# Operating Systems

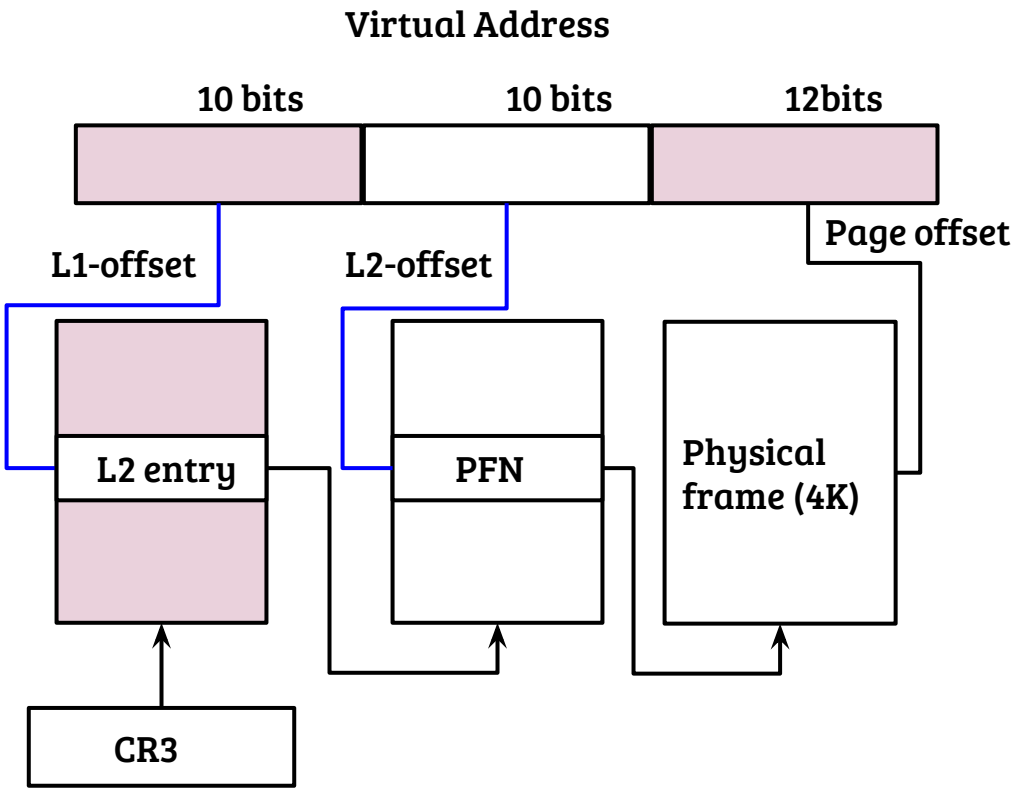
## Paging: design issues and optimizations

Debadatta Mishra, CSE, IITK

# Paging

- ➔ Every process is provided an illusion of a huge memory, called the virtual memory
- ➔ Virtual address width determines the size of virtual address
  - ◆ For example, 32-bit wide virtual address → 4GB virtual address, 0x0 to 0xFFFFFFFF
- ➔ Virtual address is partitioned into small chunks (mostly 4KB), called pages
- ➔ Compiler can place the program structures (code, data etc.) at any address in the virtual address range (ideally).
- ➔ Physical address is partitioned into chunks of same size, called page frames
- ➔ When applications use virtual address to access memory,
  - ◆ Virtual address is translated to physical address
  - ◆ Translation is per-process and performed by the hardware
  - ◆ Translation may require multiple intermediate levels of translation (will see shortly)
  - ◆ A CPU register (CR3 in X86) points to a memory location containing the first level of translation

