



IIT KANPUR
Indian Institute of Technology Kanpur

CS623: GPU Architecture and Programming

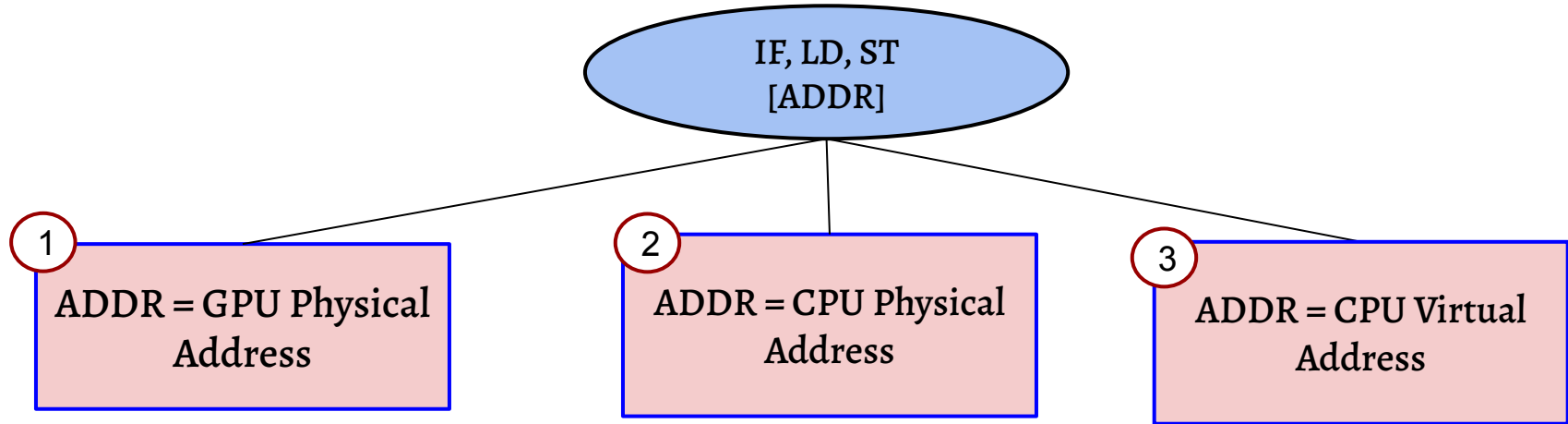
Addressing and GPU

Addressing in Discrete GPU

- Typical GPU execute logic is computation centric. Does not support memory management, data movement from I/O devices
- The GPU execution is still follows Von Neumann model
 - IF, LD and ST instructions generate memory address
- Which memory address is generated?
- Who manages the addresses?
- What is the address layout?

