \text{ cycles.}$

How can we fix this?



- Hint: What do we do for regular memory?

Caching!



- Cache the frequently used page table entries:
 - Exploit locality
 - Translation Lookaside Buffer (TLB)
 - Cache for translation entries.

Translation Lookaside Buffer (TLB)



- Stores VA-PA mappings and caches them!

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

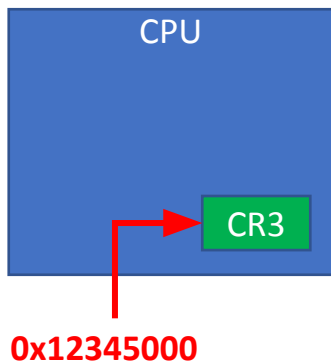
0x12345678 -> 0x678

0x12346678 -> 0x5678

0x12347678 -> 0xff678

0x12348678 -> 0xfff678

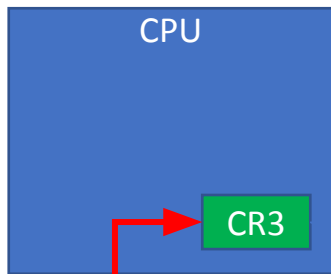
Without TLB



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

	Page Table Entry
0	Addr PT
0x345	0x10000
0x346	0x11000
0x347	0x50000

Without TLB

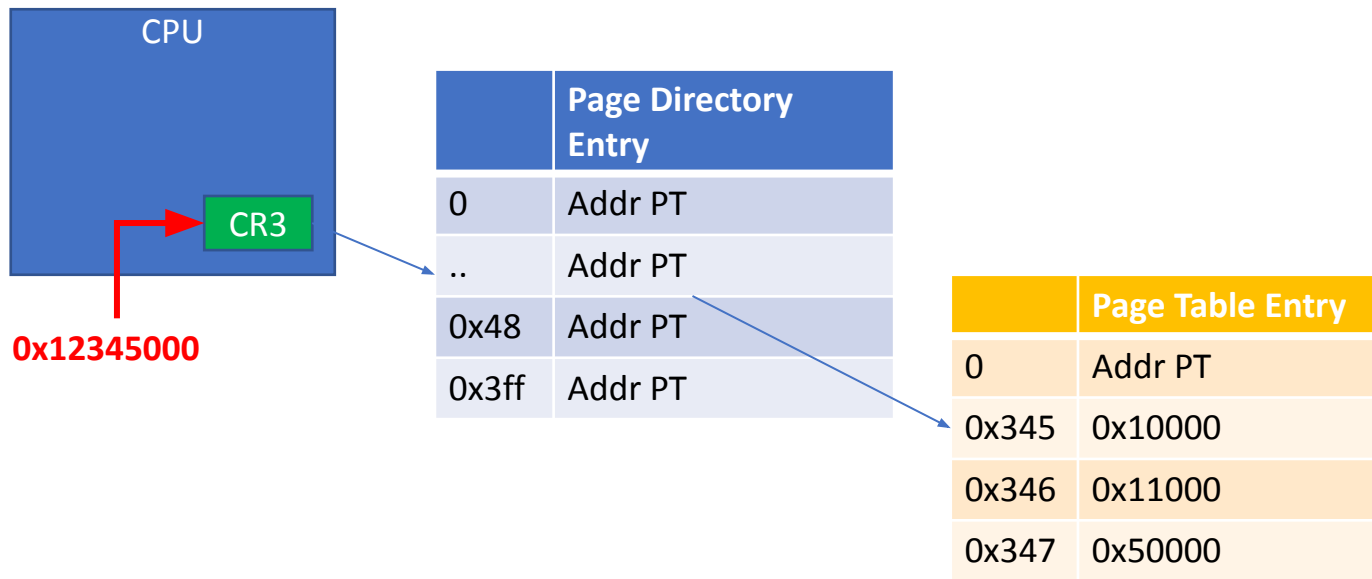
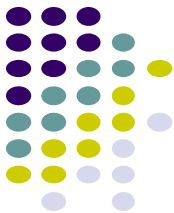


