CS444/544
Operating Systems II

Lecture 5

Virtual Memory Layout

4/15/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Oregon State University

Due Dates

- Lab 1 Due: tonight 11:59 pm
  - commit before tag

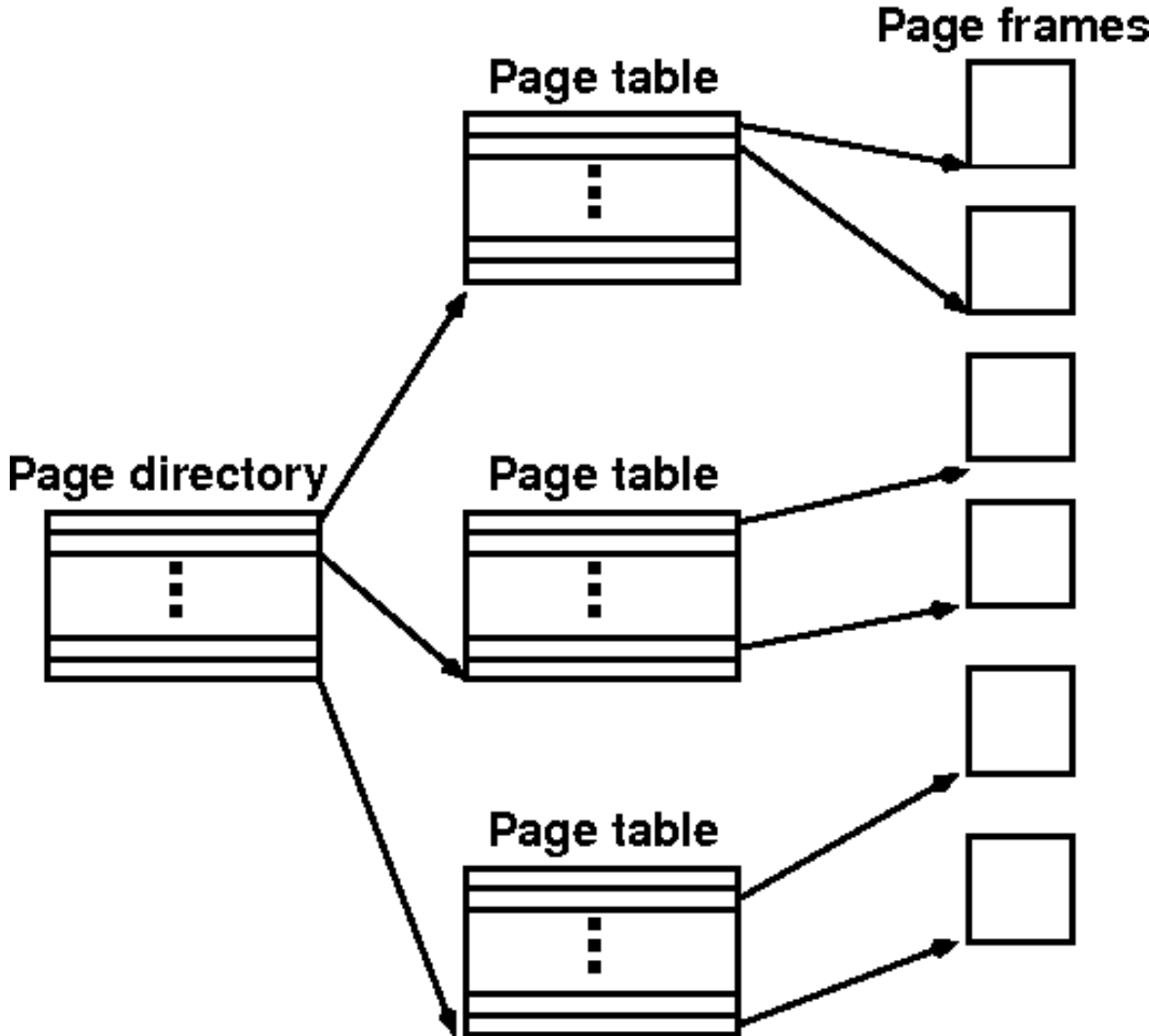
- Extra credits for challenge
  - Printing colors on console when typing ‘show’ as the command: +1%
  - Test with “make qemu” instead of “make qemu-nox”

Intel 32-bit Processor uses a 2-level page table

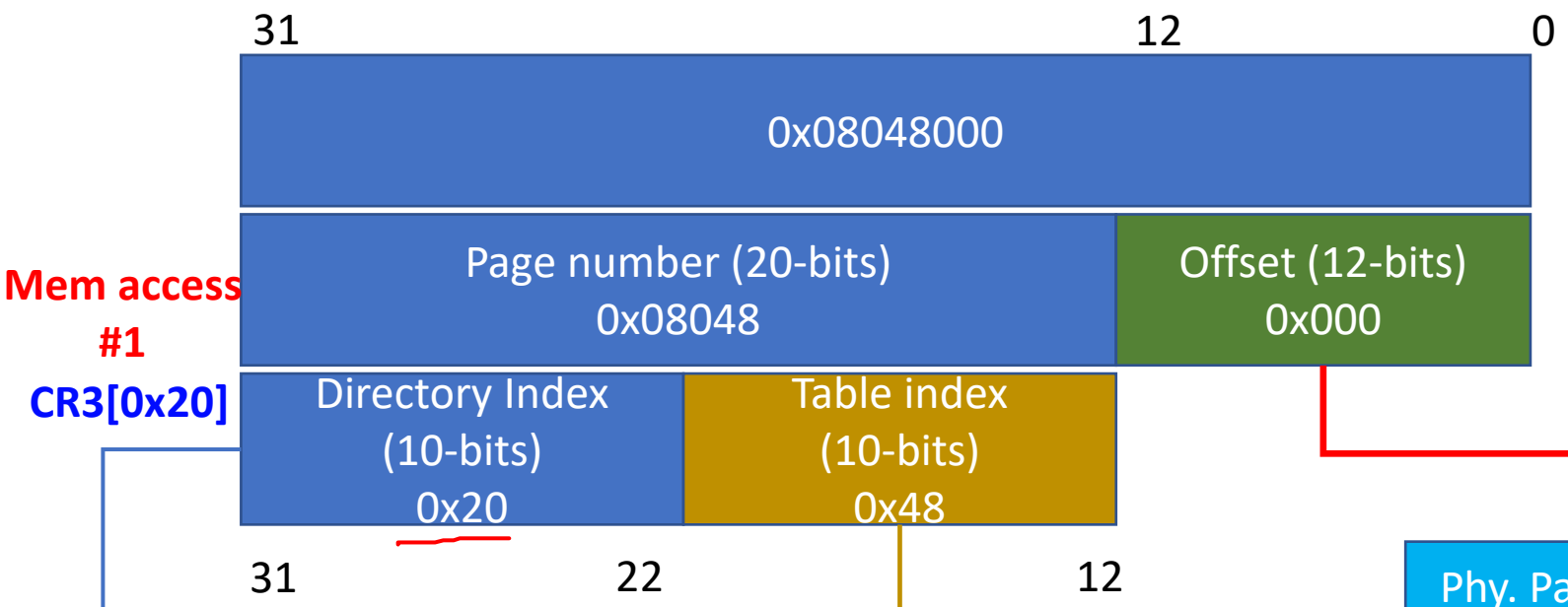
- Virtual address

- Page directory (level 1)
- Page table (level 2)

- Physical page!