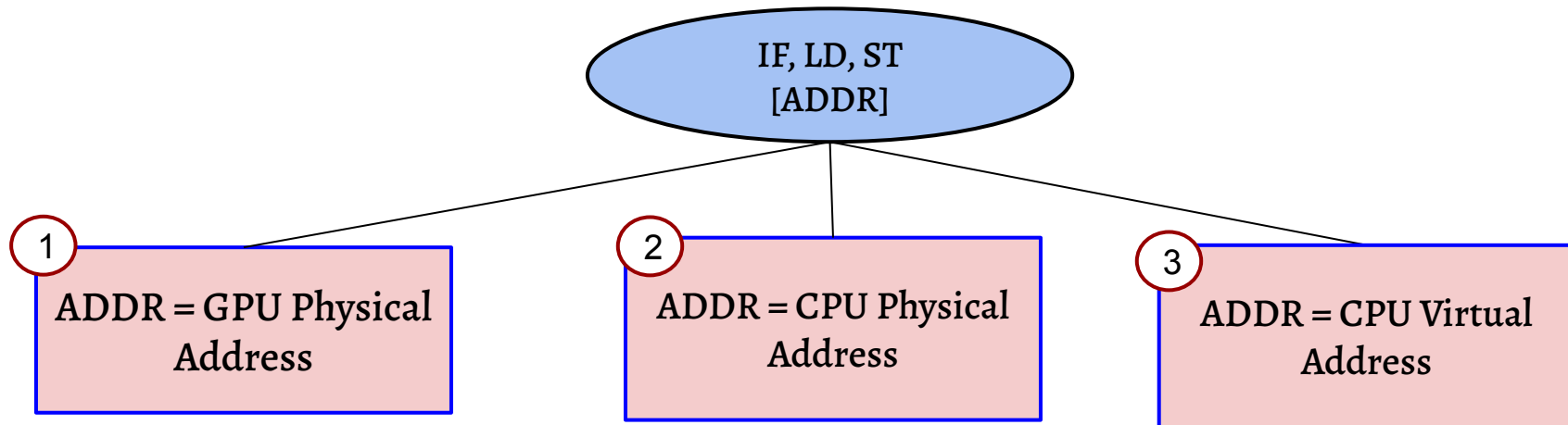
GPU Memory Access and Addressing

GPU generates memory addresses for Instructions and data Access



GPU Memory Access and Addressing

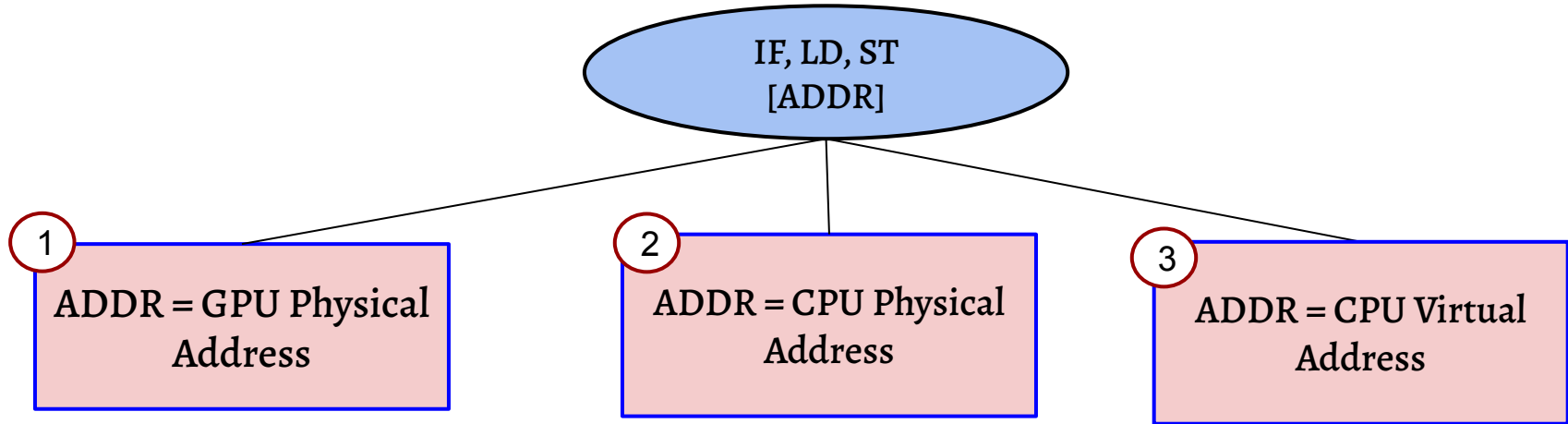
GPU generates memory addresses for Instructions and data Access