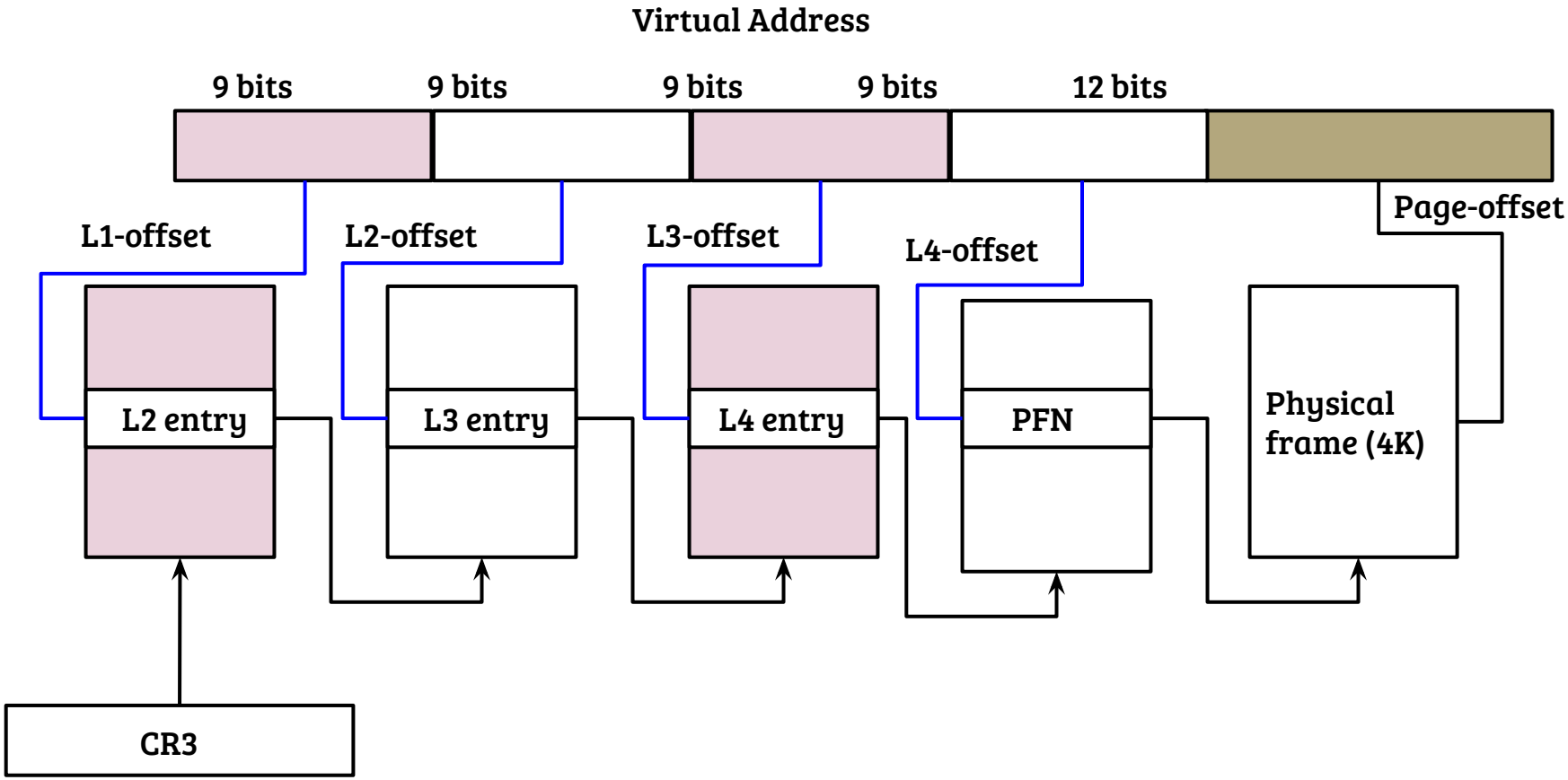
# Paging example (32-bit virtual address)