Recap – Page Table & Addr Translation



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

Phy. Page number (20-bits)	Offset (12-bits)
0x10000	0x000

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

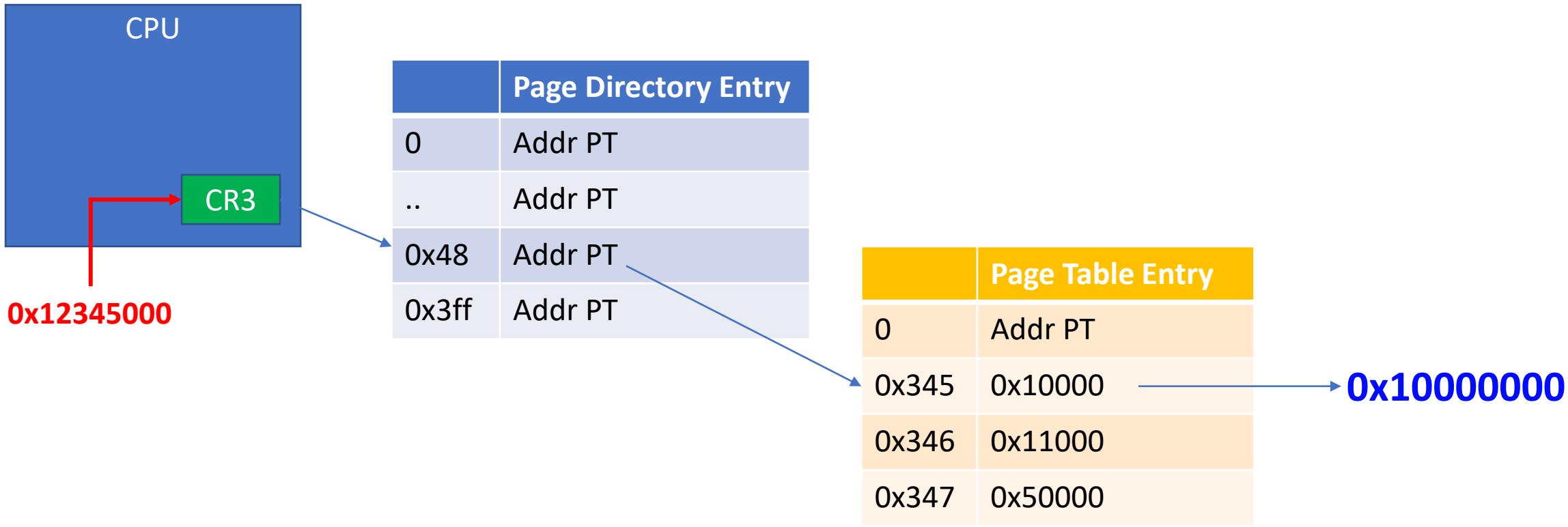
0x12346678 -> 0x5678

0x12347678 -> 0xff678

0x12348678 -> 0xfff678

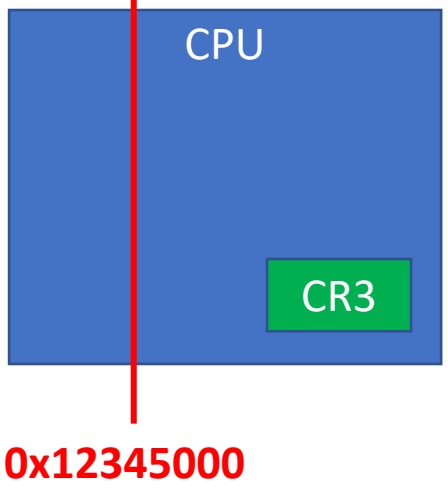
- Benefits
  - Do not have to walk down page tables for cached entries

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

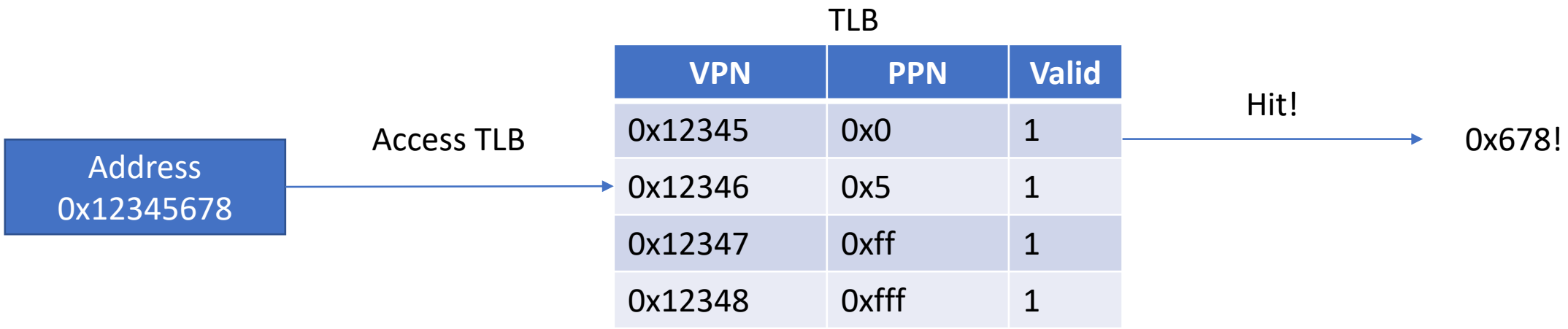
	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

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

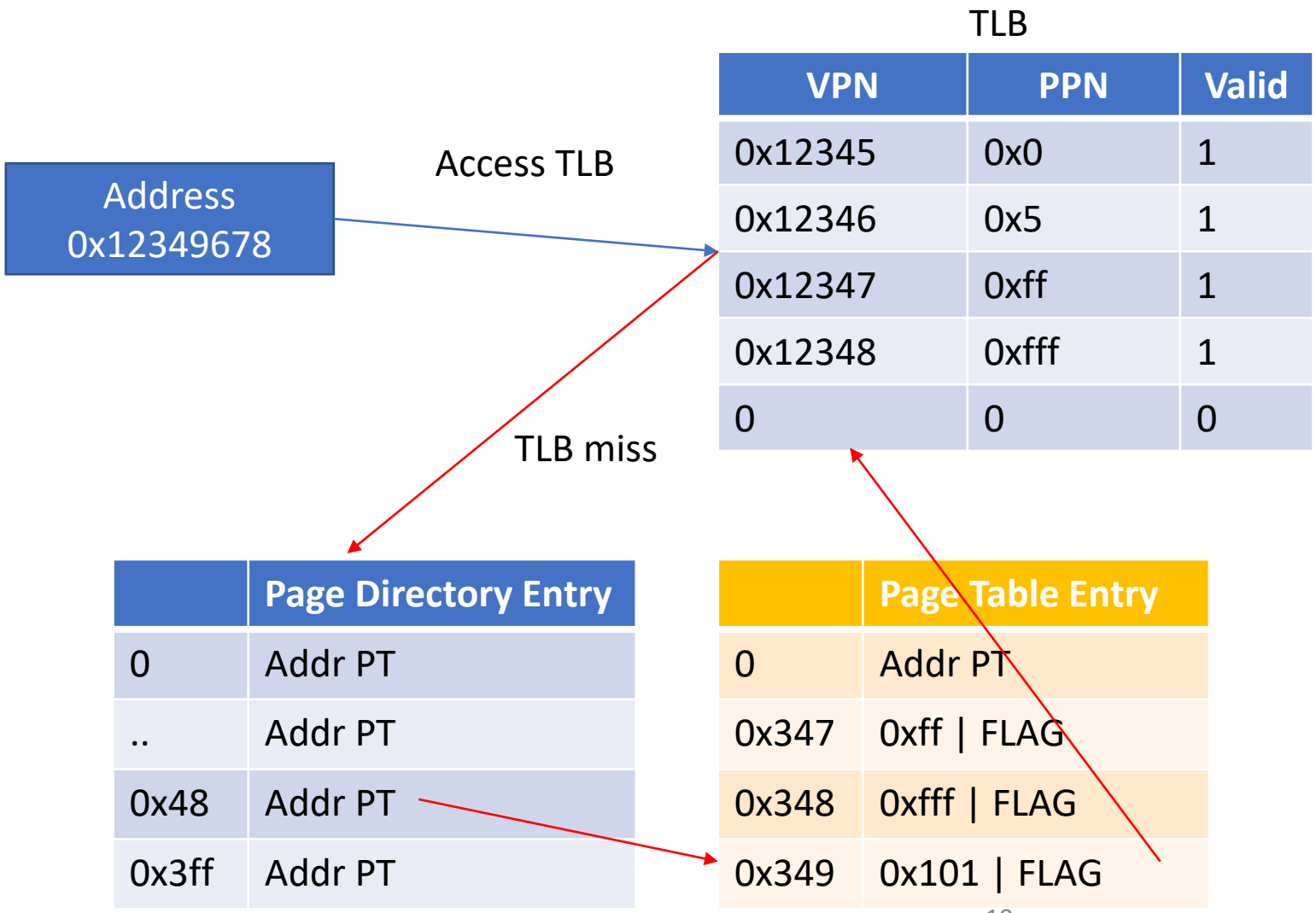
0x10000000

No page table access..