0x12345000

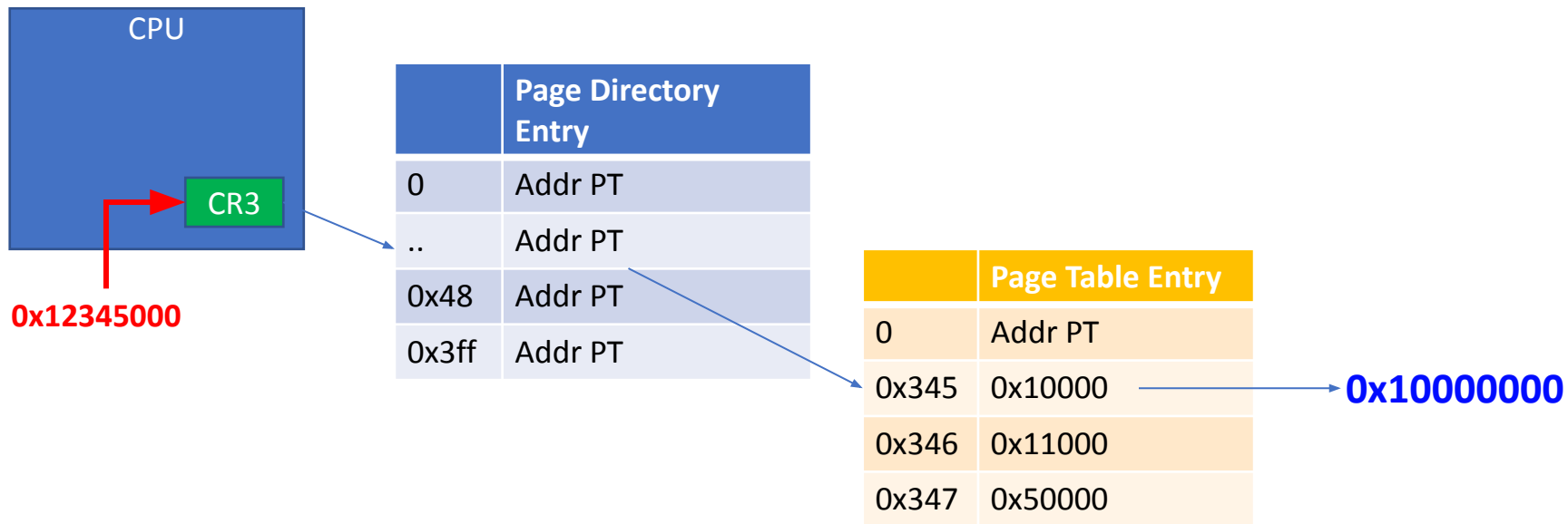
	Page Directory Entry
0	Addr PT
..	Addr PT
0x48	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x345	0x10000
0x346	0x11000
0x347	0x50000

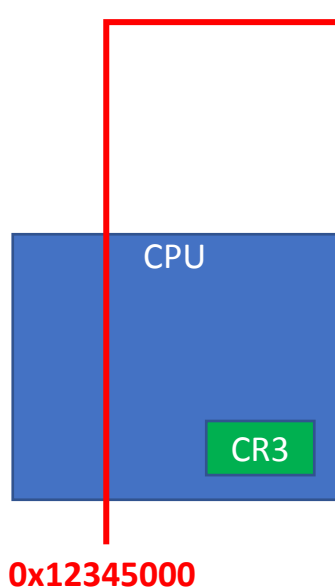
Without TLB



Without TLB



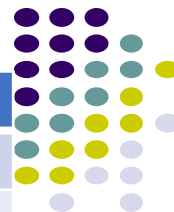
With TLB



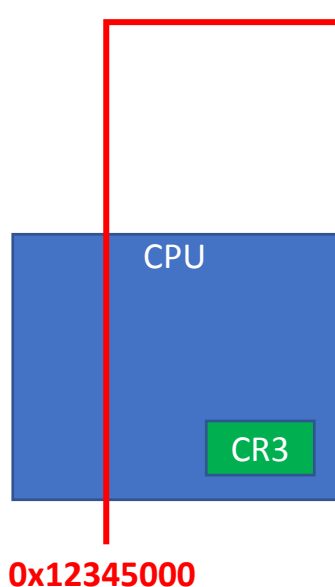
VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

