# CS330: Operating Systems

Virtual memory: Multilevel paging and TLB

Recap: Paging

- The idea of paging

    - Partition the address space into fixed sized blocks (call it pages)

    - Physical memory partitioned in a similar way (call it page frames)

    - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA

- With increased address space size, single level page table entry is not feasible, because

    - Increasing page size increases internal fragmentation

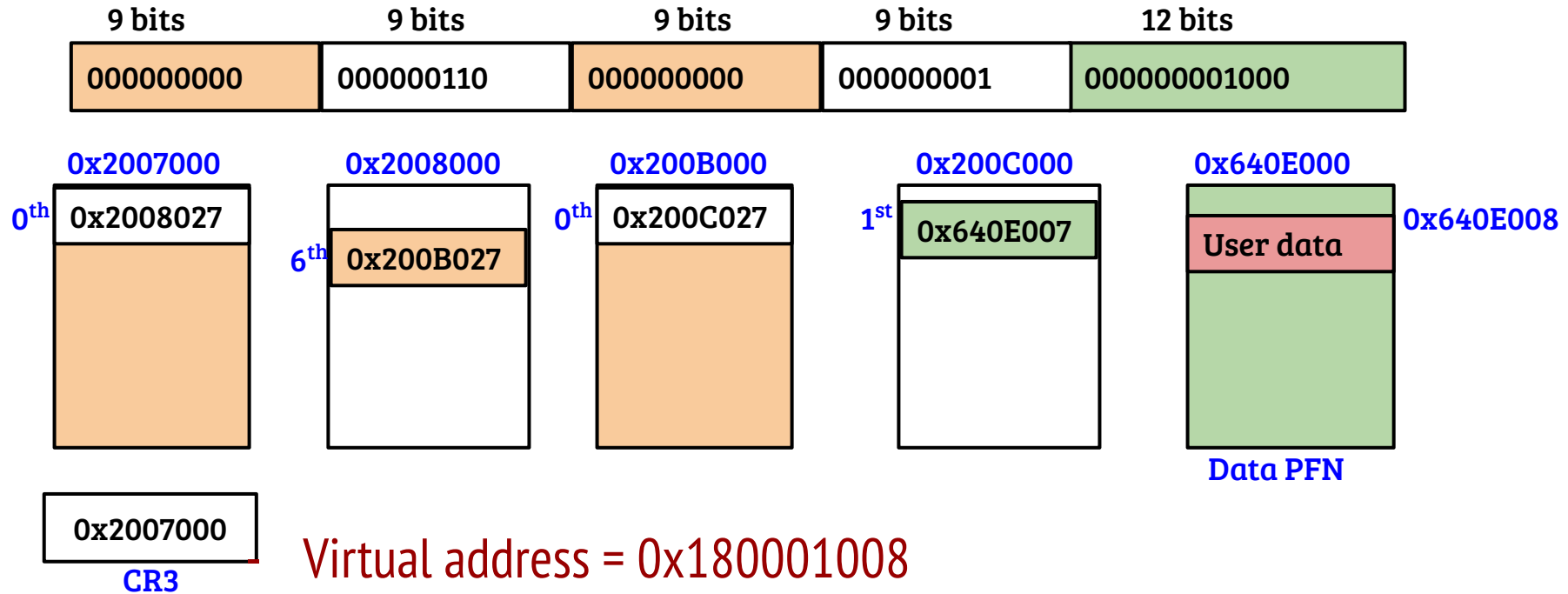   - Small pages may not be suitable to hold all mapping entries