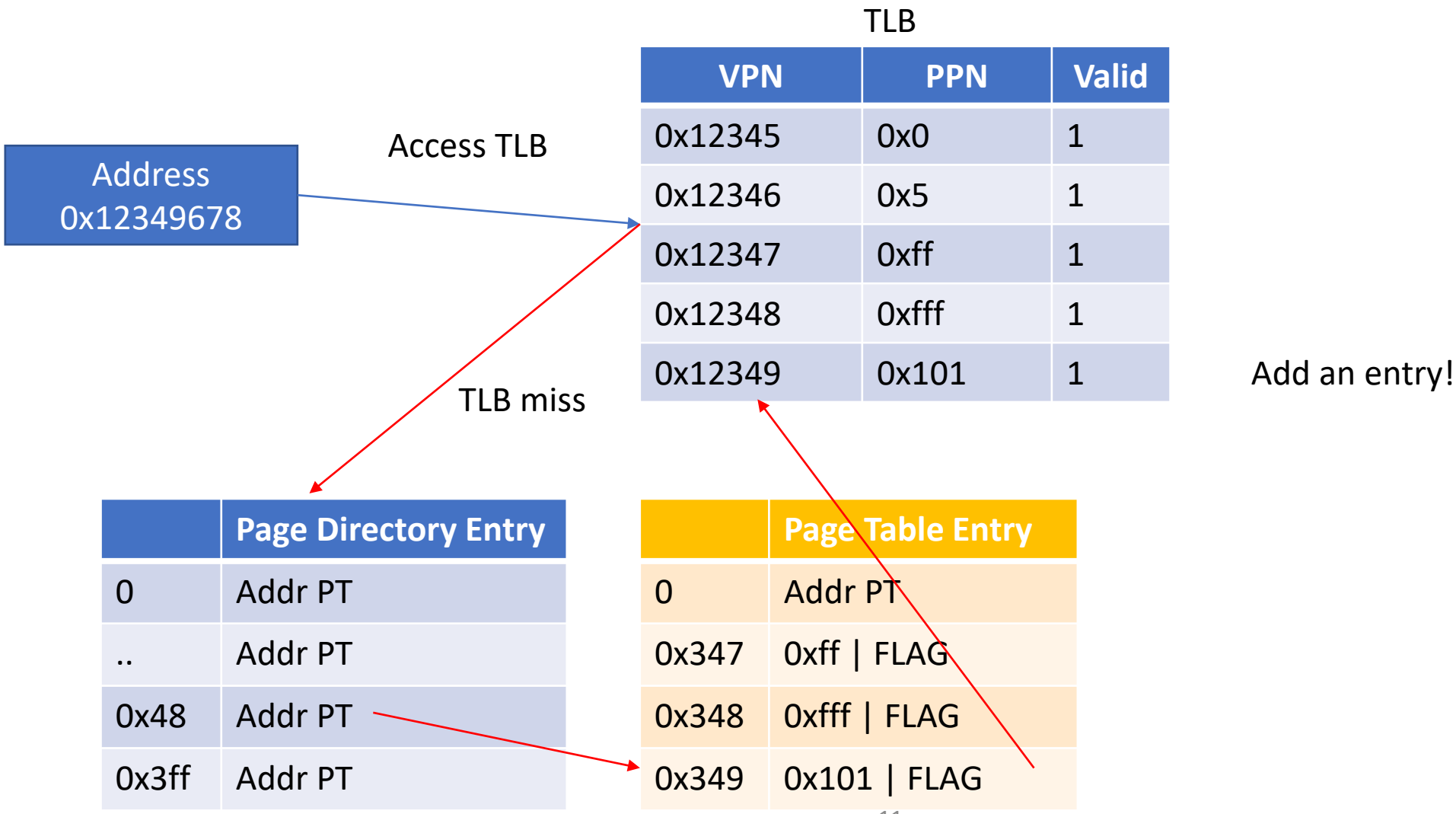
Address Translation with TLB (hit)



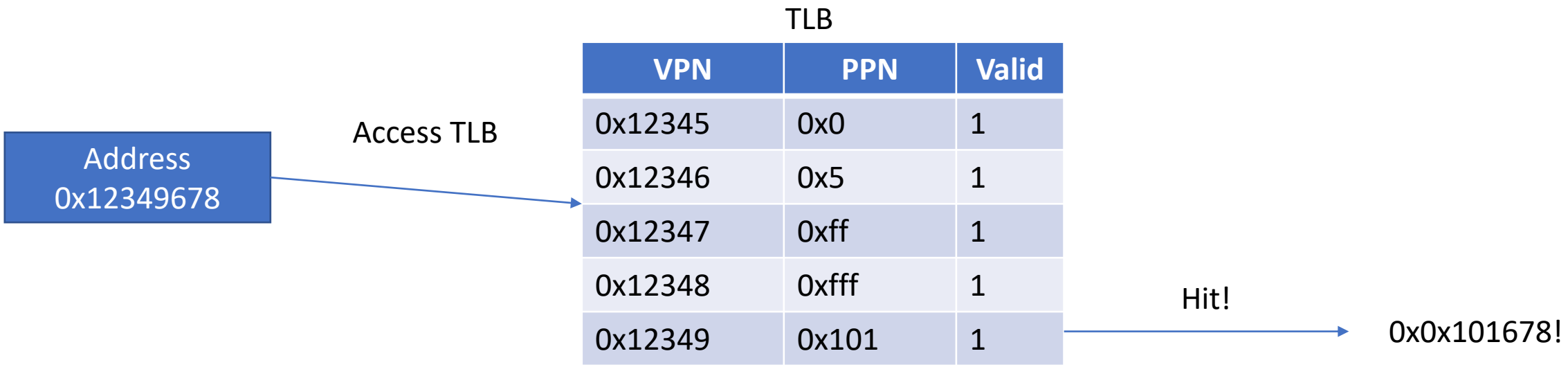
Address Translation with TLB (miss)



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…

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

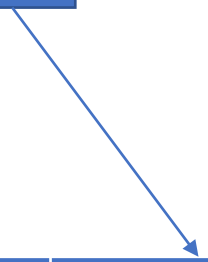
TLB and Process Context Switch

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

Address
0x12349678

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

	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

Address
0x12349678

CR3

We need to invalidate all previous entries..

	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?

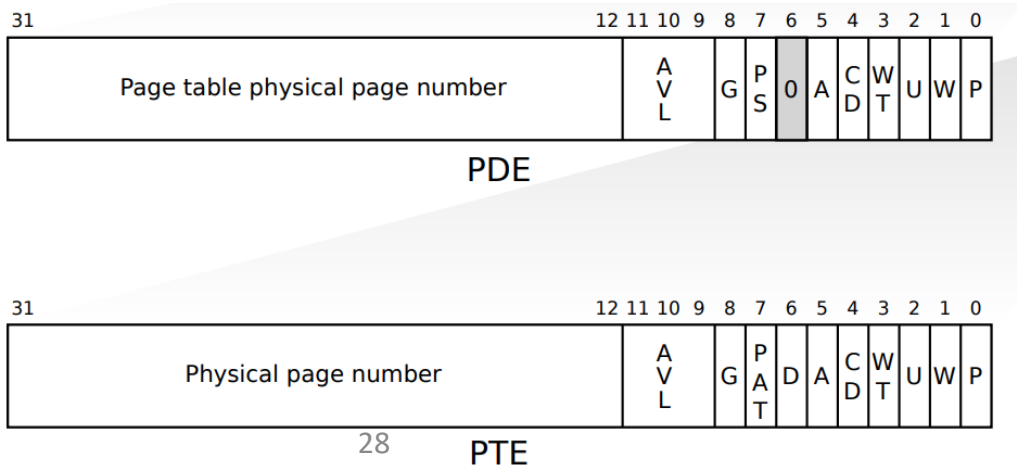
Topics

- Page Permissions
- Virtual Memory Layout
- How JOS Manages Physical Memory?

Page Directory / Table Entry (PDE/PTE)

- Top 20 bits: physical page number
  - Physical page number of a page table (PDE)
  - Physical page number of the requested virtual address (PTE)

- Lower 12 bits: some flags
  - Permission
  - Etc.