	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



With TLB



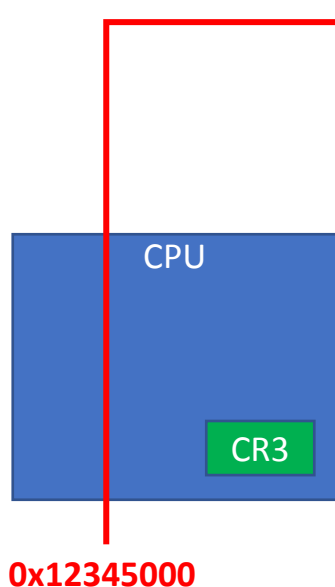
VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

	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

With TLB



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

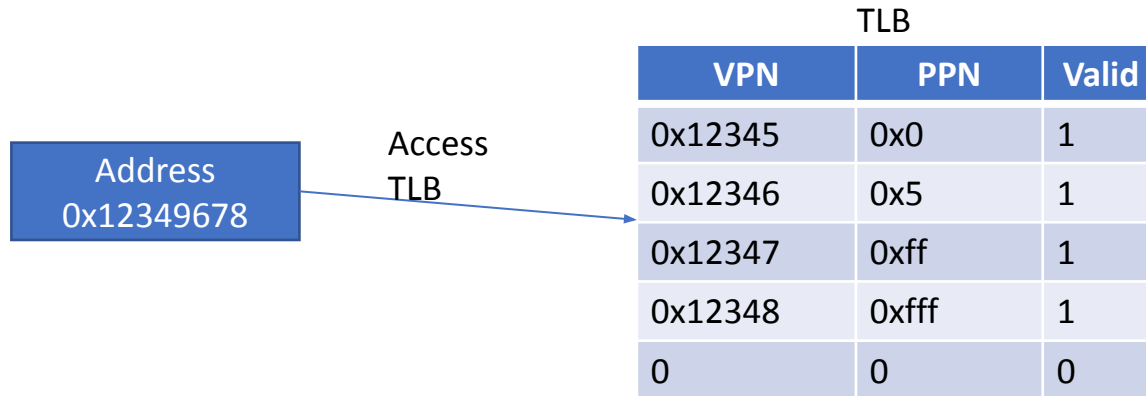
	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