Today's agenda: Multi-level pages tables and implications

4-level page tables: 48-bit VA (Intel x86_64)
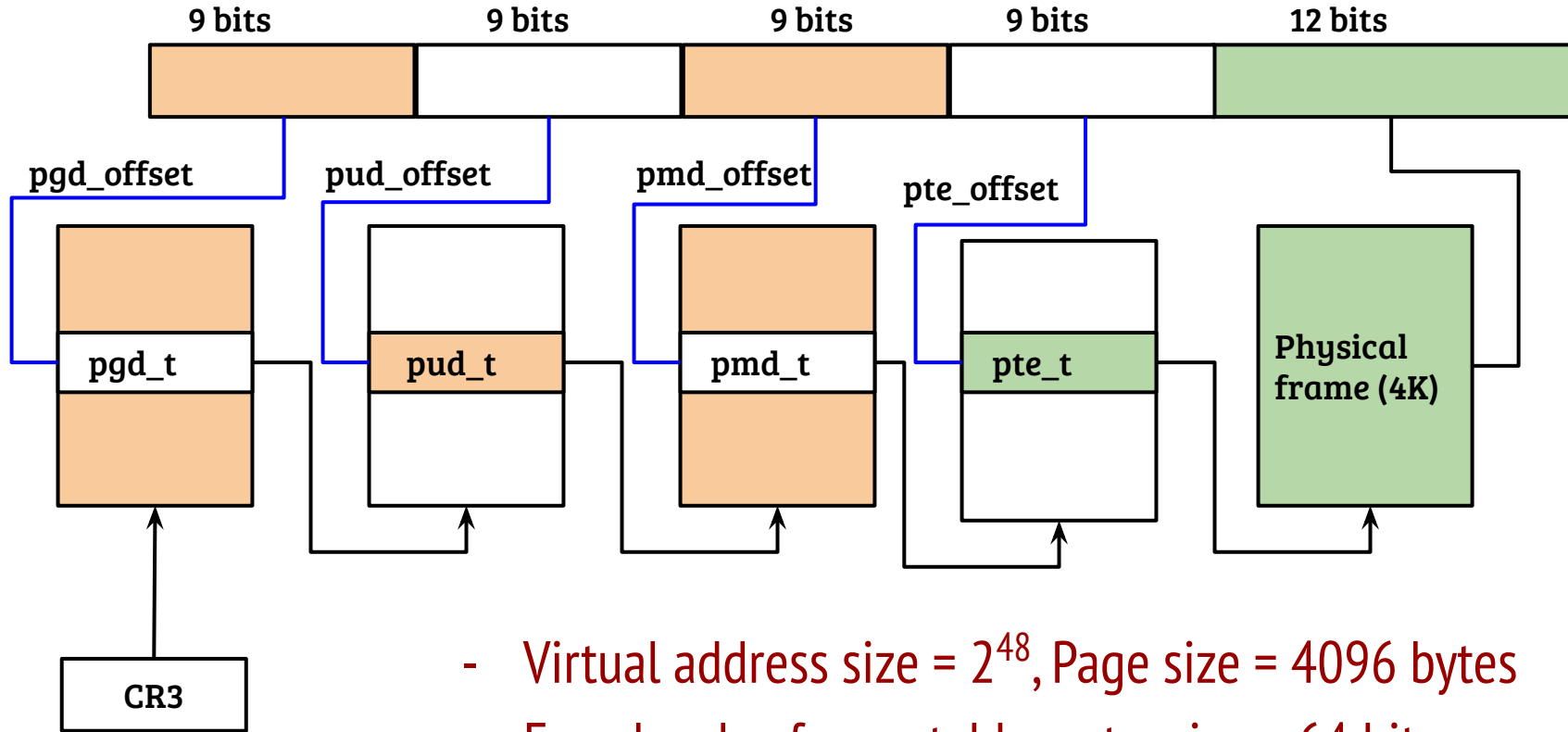
- Virtual address size = 2^{48}, Page size = 4096 bytes
- Four-levels of page table, entry size = 64 bits

4-level page tables: example translation

9 bits	9 bits	9 bits	9 bits	12 bits
000000000	000000110	000000000	000000001	000000001000

0x2007000

0th | 0x2008027

0x2008000

6th | 0x200B027

0x200B000

0th | 0x200C027

0x200C000

1st | 0x640E007

0x640E000

User data   0x640E008

Data PFN

0x2007000

CR3

- Virtual address = 0x180001008
- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
0x20100:   mov $0, %rax;
0x20102:   mov %rax, (%rbp);     // sum=0
0x20104:   mov $0, %rcx;          // ctr=0
0x20106:   cmp $10, %rcx;         // ctr < 10
0x20109:   jge  0x2011f;          // jump if >=
0x2010f:   add %rcx, %rax;
0x20111:   mov %rax, (%rbp);      // sum += ctr
0x20113:   inc %rcx               // ++ctr
0x20115:   jmp 0x20106            // loop
0x2011f:   .............
```

sum = 0;
for(ctr=0; ctr<10; ++ctr)
    sum += ctr;

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

- Instruction execution: Loop = 10 * 6, Others = 2 + 3

  - Memory accesses during translation = 65 * 4 = 260

- Data/stack access: Initialization = 1, Loop = 10

  - Memory accesses during translation = 11 * 4 = 44

- A lot of memory accesses (> 300) for address translation

- How many distinct pages are translated?

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
    PageAddress P = V >> 12;
    TLBEntry entry = lookup(P);
    if (entry.valid) return entry.pte;
    entry = PageTableWalk(V);
    MakeEntry(entry);
    return entry.pte;
}
```