The runtime and
device interface
setup the access

GPU Memory Access and Addressing

GPU generates memory addresses for Instructions and data Access

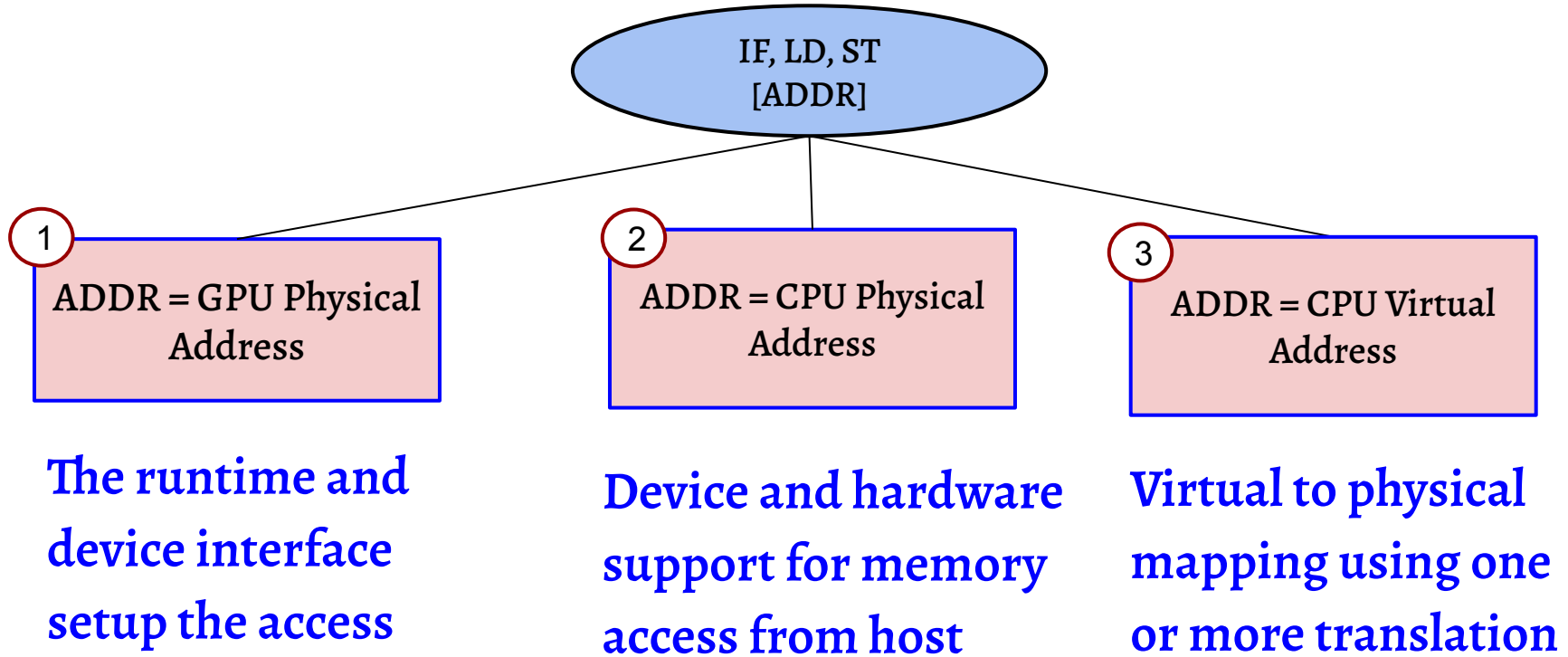


The runtime and
device interface
setup the access

Device and hardware
support for memory
access from host

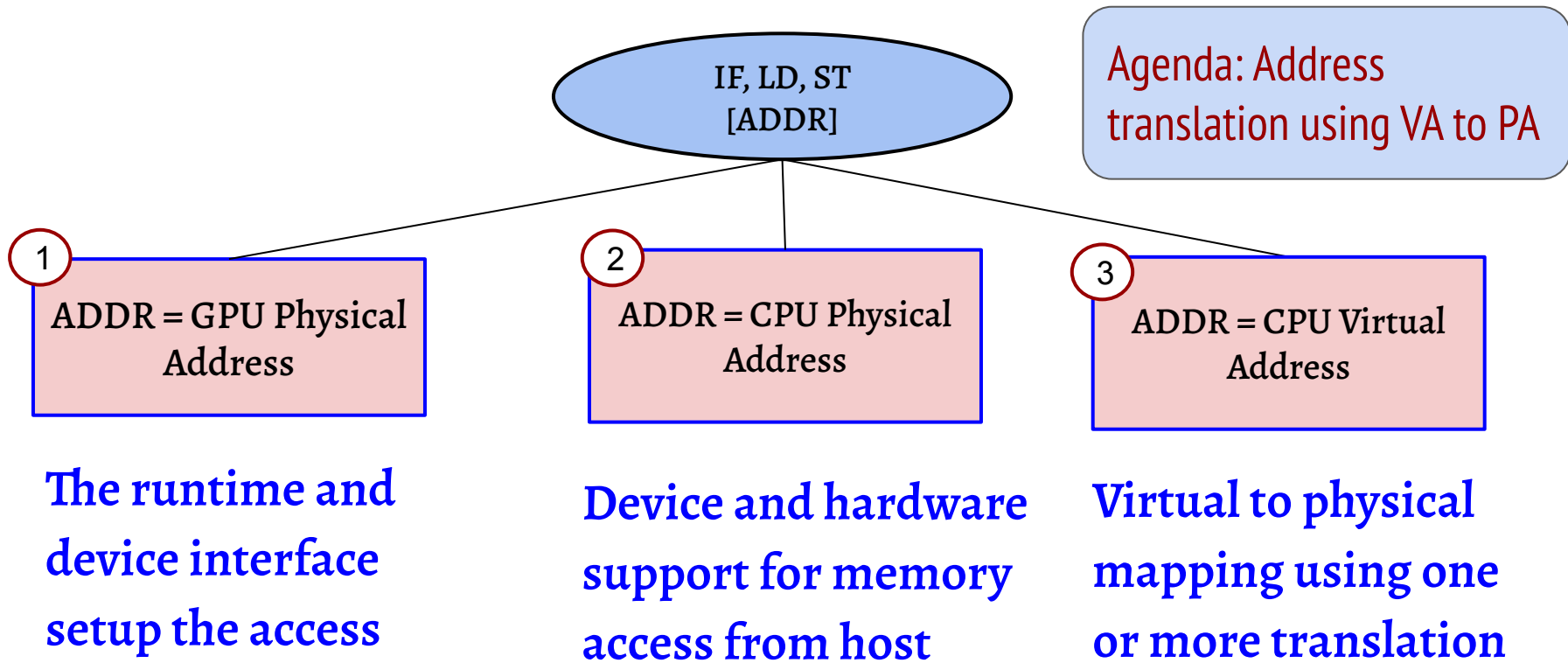