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

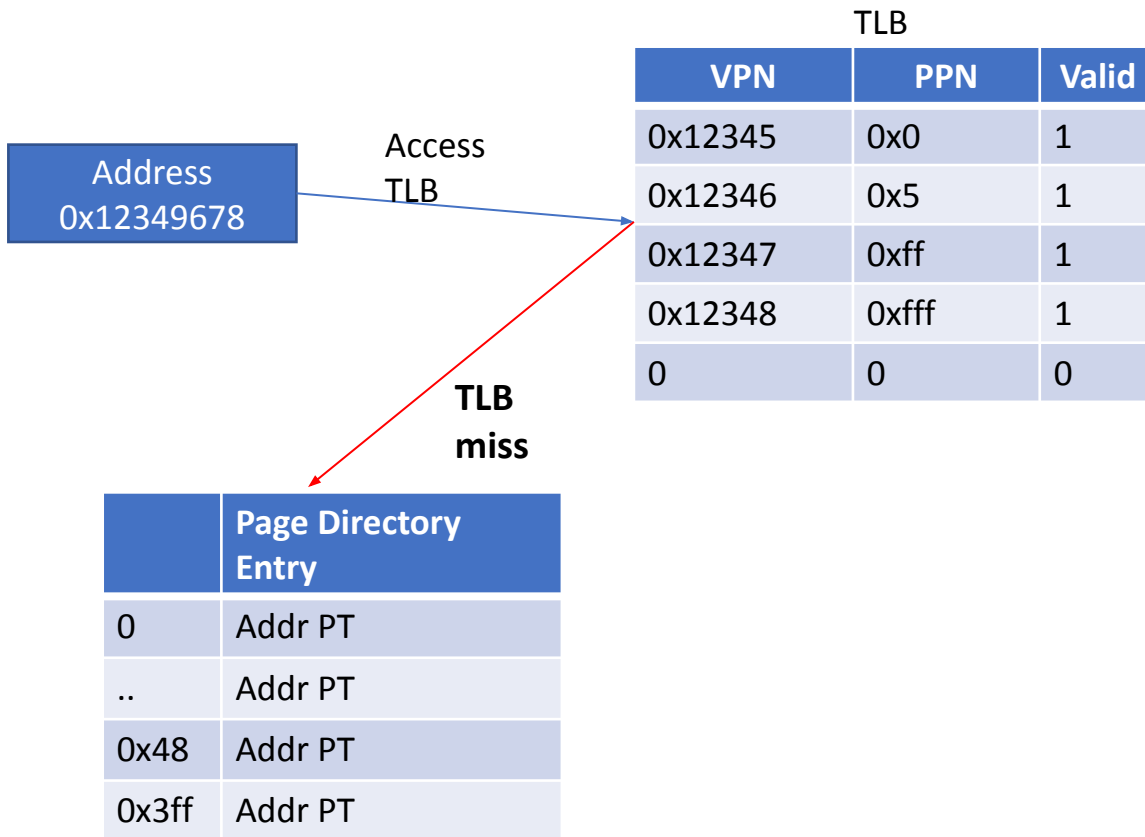
0x10000000

No page table access..