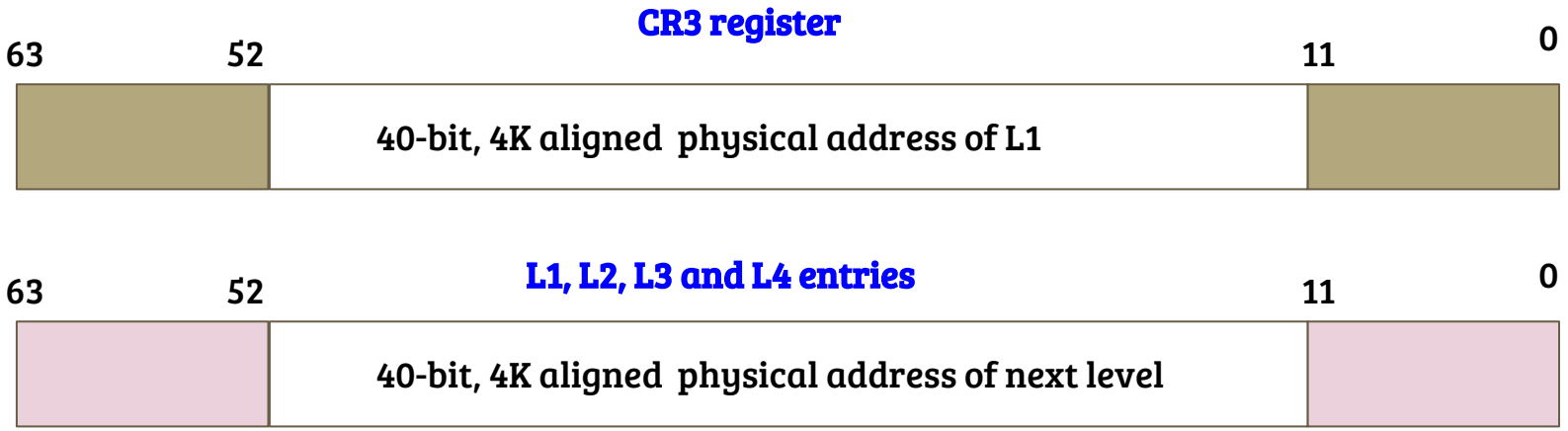

➔ Example:

- Page size = 4KB
- Entry size = 4 Bytes (access flags + next level address)
- How virtual address 0x3F50A075 will be translated?
- What is the max. physical address supported if 16 bits in page table entry are reserved for access flags?
- #of memory accesses to perform translation?

# X86_64: 4-level page tables (48-bit virtual address)



# X86_64 page table entries




Some important flags

0 (present/absent) 1 (read/write) {2} (user/kernel), 5(accessed) 63 (execute permission)

*Source: Intel manual Vol: 3A 4.5

# Paging: design parameters

- ➔ Design issue: Translation overheads
  - Virtual address size → number of translation levels → translation overhead
  - For example, With page size = 4KB,  4-levels for 48-bit VA instead of 2-levels in 32-bit
  - Large page size → reduced number of translation levels, memory fragmentation issues

# Paging: design parameters

- ➔ Design issue: Translation overheads
  - Virtual address size → number of translation levels → translation overhead
  - For example, With page size = 4KB,  4-levels for 48-bit VA instead of 2-levels in 32-bit
  - Large page size → reduced number of translation levels, memory fragmentation issues
- ➔ Design issue: Memory required for page tables
  - {Small page size and more page table levels} → memory efficient
  - {Large page size and sparse virtual address layout} → memory inefficient
  - Memory required for page tables depends on application
  - How can OS help?

# Paging: design parameters

- ➔ Design issue: Translation overheads
  - ◆ Virtual address size → number of translation levels → translation overhead
  - ◆ For example, With page size = 4KB,  4-levels for 48-bit VA instead of 2-levels in 32-bit
  - ◆ Large page size → reduced number of translation levels, memory fragmentation issues
- ➔ Design issue: Memory required for page tables
  - ◆ {Small page size and more page table levels} → memory efficient
  - ◆ {Large page size and sparse virtual address layout} → memory inefficient
  - ◆ Memory required for page tables depends on application
  - ◆ How can OS help?
- ➔ Virtual memory layout
  - ◆ Best case
  - ◆ Worst case