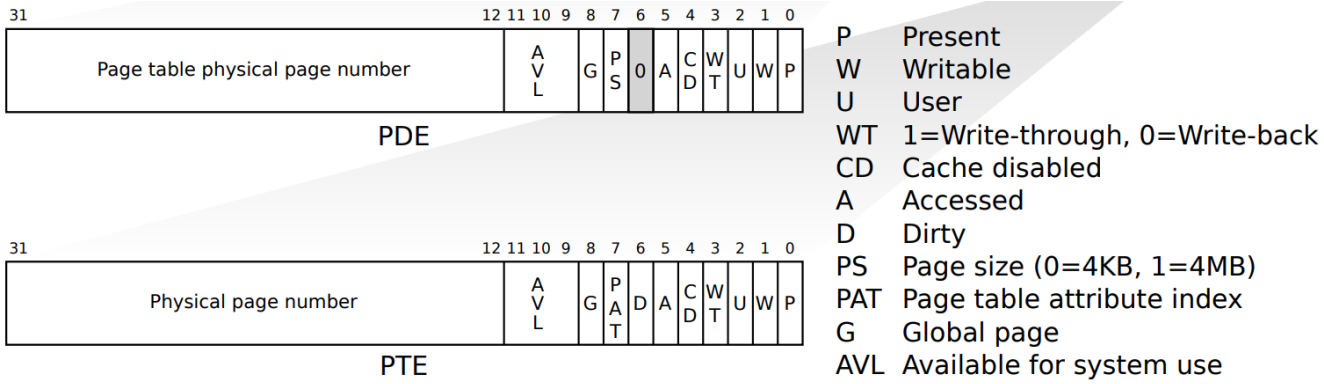


P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Permission Flags

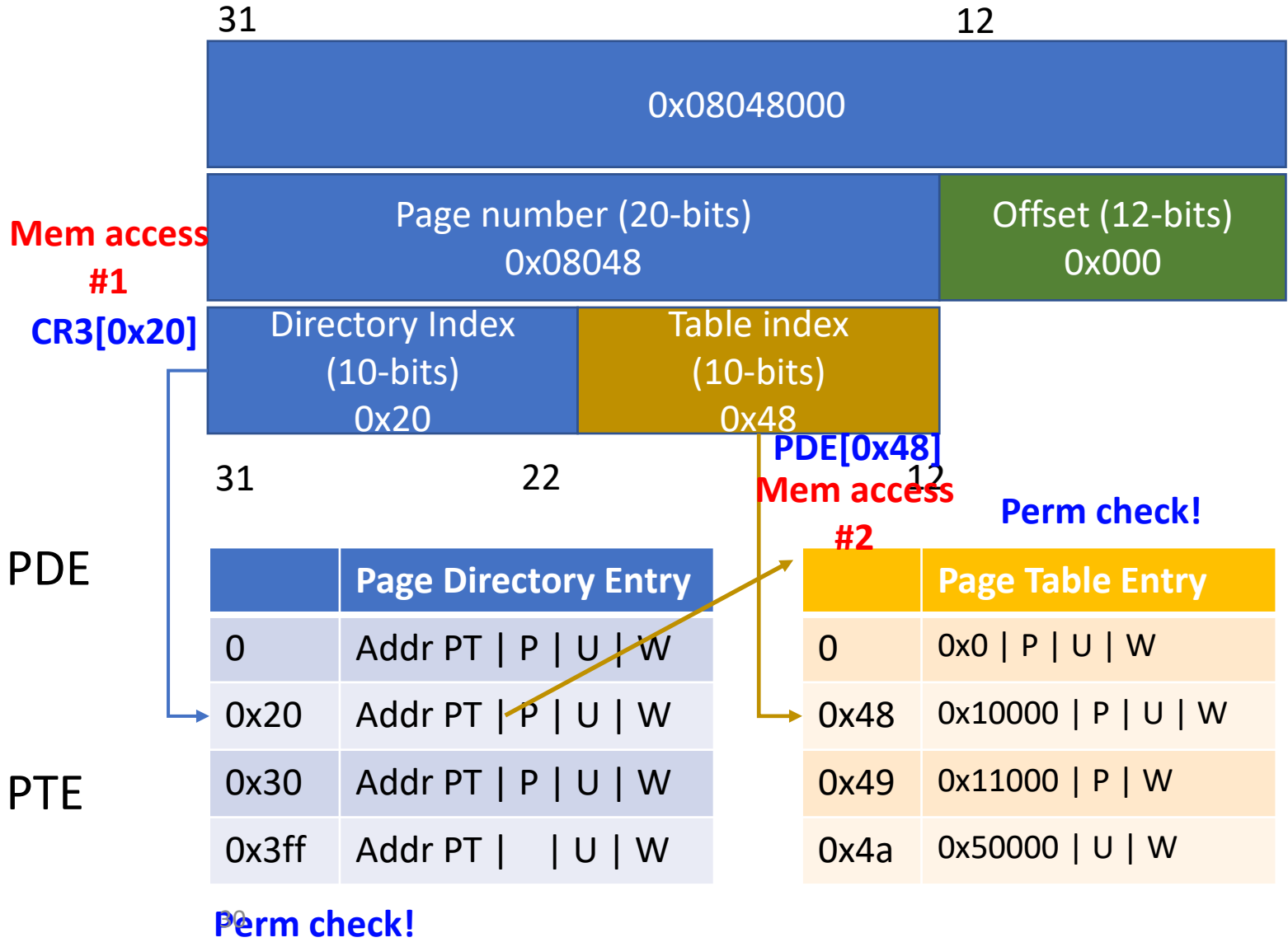
- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

	Page Table Entry	
0	Addr PT	
0x48	0x10000 << 12 | PTE_U | PTE_W	Invalid
0x49	0x11000 << 12 | PTE_P | PTE_W	Kernel, writable
0x4a	0x50000 << 12 | PTE_P | PTE_U	User, read-only



When does CPU check Permission Bits?

- In address translation
- 1. Virtual address
- 2. PDE = CR3[PDX]
  - Checks permission bits in PDE
- 3. PTE = PDE[PTX]
  - Checks permission bits in PTE



CPU checks PDE permission first and then PTE permission next…

- A virtual memory address is inaccessible if PDE disallows the access
- A virtual memory address is inaccessible if PTE disallows the access
- Both PDE and PTE should allow the access…

PDE/PTE Permission Examples 0

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- Valid, accessible by ring 3, and writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 1

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P | PTE_U
- Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 2

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_P | PTE_W | PTE_U
- Valid, accessible by ring 3, but not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 3

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W | PTE_U
- PTE: PTE_P
- valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 4

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 5

- Virtual address 0x01020304
- PDE: PTE_P | PTE_U
- PTE: PTE_U
- invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

PDE/PTE Permission Examples 6

- Virtual address 0x01020304
- PDE: PTE_U
- PTE: PTE_P | PTE_U
- invalid

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

Can you setup a page permission as…

- Kernel: R, User: --
  - PTE_P
- Kernel: R, User: R
  - PTE_P | PTE_U
- Kernel: RW, User: RW
  - PTE_P | PTE_U | PTE_W

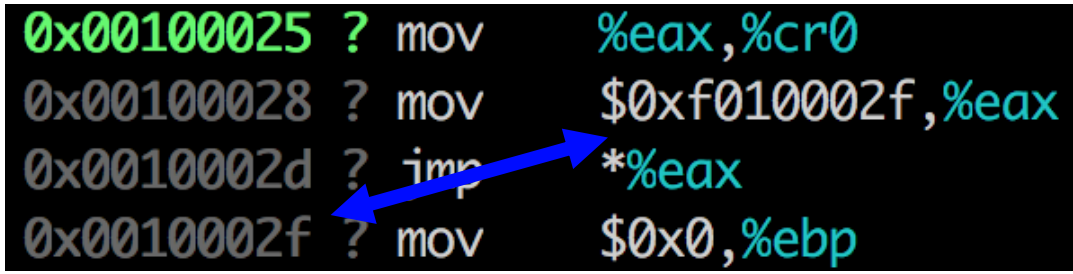
You can’t setup a page permission as…

- Kernel: RW, User: R
  - PTE_P | PTE_W | PTE_U -> User RW…
  - PTE_P | PTE _W -> User --
- Kernel: R, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…
  - PTE_P | PTE_U -> User R…
- Kernel: --, User: RW
  - PTE_P | PTE_U | PTE_W -> Kernel RW…

You can enable such a conflicting permission setup by having N-to-1 mapping

- Virtual to physical address mapping is in N-to-1 relation
  - N number of virtual addresses could be mapped to 1 physical address

- E.g., for a physical address 0x100000
  - JOS maps VA 0x100000 to PA 0x100000
  - JOS maps VA 0xf0100000 to PA 0x100000



- Why?
  - EIP before enabling paging: 0x100025
  - EIP after enabling paging: 0x100028

Sharing a Physical Page

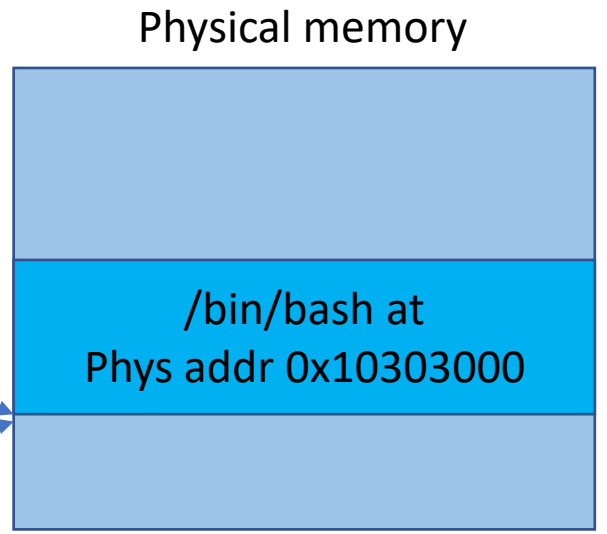
- Example: Loading of the same program

Process 0, runs /bin/bash,
loads at virt addr 0x35555000

	Page Table Entry
0	…
0x155	0x10303 | FLAG

Process 1, runs /bin/bash,
loads at virt addr 0x43132000

	Page Table Entry
0	…
0x132	0x10303 | FLAG



2 or more mappings to 0x10303000 is possible!