GPU Memory Access and Addressing

GPU generates memory addresses for Instructions and data Access

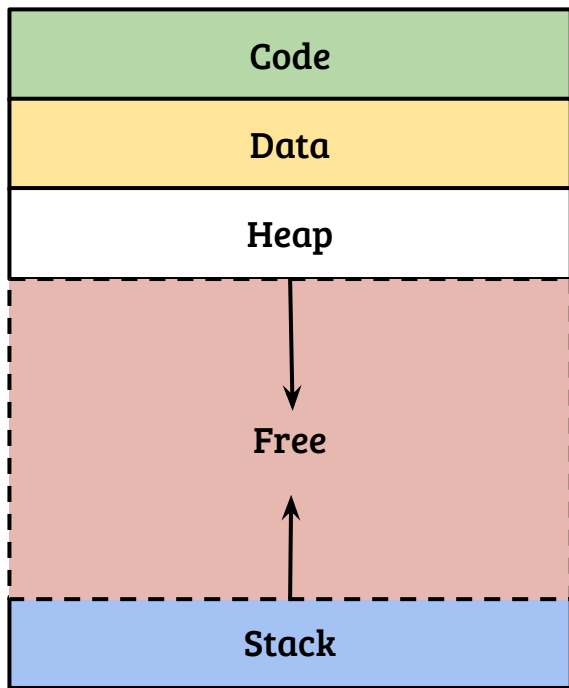


GPU Memory Access and Addressing

GPU generates memory addresses for Instructions and data Access

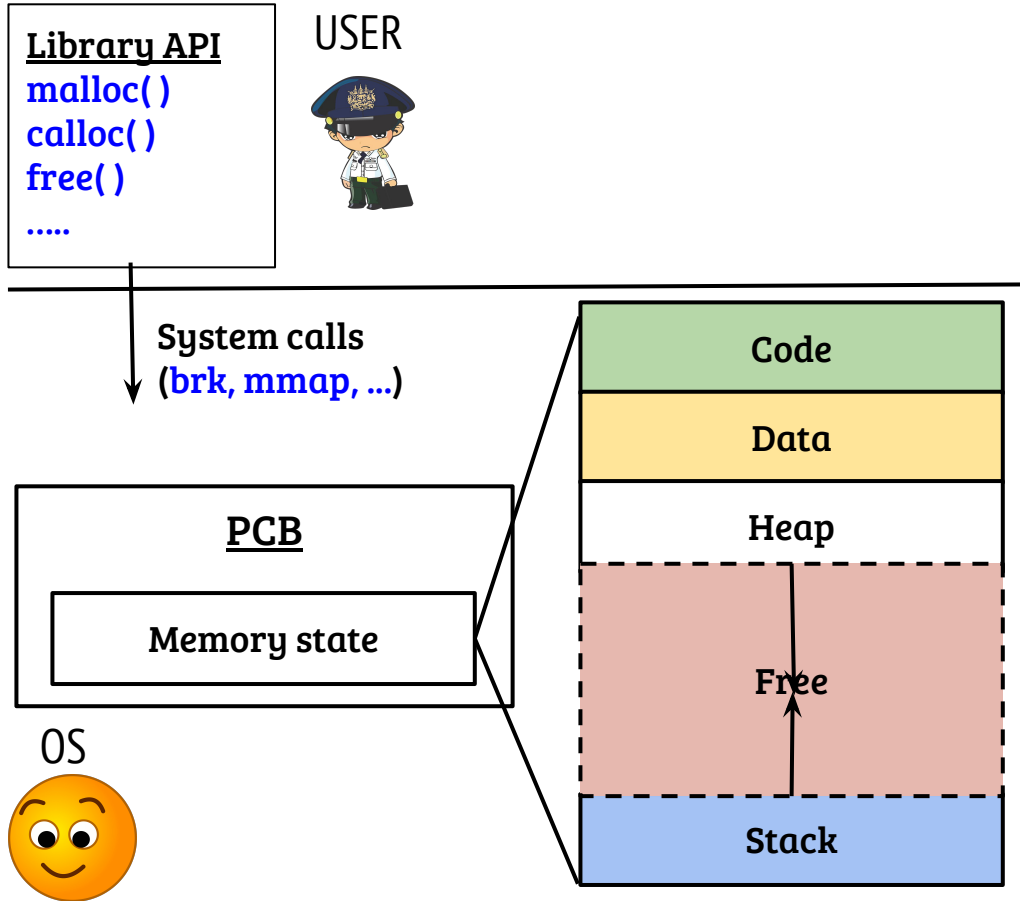


Address Space Abstraction