# Paging: translation efficiency

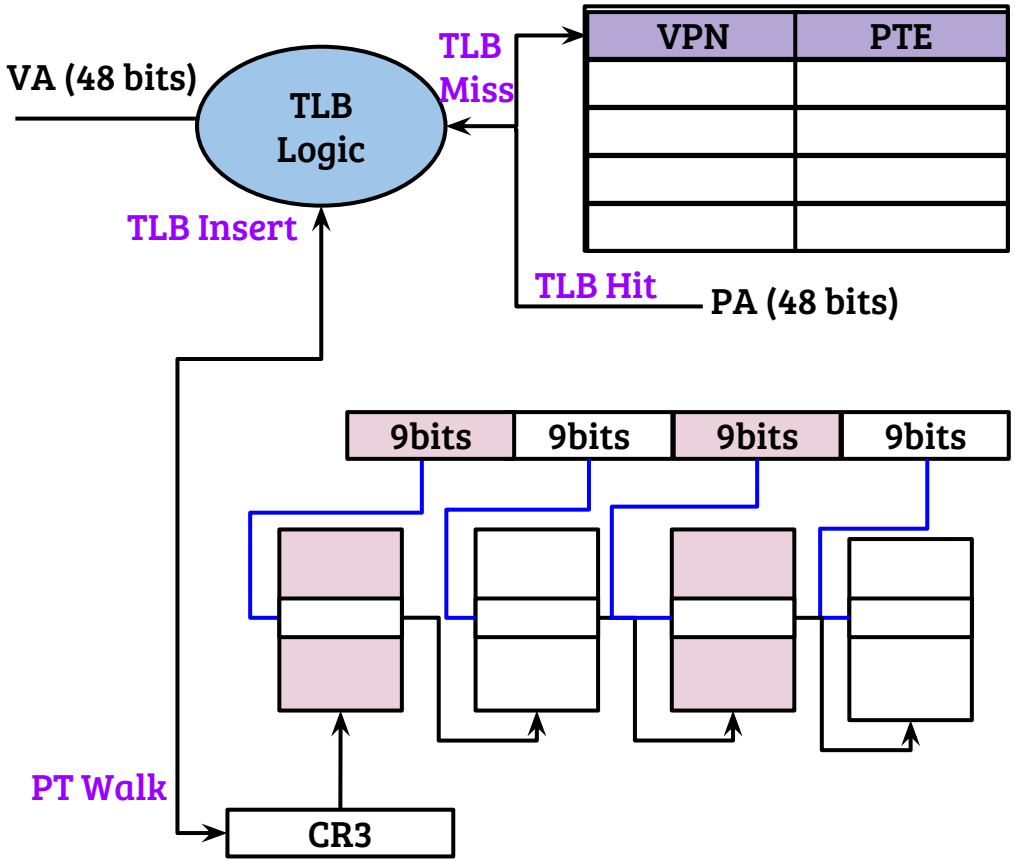
```
unsigned  long *V1, *V2, *V3;
int size = 32 * 1024;

for ( ctr=0; ctr < size; ctr++){
      V3[ctr] = V1[ctr] + V2[ctr];
}

/* RSP = 0x8000000 - 0x7FFF000,
   RIP = 0x10000 - 0x10080
   V1 = 0x4000000,
   V2 = 0x4200000,
   V3 = 0x440000 */
```

- ➔ Consider 4-levels of translation, 48-bit virtual address
- ➔ Memory accesses required for translation, considering
  - ◆ Code execution
  - ◆ Data access
  - ◆ Stack operations
- ➔ Caches (L1, L2 etc.) can help, How?
- ➔ L1, L2 Caches are not sufficient, Why?
- ➔ A specialized cache to store recent translations required

# Translation Lookaside buffer (TLB)




- ➔ TLB stores VPN to PTE mapping
- ➔ Lookup {VPN = VA >> 12}
- ➔ Hit: Physical address = PA(PTE, VA)
- ➔ Miss: PTE = PTWalk (VPN), InsertTLB(VPN, PTE) , PhysicalAddress = (PTE, VA)

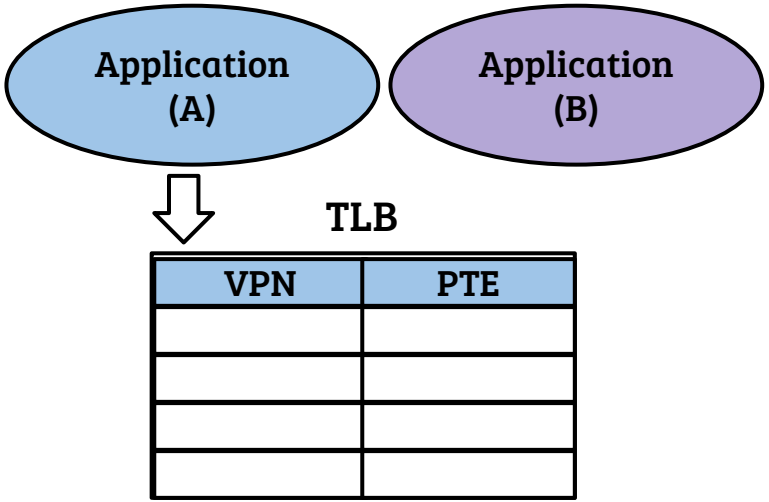
# Paging with TLB: translation efficiency