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Paging: translation efficiency

0x20100: mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
    - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
    - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.
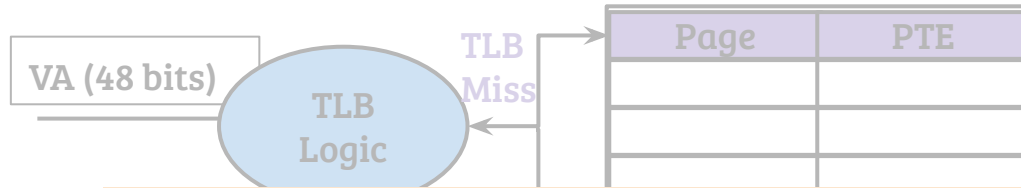
required (for translation) during the execution of the above code?

Address translation (TLB + PTW)
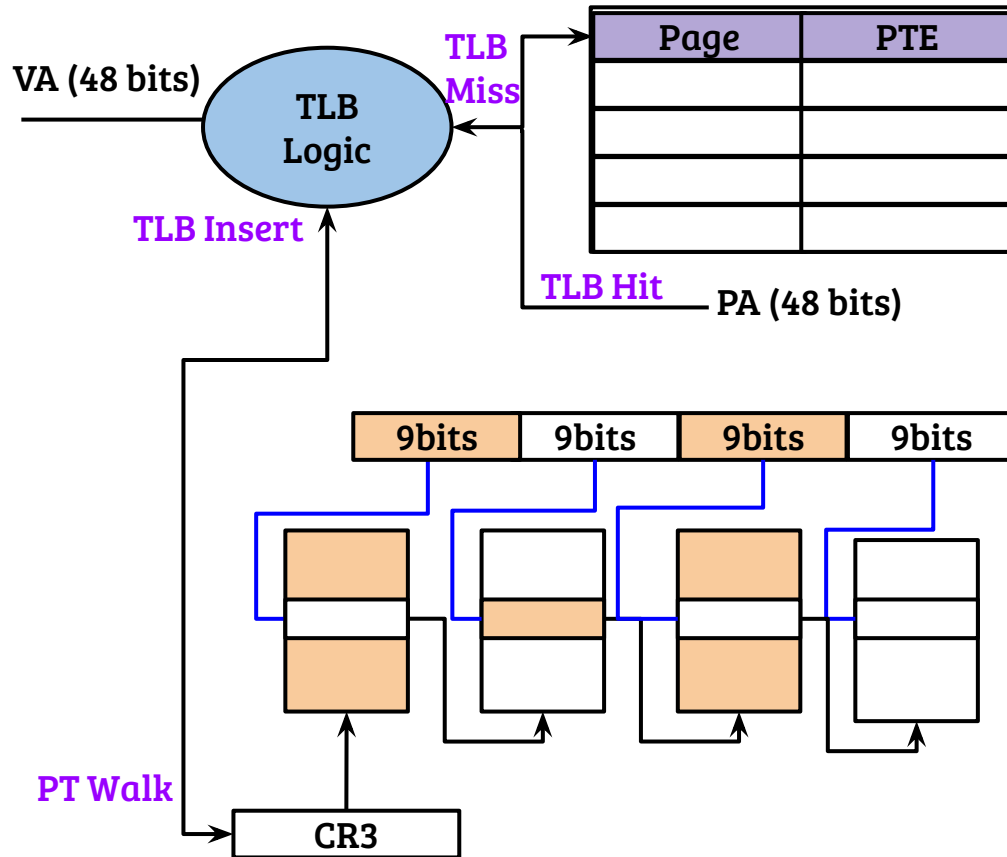
- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)

VA (48 bits)

TLB Logic

TLB Miss

Page	PTE

- TLB in the path of address

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?
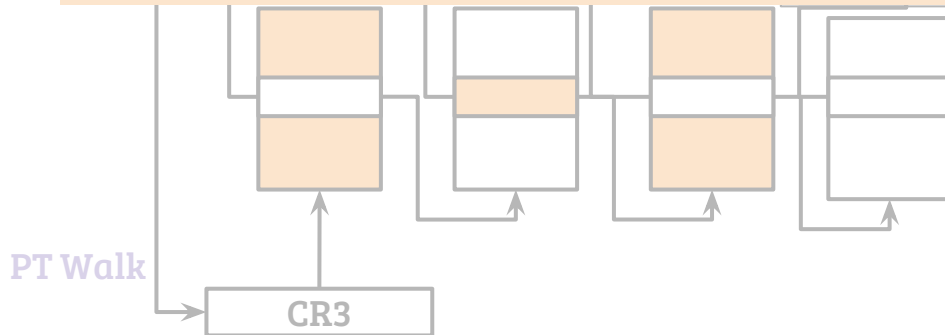
into the TLB directly, it can flush entries