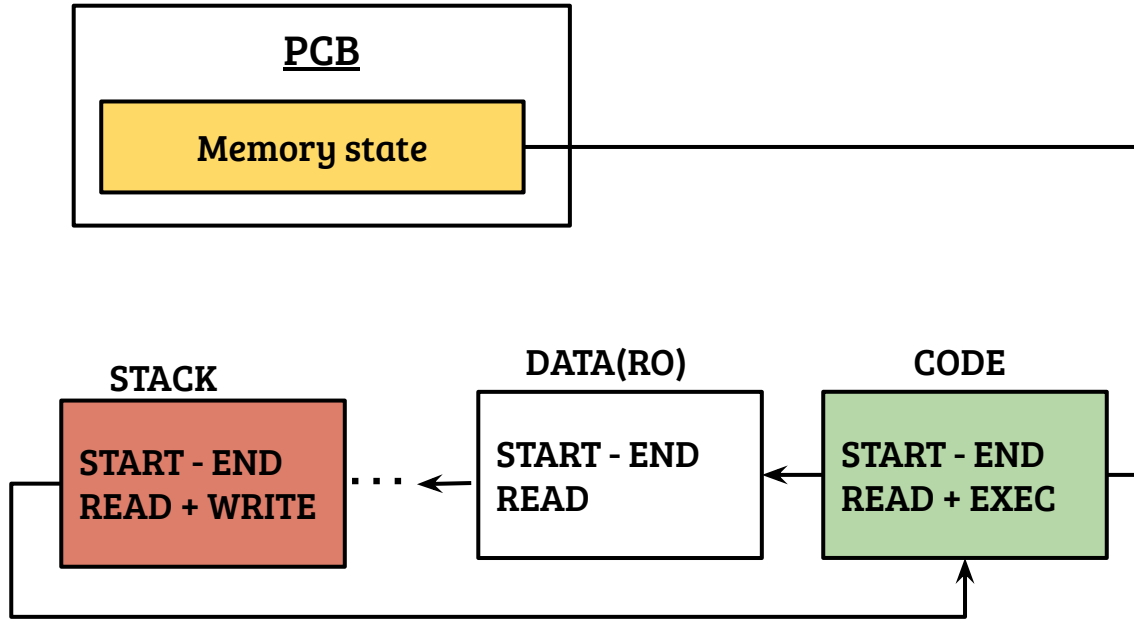
- Address space layout presents the same memory view to *all users (e.g., processes)*
 - Typically address space is virtual
 - Memory manager (e.g., OS) enables this virtual view
- The runtime system has the flexibility to layout the code and data with little constraints