**TLB**

| VPN | PTE |
|---|---|
| 0x10 | |
| 0x7FFF | |
| 0x403E | |
| 0x403F | |
| 0x423E | |
| 0x423F | |
| 0x443E | |
| 0x443F | |

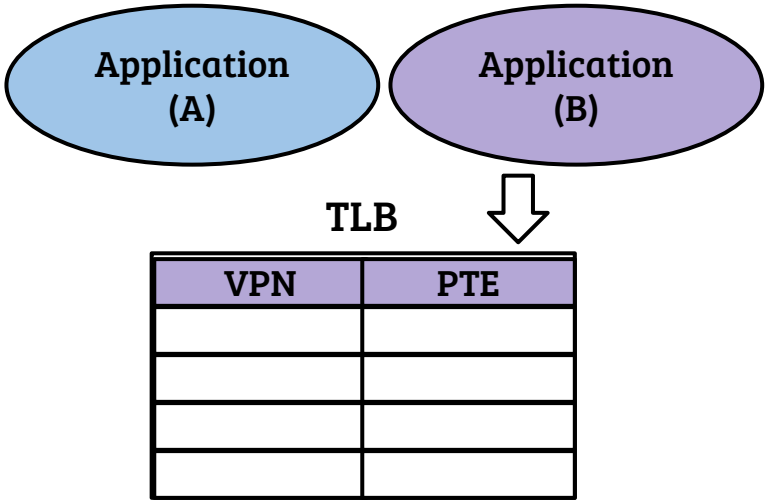
- ➔ TLB caches most recently used V to P translations
- ➔ How does TLB help addressing page table walk overheads?
- ➔ TLB + (L1, L2)
- ➔ When TLB fails to help?

# TLB: Sharing across applications



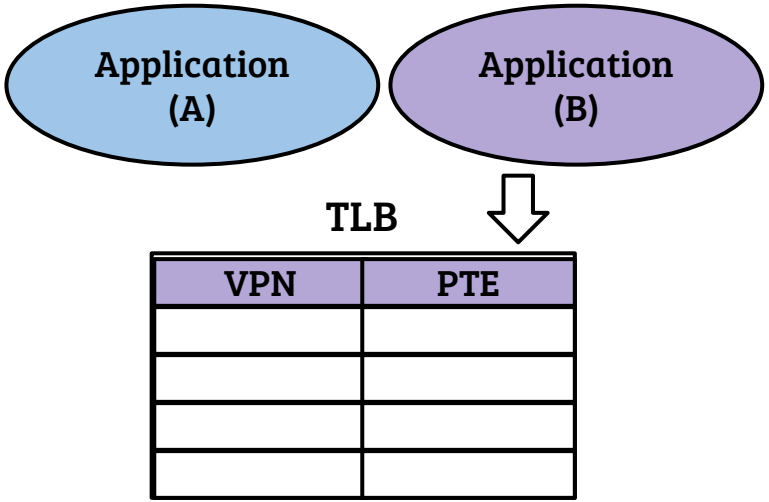

➔ What happens when OS schedules application B switching out A?

# TLB: Sharing across applications



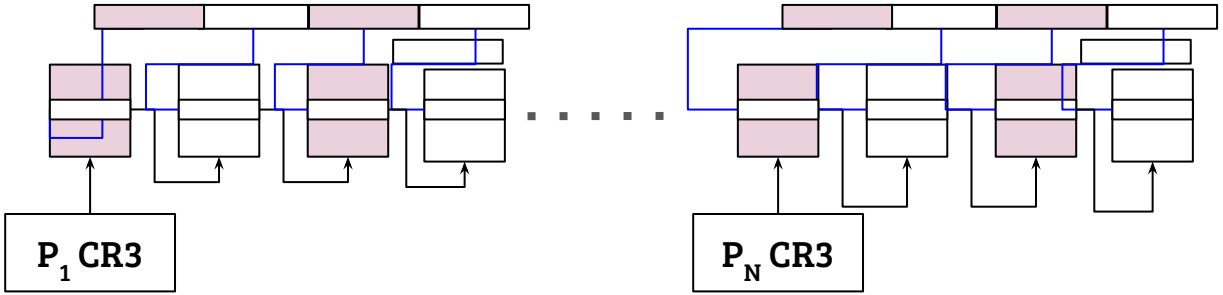

- ➔ What happens when OS schedules application B switching out A?
- ➔ Solution 1 : All entries of A are purged
  - ◆ Disadvantages?