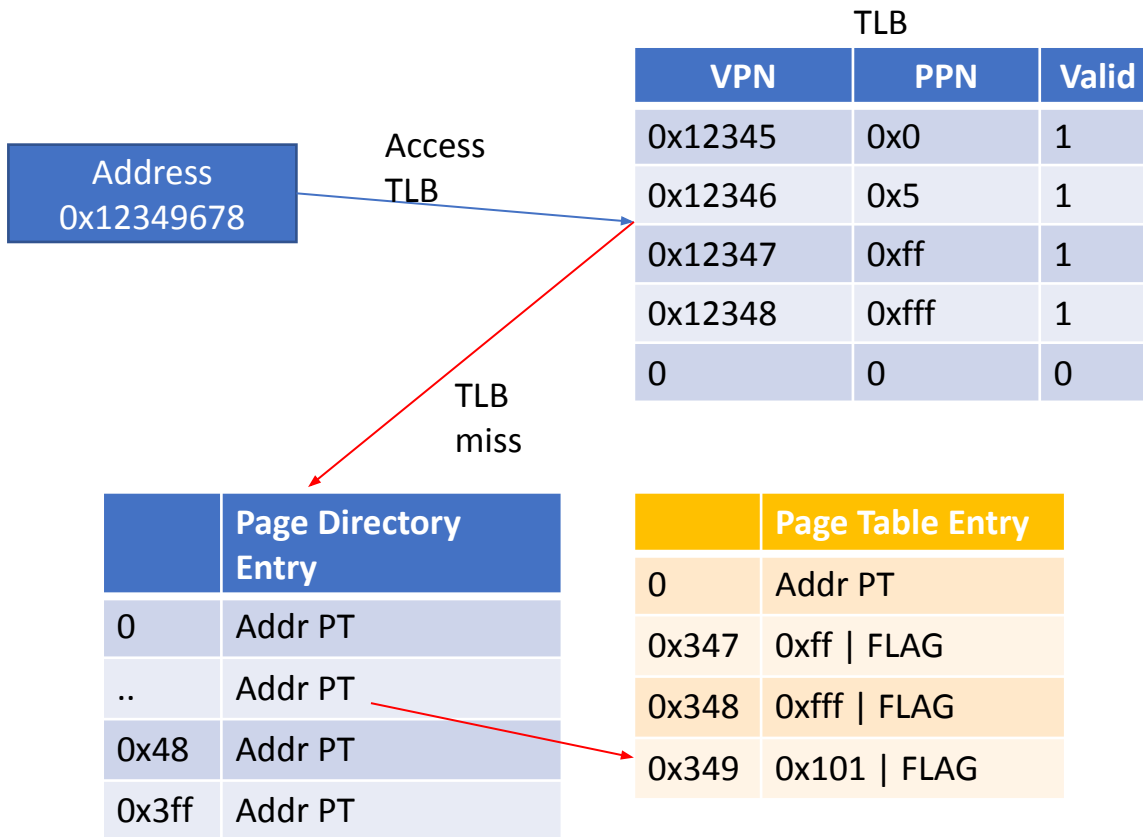
Populating TLB



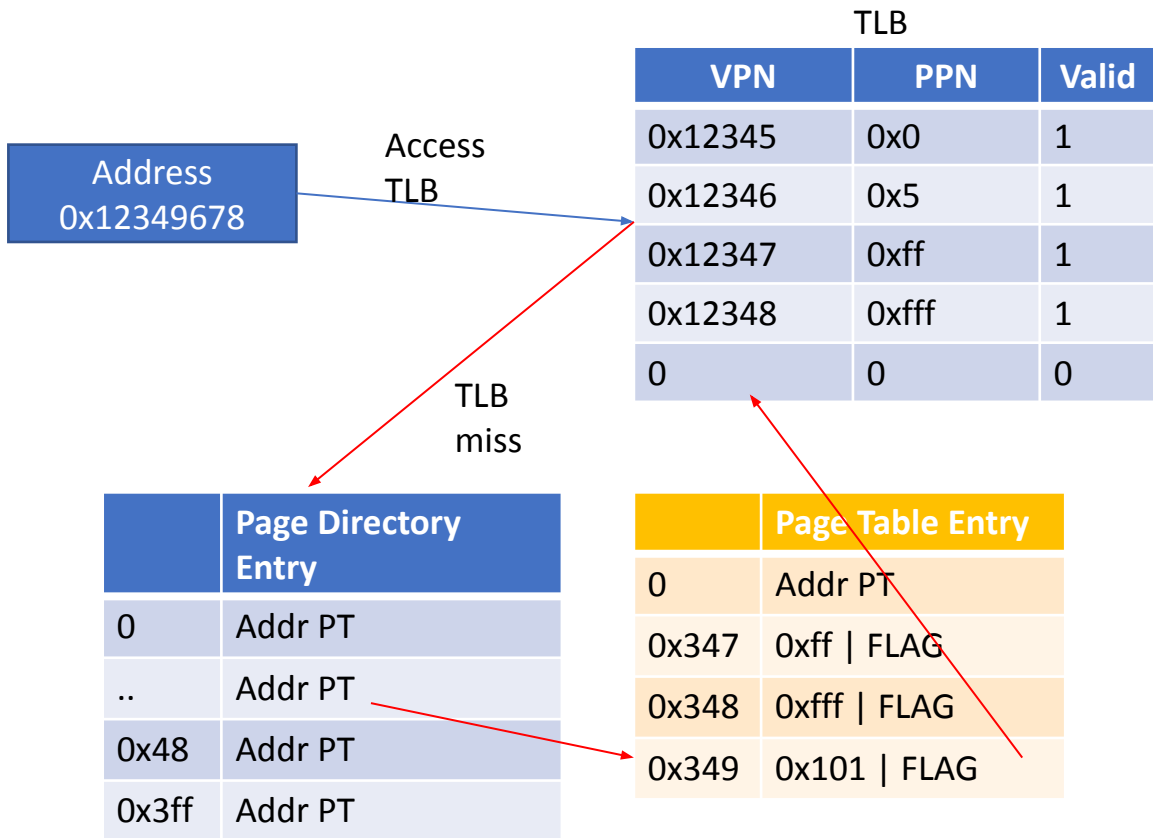
Populating TLB



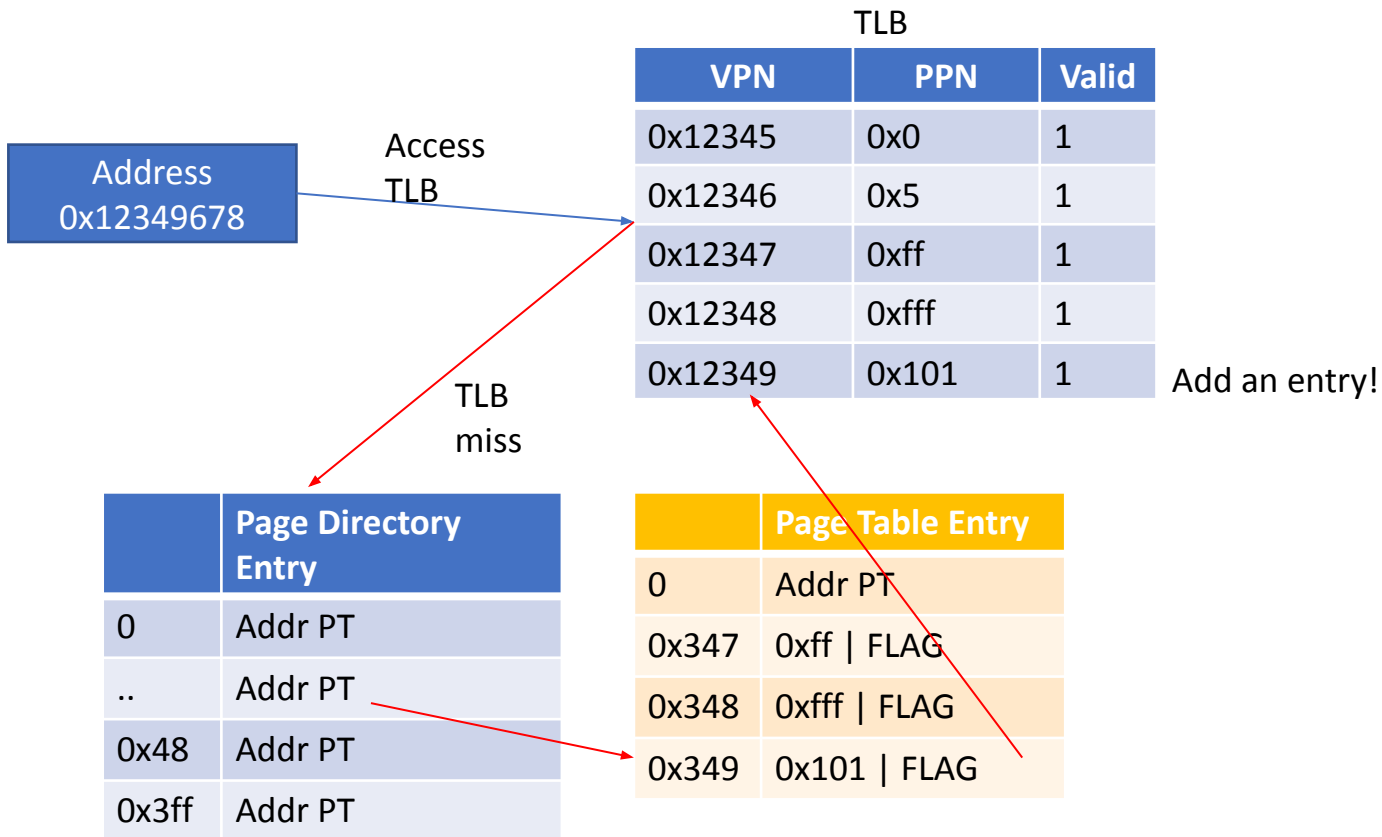
Populating TLB



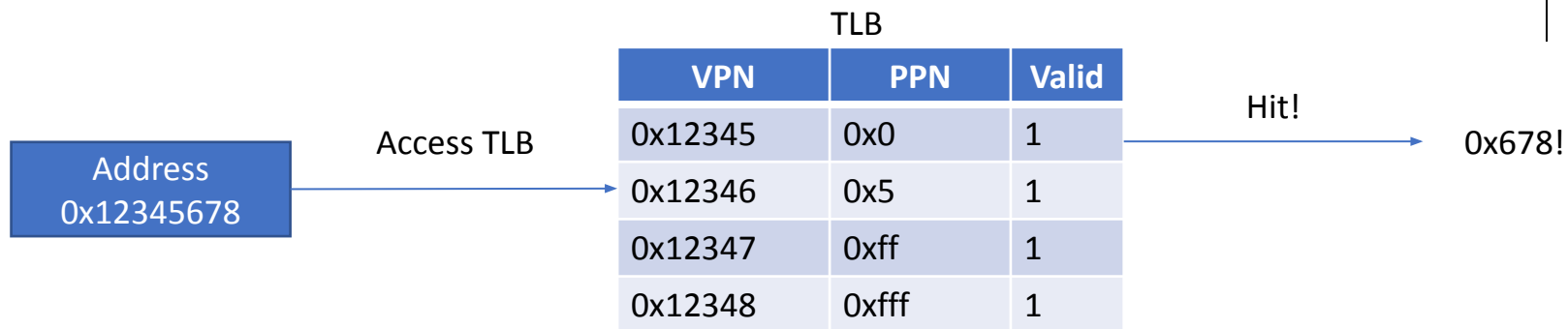
Populating TLB



Populating TLB



TLB hit



TLB Performance



- 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!

TLB Performance



- 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]
 $2\text{ns} + (10\text{ns} + 51\text{ns}) * 2 = 124\text{ns}$ (**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)

Limited TLB Size



- 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...

TLB Synchronization



- 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 Context Switch