User API for memory management



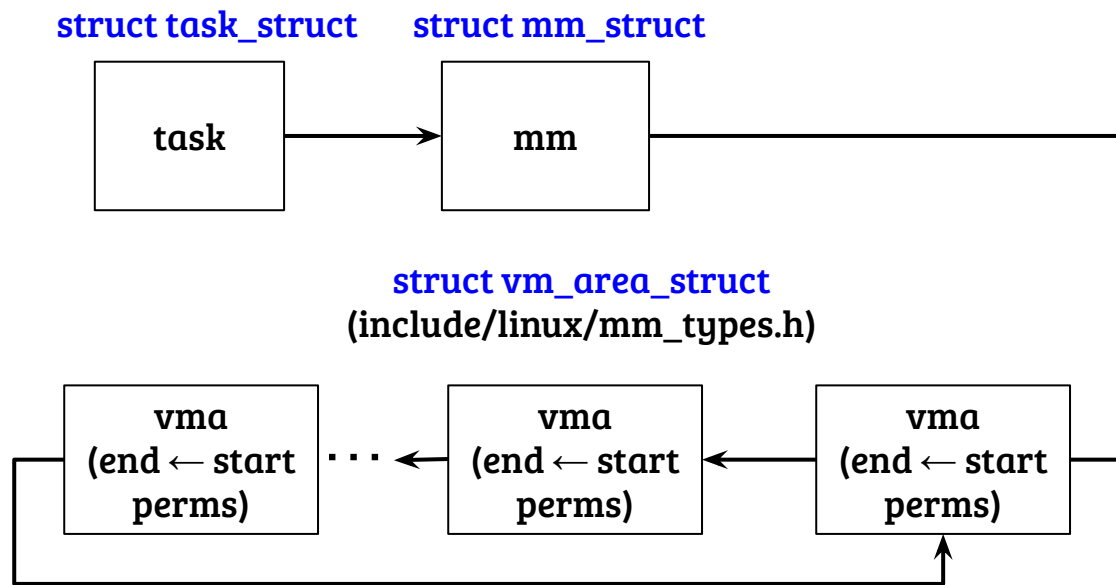
- Generally, user-level programs use library routines to (de)allocate memory (from the address space)
- OS provides address space manipulation system calls (e.g., `brk`, `mmap` etc.)

Memory state of OS PCB (example)



- Maintained as a sorted circular list accessible from PCB
- START and END never overlap between two segment areas
- Can merge/extend areas if permissions match

Linux Address Space (MM and VMA)



- start and end never overlaps between two vm areas
- can merge/extend vmas if permissions match
- Kernel maintains a maple tree to manage vmas (see `mm/mmap.c`)

Related code: `mmap_region`, `mas_find`, `vma_mas_store`, `vma_link`, `mt_find`, `unmap_vmas`

Paging

- Paging addresses the following issues with segmentation
 - External fragmentation caused due to variable sized segments
 - No support for discontinuous/sparse mapping

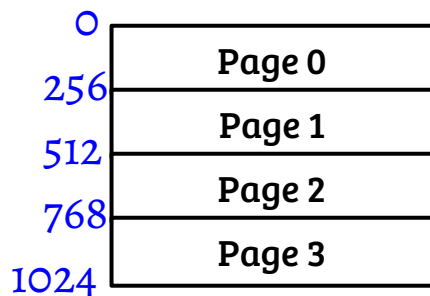
Paging

- Paging addresses the following issues with segmentation
 - External fragmentation caused due to variable sized segments
 - No support for discontinuous/sparse mapping
- The idea of basic paging
 - Partition the address space into fixed sized blocks (call it page)
 - Physical memory partitioned in a similar way (call it page frame)

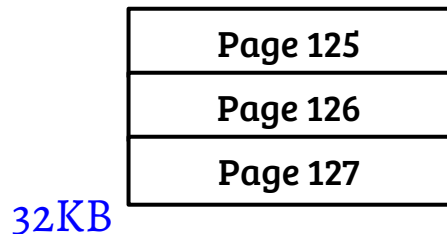
Paging

- Paging addresses the following issues with segmentation
 - External fragmentation caused due to variable sized segments
 - No support for discontinuous/sparse mapping
- The idea of basic 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

Paging example (pages)

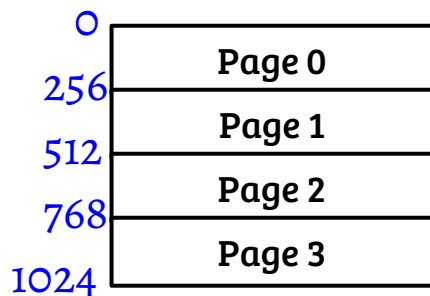


- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128

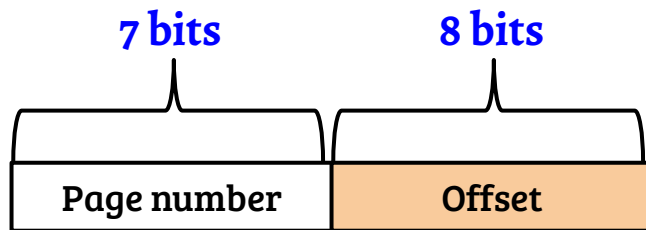


Process address
space

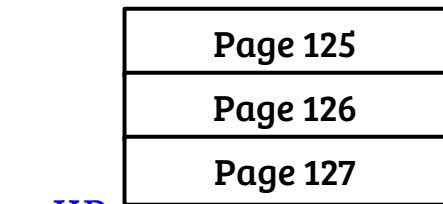
Paging example (pages)