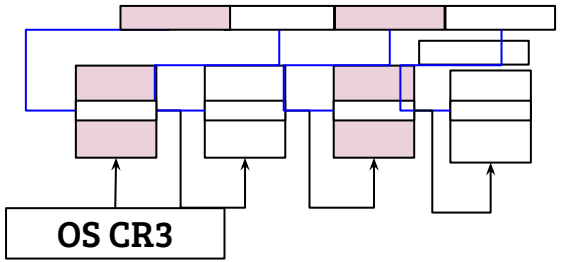
# TLB: Sharing across applications




- ➔ What happens when OS schedules application B switching out A?
- ➔ Solution 1: All entries of A are purged
  - ◆ Disadvantage?
- ➔ Solution 2: A and B share the TLB
  - ◆ How?

# OS virtual address space: Alternate 1




- ➔ What should happen
  - When switching user processes?
  - During system call and return?
  - More than one system call alive in OS?
- ➔ Can the design be more efficient?

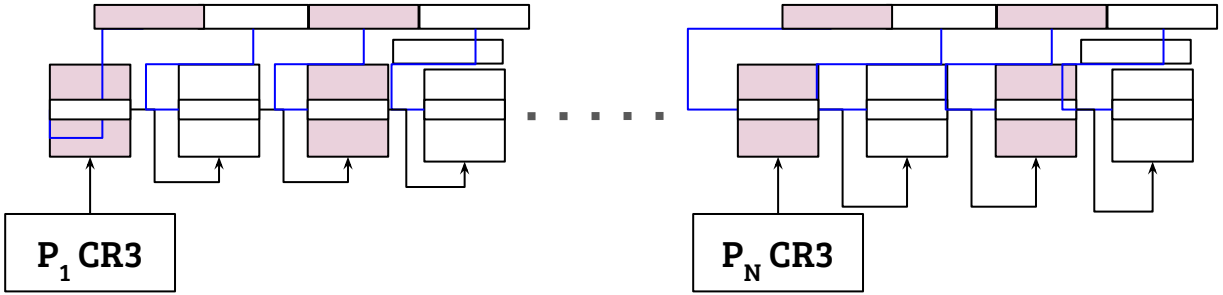
# OS virtual address

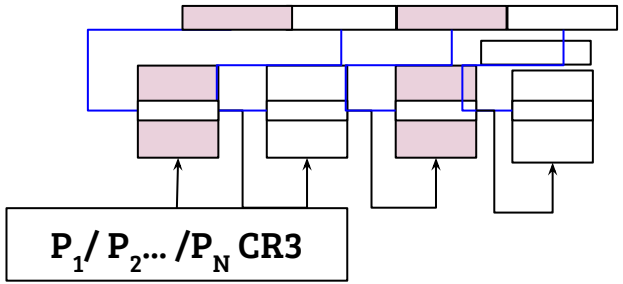
➔ Does the OS require a separate page table for isolation?

# OS virtual address

- ➔ Does the OS require a separate page table to achieve isolation?
- ➔ Can we split the available virtual address space?
  - For example, with 48-bit wide VA, the OS and applications use $2^{47}$ bytes each
  - OS should be present in all applications, Why?

# OS virtual address space: Alternate 2




➔ What should happen

- When switching user processes?
- During system call and return?
- More than one system call alive in OS?

# OS virtual address

- ➔ Does the OS require a separate page table for isolation?
- ➔ Can we split the available virtual address space?
  - For example, with 48-bit wide VA, the OS and applications use $2^{47}$ bytes each
  - OS should be present in all applications, Why?
- ➔ Advantages
  - Efficient user to OS context switch
  - No TLB flush required, even w/o ASID

# OS virtual address

- ➔ Does the OS require a separate page table for isolation?
- ➔ Can we split the available virtual address space?
  - ◆ For example, with 48-bit wide VA, the OS and applications use $2^{47}$ bytes each
  - ◆ OS should be present in all applications, Why?
- ➔ Advantages
  - ◆ Efficient user to OS context switch
  - ◆ No TLB flush required, even w/o ASID
- ➔ Disadvantages
  - ◆ Reduced VA space for applications
  - ◆ Security holes: Specter and Meltdown
  - ◆ A side effect of Hardware + Software optimizations and some silly mistakes!