PT Walk

CR3

TLB: Sharing across applications

| Process (A) | Process (B) |

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution

TLB: Sharing across applications

Process (A)	Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
    - A) Do nothing
    - B) Flush the whole TLB
    - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications

Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

Process (A)   Process (B)

- Assume that, process A is currently executing. What happens when process B is scheduled?
    - A) Do nothing
    - B) Flush the whole TLB
    - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

ASID	Page	PTE
A	0x100	0x200007
A	0x101	0x205007
B	0x100	0x301007
B	0x101	0x302007

TLB

Process (A)   Process (B)

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
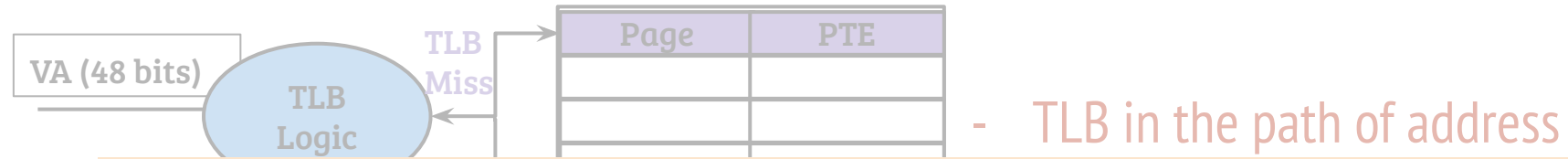 - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)

VA (48 bits)

TLB Logic

TLB Miss

| Page	PTE

- TLB in the path of address

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

entries

PT Walk

CR3

Address translation (TLB + PTW)

VA (48 bits)

TLB

TLB Miss

Page	PTE
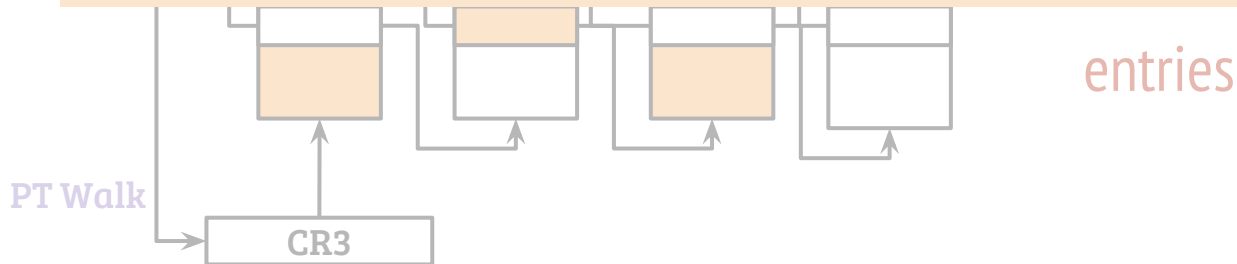
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

PT Walk

CR3
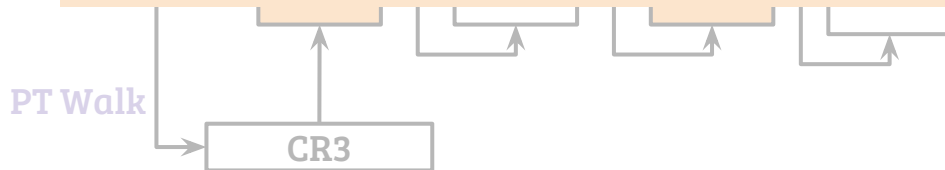
Page fault handling in X86: Hardware

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                        | access << 1
                        | cpl << 2;
        Raise pageFault;
} // Simplified
```

Page fault handling in X86: Hardware

Error code

Other and unused			I	R	U	W	P

```
If( !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```

P — **Present bit, 1 ⇒ fault is due to protection**

W — **Write bit, 1 ⇒ Access is write**

U — **Privilege bit, 1 ⇒ Access is from user mode**

R — **Reserved bit, 1 ⇒ Reserved bit violation**

I — **Fetch bit, 1 ⇒ Access is Instruction Fetch**

- Error code is pushed into the kernel stack by the hardware

Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Address translation (TLB + PTW)

VA (4

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

PT