You can’t setup a page permission as…

- Kernel: RW, User: R
  - VA 0x00001000 -> PA 0x50000, PTE_P | PTE_U (User R)
  - VA 0xf0050000 -> PA 0x50000, PTE_P | PTE_W (Kernel RW)
- Kernel: R, User: RW
  - VA 0x00002000 -> PA 0x60000, PTE_P | PTE_U | PTE_W (User RW)
  - VA 0xf0060000 -> PA 0x60000, PTE_P (Kernel R)

- Kernel: --, User: RW
  - VA 0x00003000 -> PA 0x70000, PTE_P | PTE_U | PTE_ W
  - VA 0xf0070000 -> PA 0x70000, 0 for flag…

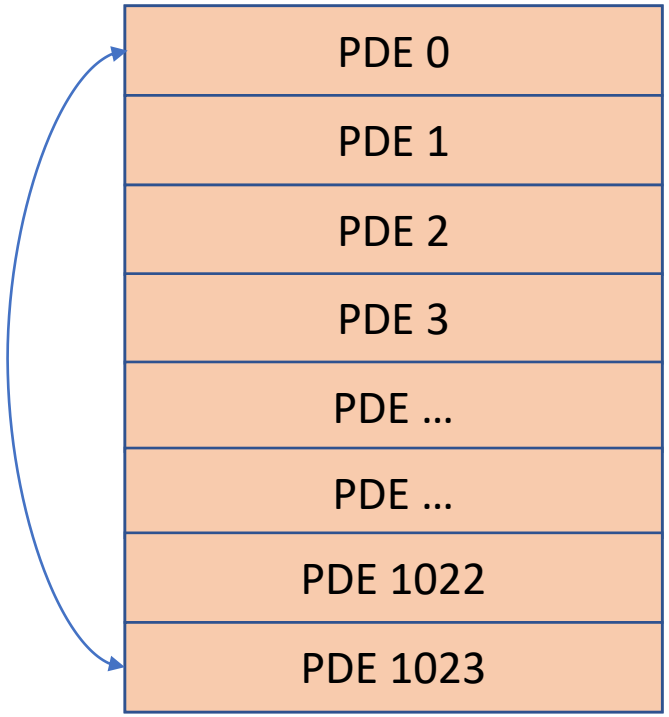
PDE/PTE Permissions CAVEAT

- A virtual address access is allowed if both **PDE** and **PTE** entries allows the access…
- General practice: put **a more permissive** permission bits in **PDE**, and **be strict** on setting permission bits in **PTE**
- For a conflicting permission setup for Kernel/User, add **an additional virtual address mapping** can enable such a setup

How To Create a Page Directory?

- For a 32-bit Intel processor, we use only 1 page for a Page Directory

One page, 4KB



4096 / **4** = 1024 entries

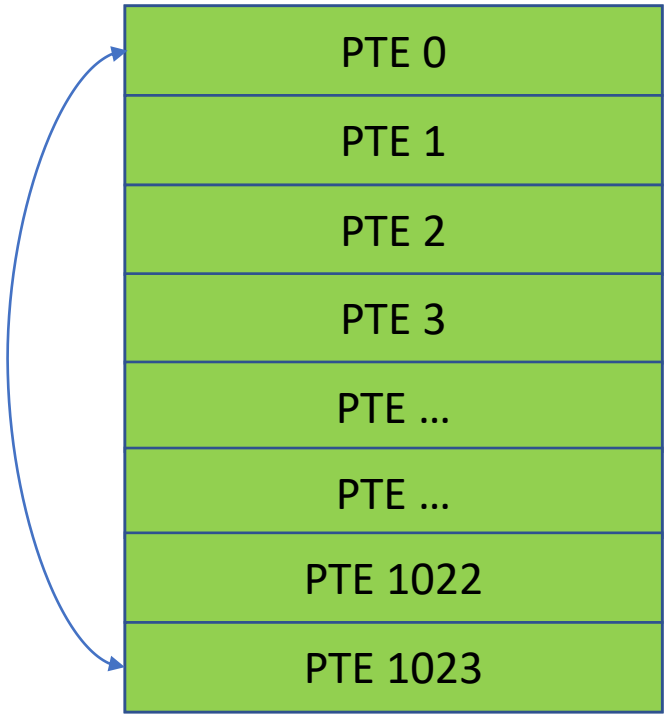
1024 == 2^{10}

10-bit index for PD

Each entry is 4-byte (32 bits)

How To Create a Page Table?

- For a 32-bit Intel processor, we use only 1 page for a Page Table



One page, 4KB

4096 / **4** = 1024 entries

1024 == 2^{10}

10-bit index for PT

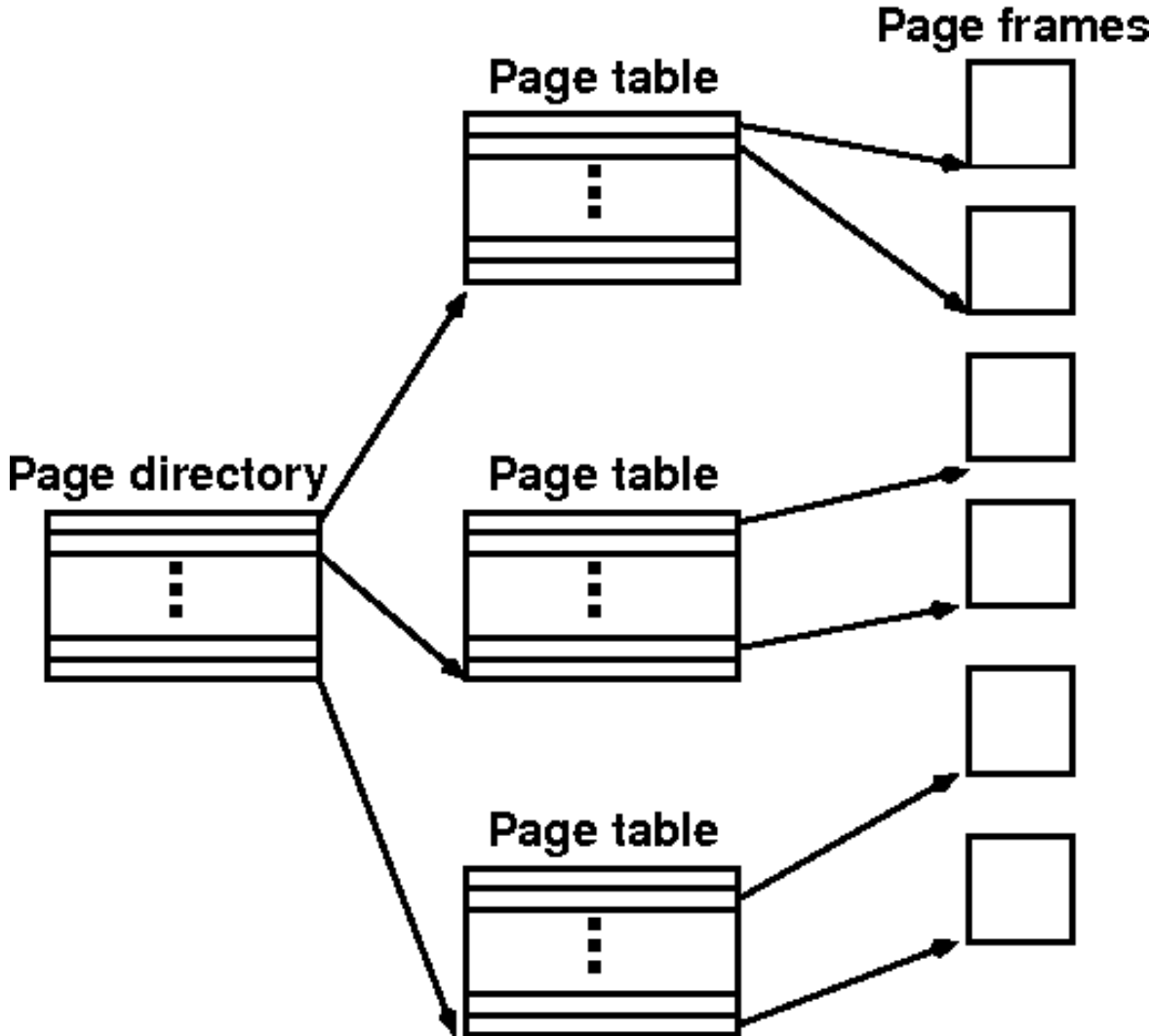
Each entry is 4-byte (32 bits)