- Virtual address size = 32KB, Page size = 256 bytes
- Address length = 15 bits {0x0 - 0x7FFF}
- # of pages = 128



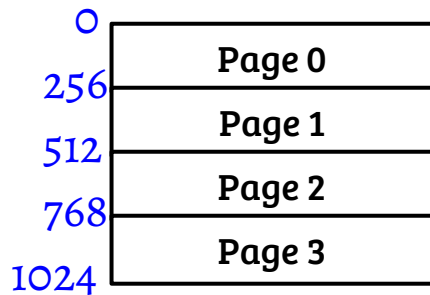
Virtual address



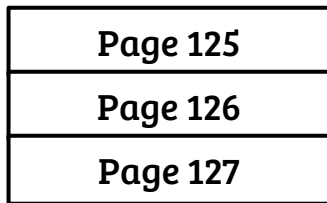
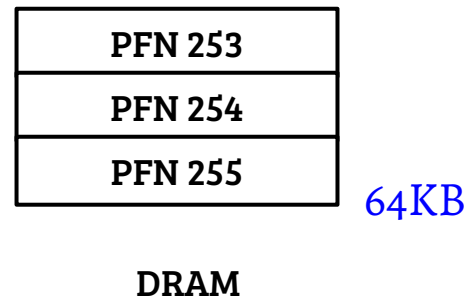
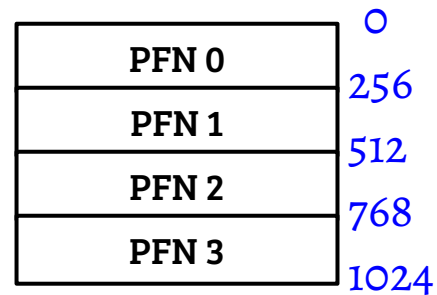
Process address
space

- Example: For Virtual address *0x0510*, Page number = 5, offset = 16

Paging example (page frames)



- Physical address size = 64KB
- Address length = 16 bits {0x0 - 0xFFFF}
- # of page frames = 256



Process address
space

32KB

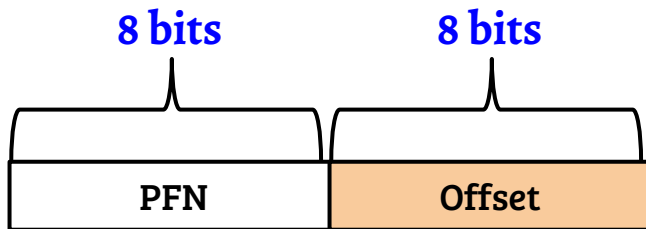
Paging example (page frames)

0	Page 0
256	Page 1
512	Page 2
768	Page 3
1024	

Page 125
Page 126
Page 127

Process address space

- Physical address size = 64KB
- Address length = 16 bits {0x0 - 0xFFFF}
- # of page frames = 256