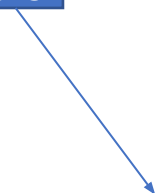


TLB

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 Context Switch



Address
0x12359888

CR3

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 Context Switch



TLB

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

Address

0x12359888

CR3

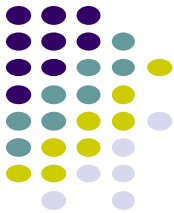
	Page Directory Entry
0	Addr PT
..	Addr PT
0x48	Addr PT
0x3ff	Addr PT

	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 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
0x12359888

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 <i>m</i> .



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");  
}
```

Manipulating Page Tables in JOS



- **PGNUM(x)**
 - Get the page number ($x \gg 12$) of the address x
- **PDX(x)**
 - Get the page directory index (top 10 bits) of the address x
- **PTX(x)**
 - Get the page table index (mid 10 bits) of the address x
- **PGOFF(x)**
 - Get the page offset (lower 12 bits) of the address x

Manipulating Page Tables in JOS



- `PTE_ADDR(pte)`
 - Get the physical address that a pte points
 - `PTE_ADDR(pte) + PGOFF(x)`
 - The physical address pointed by the PTE, translated by a virtual address `x`
- Translate a virtual address `va` to physical address `pa`
- `pte = CR3[PDX(va)][PTX(va)]`
 - `pte_t **cr3 = lcr3();` # or, `kern_pgdir`
 - `pte_t *pt = cr3[PDX(va)]`
 - `pte_t pte = pt[PTX(va)]`
- `physaddr_t pa = PTE_ADDR(pte) + PGOFF(va);`