Intel 32-bit Processor uses a 2-level page table

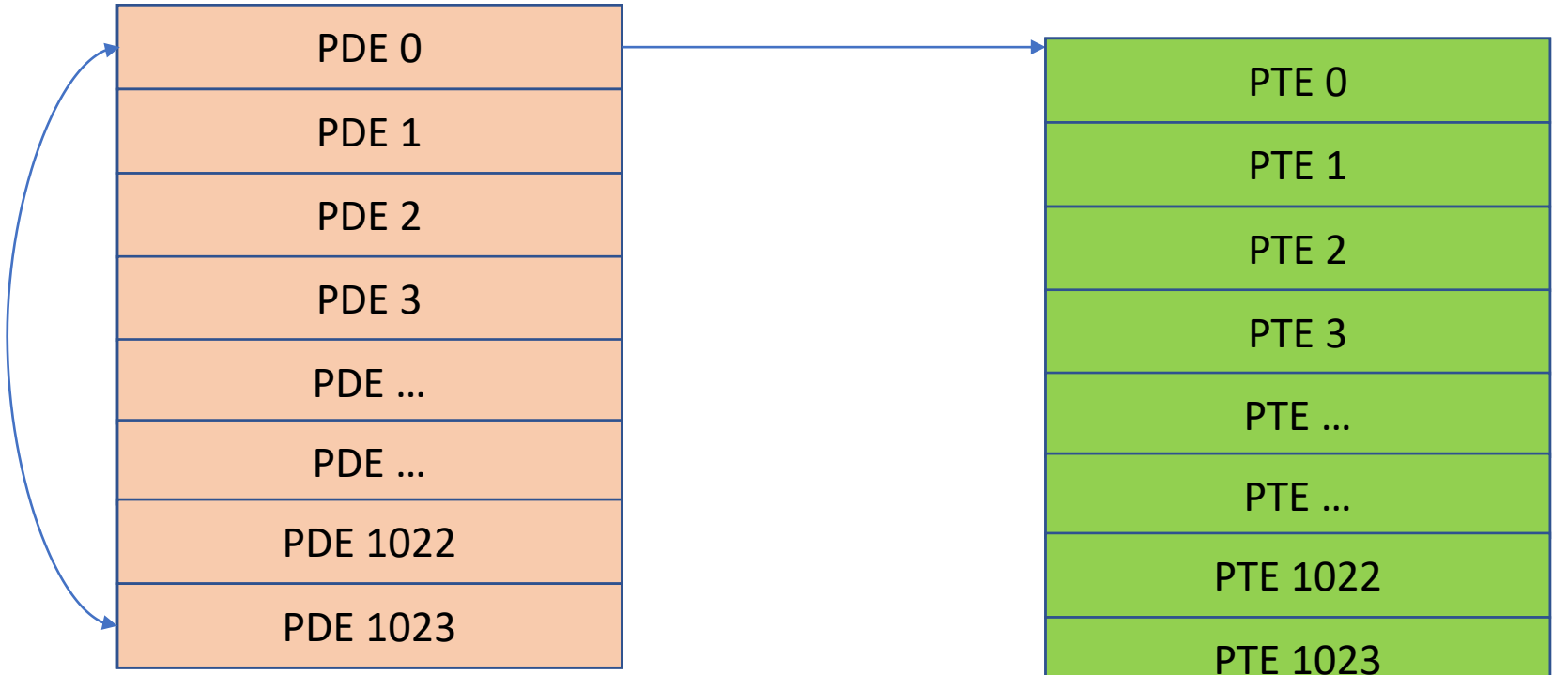
- Virtual address

- Page directory (level 1)
- Page table (level 2)

- Physical page!



What will be the size of full PD/PT?



One page, 4KB

Each entry is 4-byte (32 bits)

4KB per each Page Table…

4KB for page directory

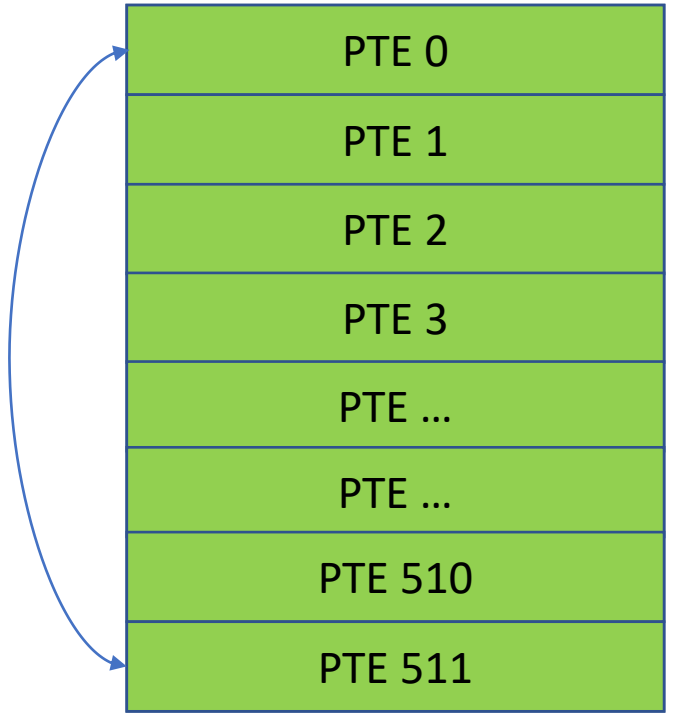
Size of Page Table

- 4 KB for a Page Directory (only one per each process)
- 1024 Page Tables available
  - 4 KB for a Page Table
- 4 KB (PD) + 1024 * 4 KB (PT)
  - 4 KB + 4MB
  - ~ 4MB, 4,198,400 bytes…

*What about amd64 (x86_64)?

- Pointer/address size in 64-bit computer: 8 bytes
- Page size in 64-bit computer: still 4KB
- How many entries are there in each PD/PT?
  - Page size / pointer size => 4KB / 8B = 512 entries

One page, 4KB

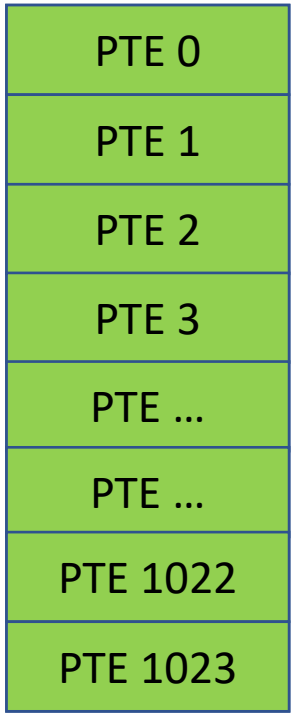


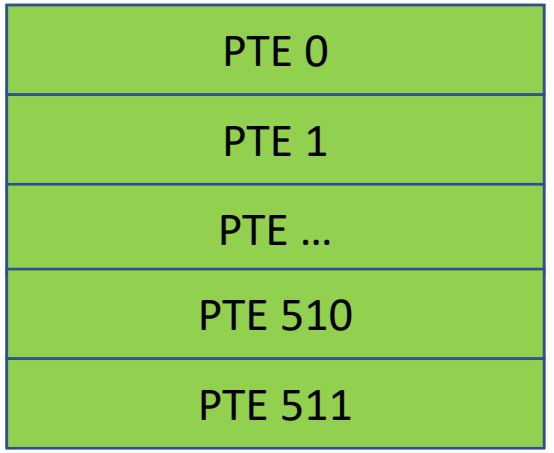
Each entry is 8-byte (64 bits)

*We do not use the entire 64 bits memory space …

- 32-bit memory space
  - 2^{32} == 4GB
  - Lower 12 bits == offset
  - Only translate top 20 bits
    - 20 bits required to index all pages
  - 10 bits PDE, 10 bits PTE → 2 level page table

- 64-bit memory space
  - 2^{64} == 16 EB == 16384 PB == 16777216 TB
  - Lower 12 bits == offset
  - Need to translate 64 – 12 == 52 bits
  - Each table holds 512 entries, indexed by 9 bits
  - Need 52/9 → 6 level page table …





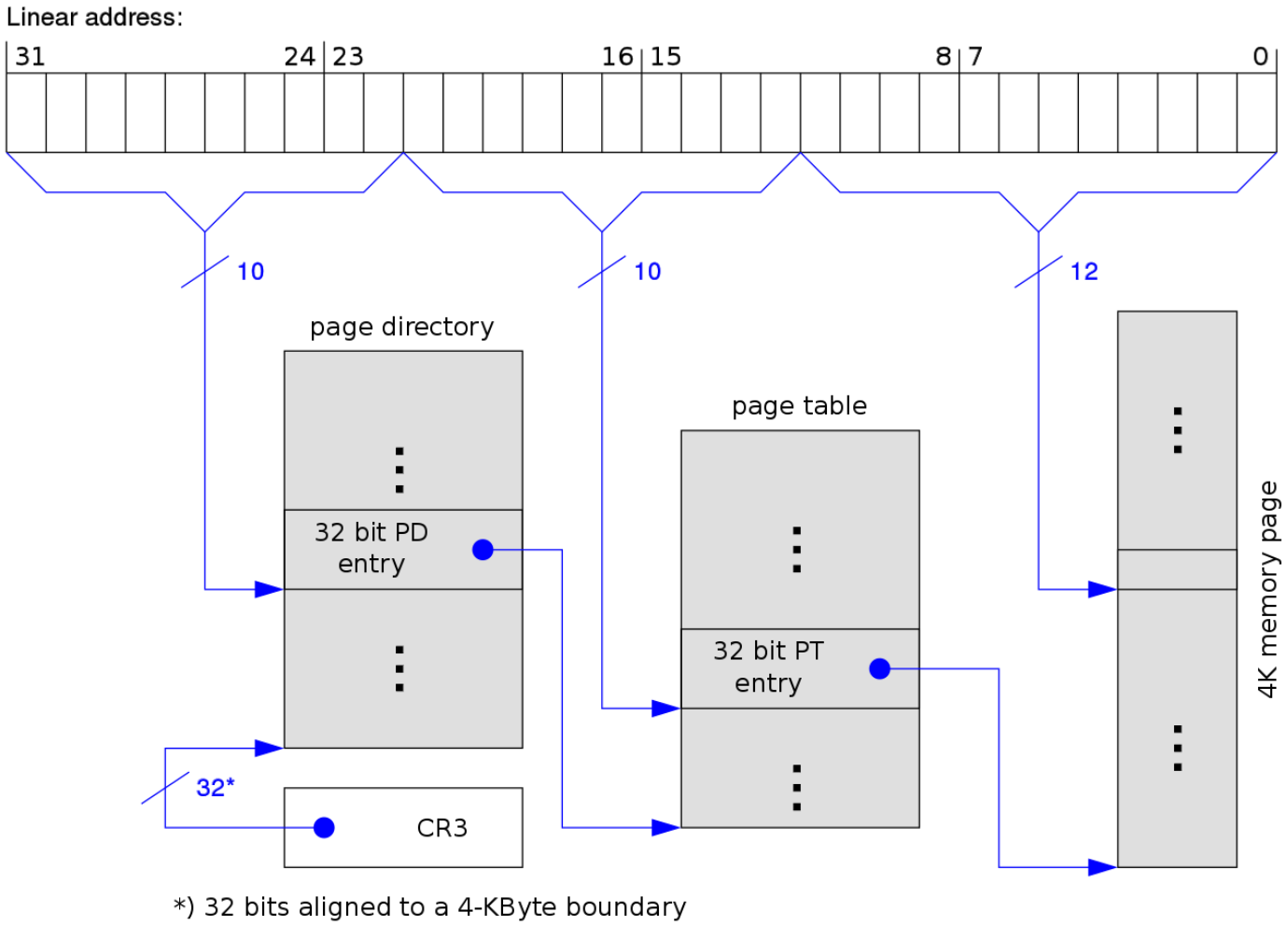
*Use 48-bit address space instead

- Initial amd64 processors only use 48-bit virtual address space
  - 2^{48} = 256 TB

- Each level: 512 entries → process 9 bits
- Lower 12 bits == offset
  - 48 – 12 = 36
  - 36 / 9 = 4

- Can be done using 4-level page table

Two-level Page Table (32 bit)



*) 32 bits aligned to a 4-KByte boundary

Four-level Page Table (64 bit)

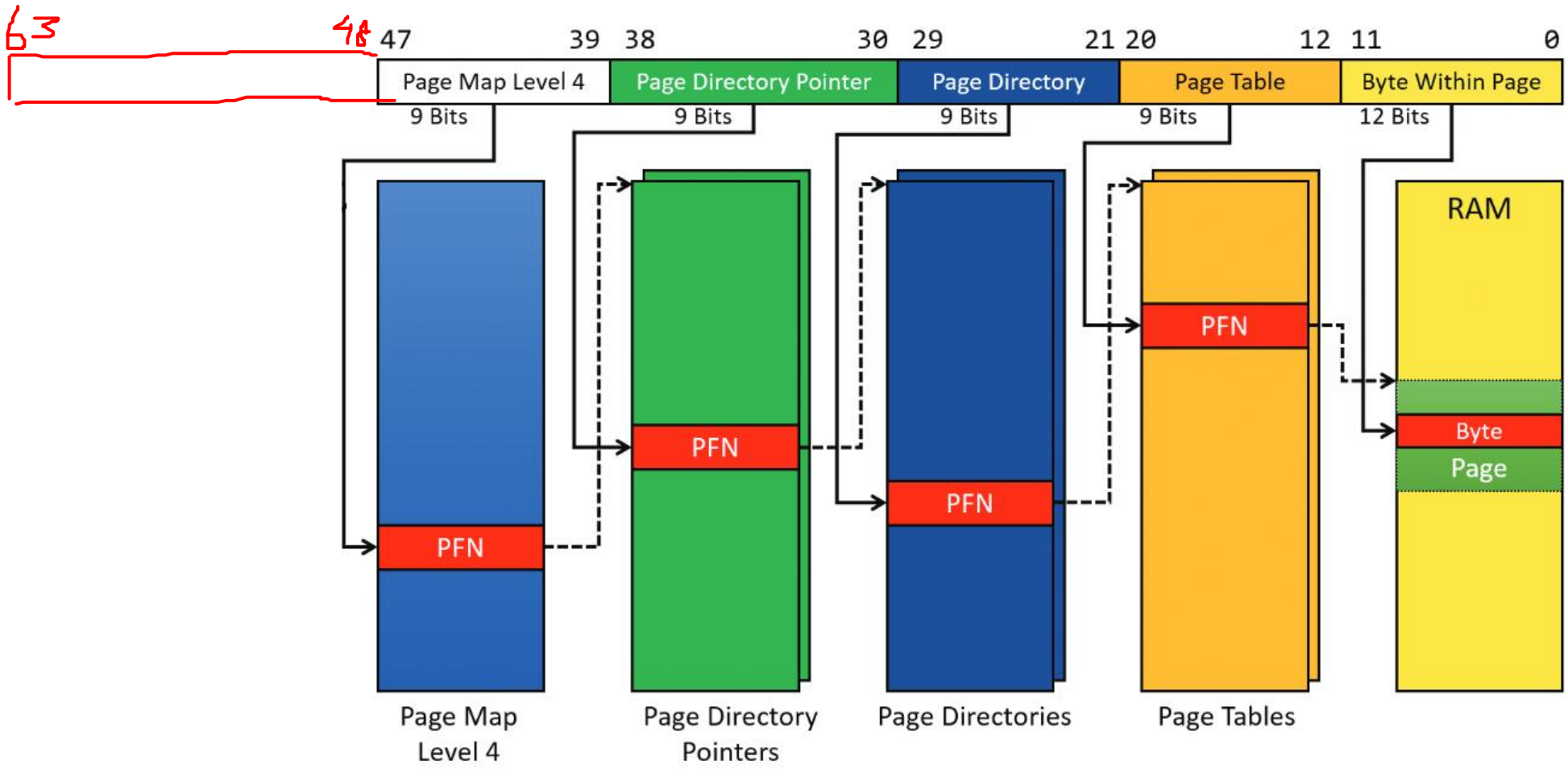


FIGURE 5-20 x64 address translation.

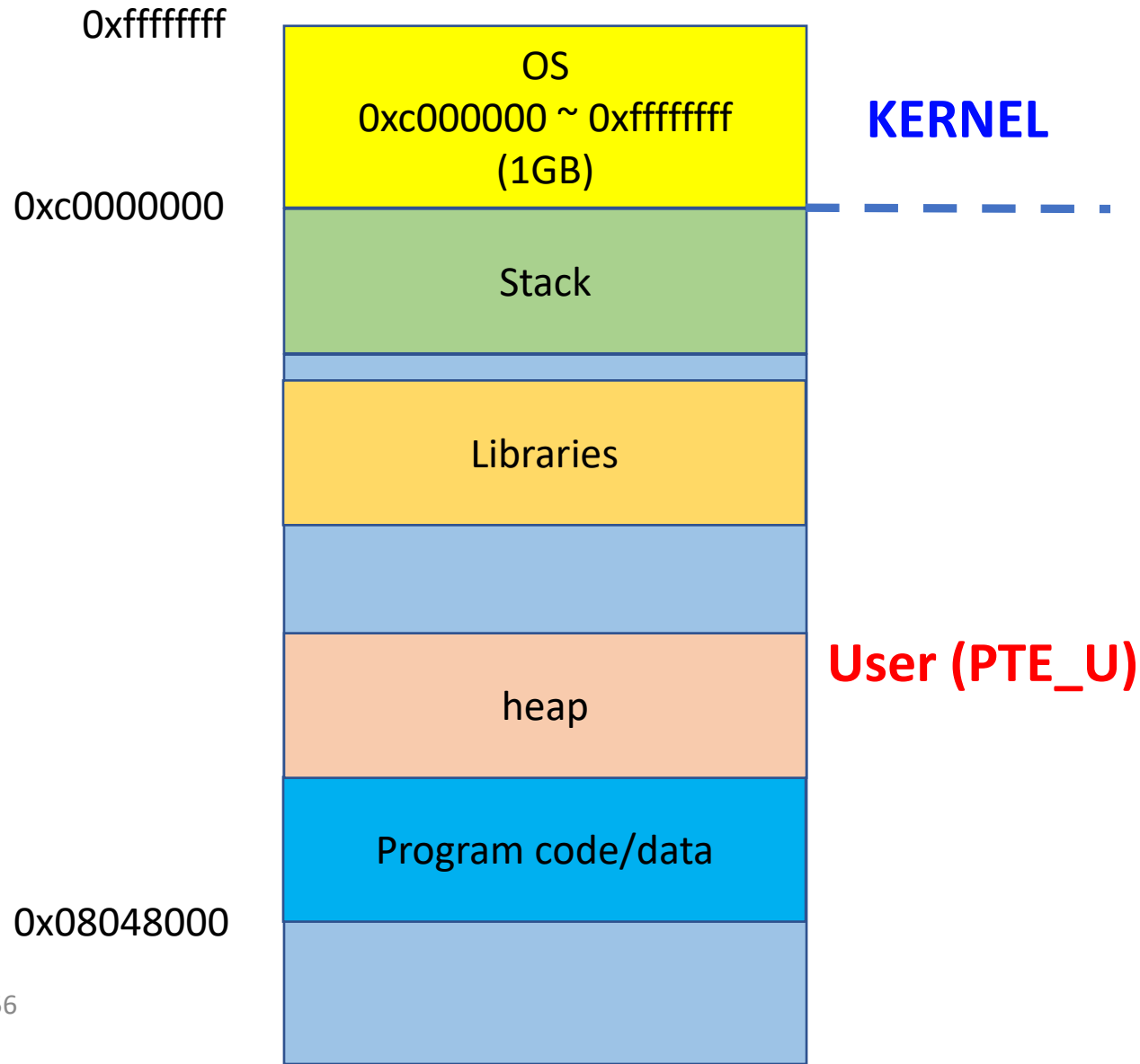
*What's more?

- 256 TB might not be enough…
  - E.g., analyzing online social network users in Facebook
    - More than 1 billion users, more than 1 trillion edges
    - 1 byte per edge = 1 TB

- 4 level == 48 bit
- Can be extended to 5 levels → 48 + 9 = 57 bits
- 6 levels? Maybe, but 57 + 9 = 68 bits > 64 bits

Virtual Memory Layout

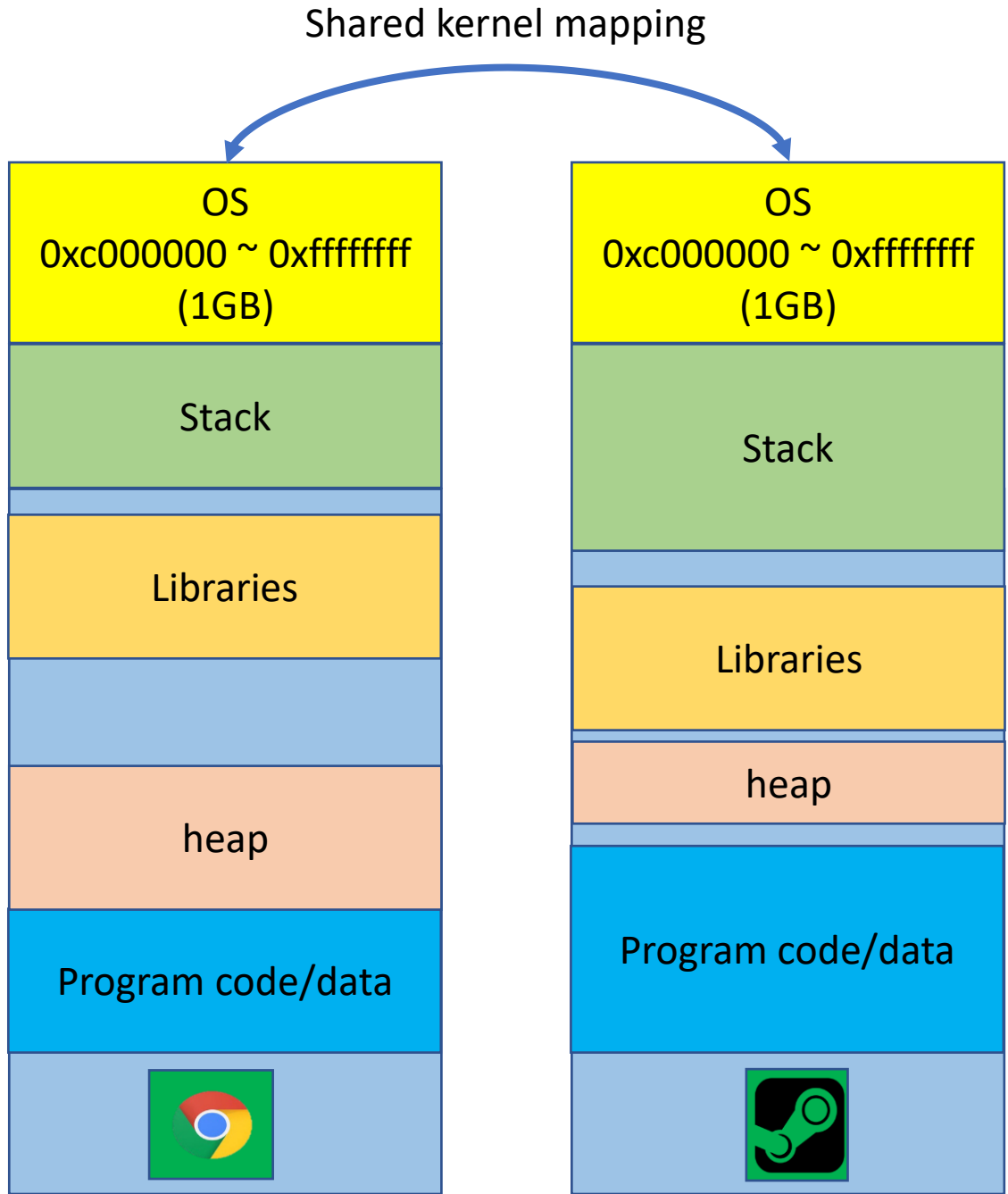
- OS allocates a separate virtual memory space for each process
- Transparency
  - Do not have to worry about a system's memory usage status
- Isolation
  - Others can't access my virtual memory space



Virtual Memory Layout

- Each process will have almost the same mapping for the kernel but having a different mapping for user space

- Why?
  - Kernel is shared among processes
  - Each process could run different apps



Example: cat binary

```
os2 ~ 150% cat /proc/self/maps
00400000-0040b000 r-xp 00000000 fd:02 145876                             /usr/bin/cat
0060b000-0060c000 r--p 0000b000 fd:02 145876                             /usr/bin/cat
0060c000-0060d000 rw-p 0000c000 fd:02 145876                             /usr/bin/cat
01e47000-01e68000 rw-p 00000000 00:00 0                                  [heap]
7f6f08062000-7f6f0e5a4000 r--p 00000000 fd:02 51142142                   /usr/lib/locale/locale-archive
7f6f0e5a4000-7f6f0e768000 r-xp 00000000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6f0e768000-7f6f0e967000 ---p 001c4000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6f0e967000-7f6f0e96b000 r--p 001c3000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6f0e96b000-7f6f0e96d000 rw-p 001c7000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6f0e96d000-7f6f0e972000 rw-p 00000000 00:00 0
7f6f0e972000-7f6f0e994000 r-xp 00000000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6f0eb52000-7f6f0eb55000 rw-p 00000000 00:00 0
7f6f0eb92000-7f6f0eb93000 rw-p 00000000 00:00 0
7f6f0eb93000-7f6f0eb94000 r--p 00021000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6f0eb94000-7f6f0eb95000 rw-p 00022000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6f0eb95000-7f6f0eb96000 rw-p 00000000 00:00 0
7ffde563d000-7ffde565f000 rw-p 00000000 00:00 0                          [stack]
7ffde5717000-7ffde5719000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Example: more binary

```
os2 ~/cs444/s21 337% more /proc/self/maps
00400000-00409000 r-xp 00000000 fd:02 872977                             /usr/bin/more
00608000-00609000 r--p 00008000 fd:02 872977                             /usr/bin/more
00609000-0060a000 rw-p 00009000 fd:02 872977                             /usr/bin/more
00f80000-00fa1000 rw-p 00000000 00:00 0                                  [heap]
7f6d5f4e6000-7f6d65a28000 r--p 00000000 fd:02 51142142                   /usr/lib/locale/locale-archive
7f6d65a28000-7f6d65bec000 r-xp 00000000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6d65bec000-7f6d65deb000 ---p 001c4000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6d65deb000-7f6d65def000 r--p 001c3000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6d65def000-7f6d65df1000 rw-p 001c7000 fd:02 16806708                   /usr/lib64/libc-2.17.so
7f6d65df1000-7f6d65df6000 rw-p 00000000 00:00 0
7f6d65df6000-7f6d65e1b000 r-xp 00000000 fd:02 16806782                   /usr/lib64/libtinfo.so.5.9
7f6d65e1b000-7f6d6601b000 ---p 00025000 fd:02 16806782                   /usr/lib64/libtinfo.so.5.9
7f6d6601b000-7f6d6601f000 r--p 00025000 fd:02 16806782                   /usr/lib64/libtinfo.so.5.9
7f6d6601f000-7f6d66020000 rw-p 00029000 fd:02 16806782                   /usr/lib64/libtinfo.so.5.9
7f6d66020000-7f6d66042000 r-xp 00000000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6d661ff000-7f6d66203000 rw-p 00000000 00:00 0
7f6d66237000-7f6d66238000 rw-p 00000000 00:00 0
7f6d66238000-7f6d6623f000 r--s 00000000 fd:02 1893394                    /usr/lib64/gconv/gconv-modules.cache
7f6d6623f000-7f6d66241000 rw-p 00000000 00:00 0
7f6d66241000-7f6d66242000 r--p 00021000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6d66242000-7f6d66243000 rw-p 00022000 fd:02 17757900                   /usr/lib64/ld-2.17.so
7f6d66243000-7f6d66244000 rw-p 00000000 00:00 0
7ffc06a30000-7ffc06a52000 rw-p 00000000 00:00 0                          [stack]
7ffc06b5d000-7ffc06b5f000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Summary

- Page Directory Entry / Page Table Entry
  - Permission bits (P, W, U)
  - Permission: {bits in PDE} ⋂ {bits in PTE}
- Virtual memory is N-to-1 mapping
  - Sharing physical page
  - Allowing conflicting permission assignment
    - Kernel RW and User R
- Virtual Memory Layout
  - Shares kernel space (typically at the top of virtual memory space)
  - Can use user space arbitrarily (full transparency and isolation)