Physical address

0	PFN 0	256
	PFN 1	512
	PFN 2	768
	PFN 3	1024

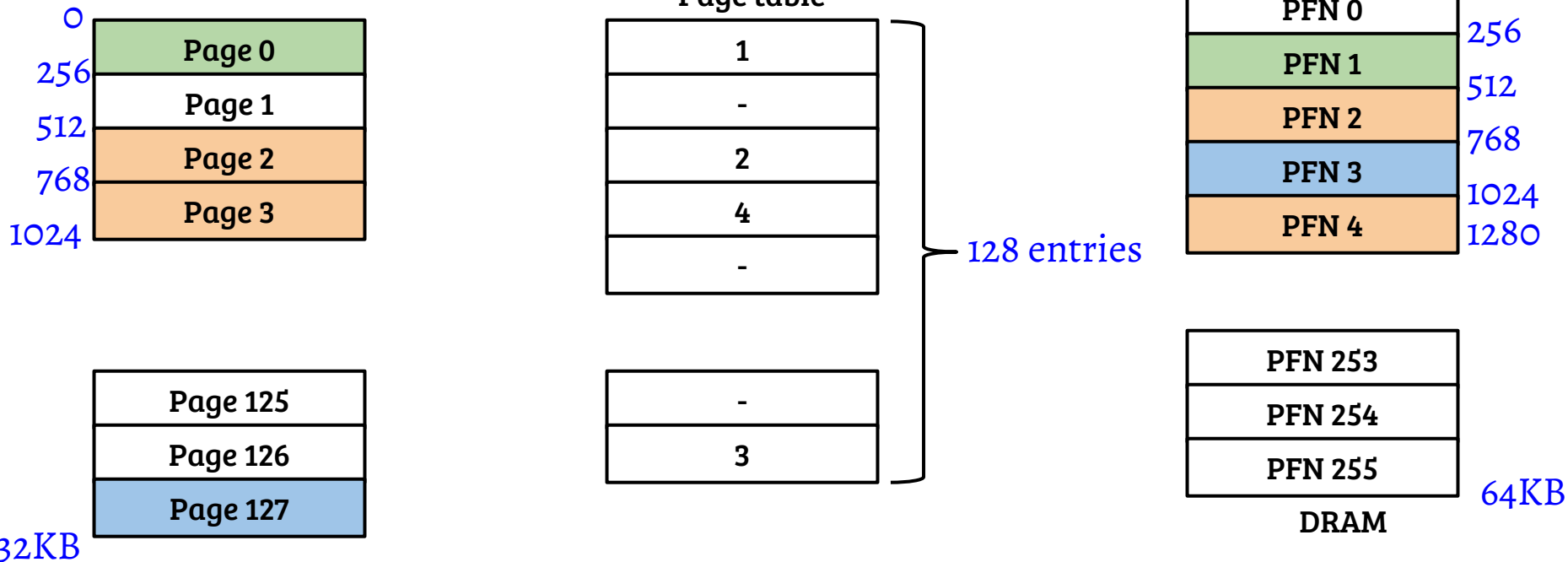
PFN 253
PFN 254
PFN 255

64KB

DRAM

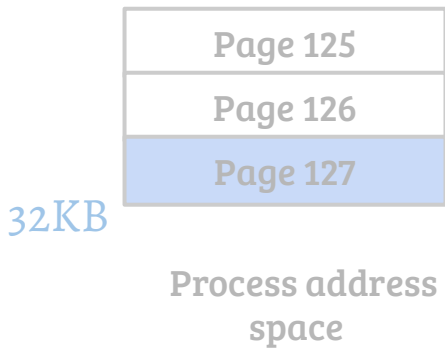
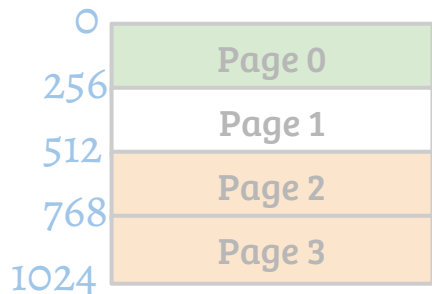
- Example: For physical address *0x1F51*, PFN = 31, offset = 81

Paging example (page table mapping)

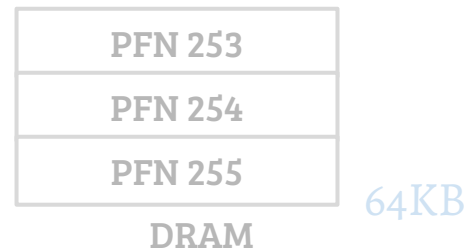
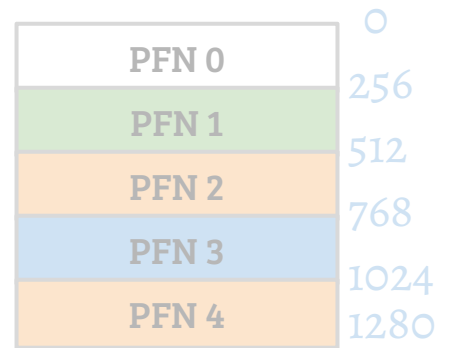


- Each entry in page table is called page table entry (PTE)
- Example mapping: page 0 $\Rightarrow$ PFN 1, page 2 $\Rightarrow$ PFN 2 and so on

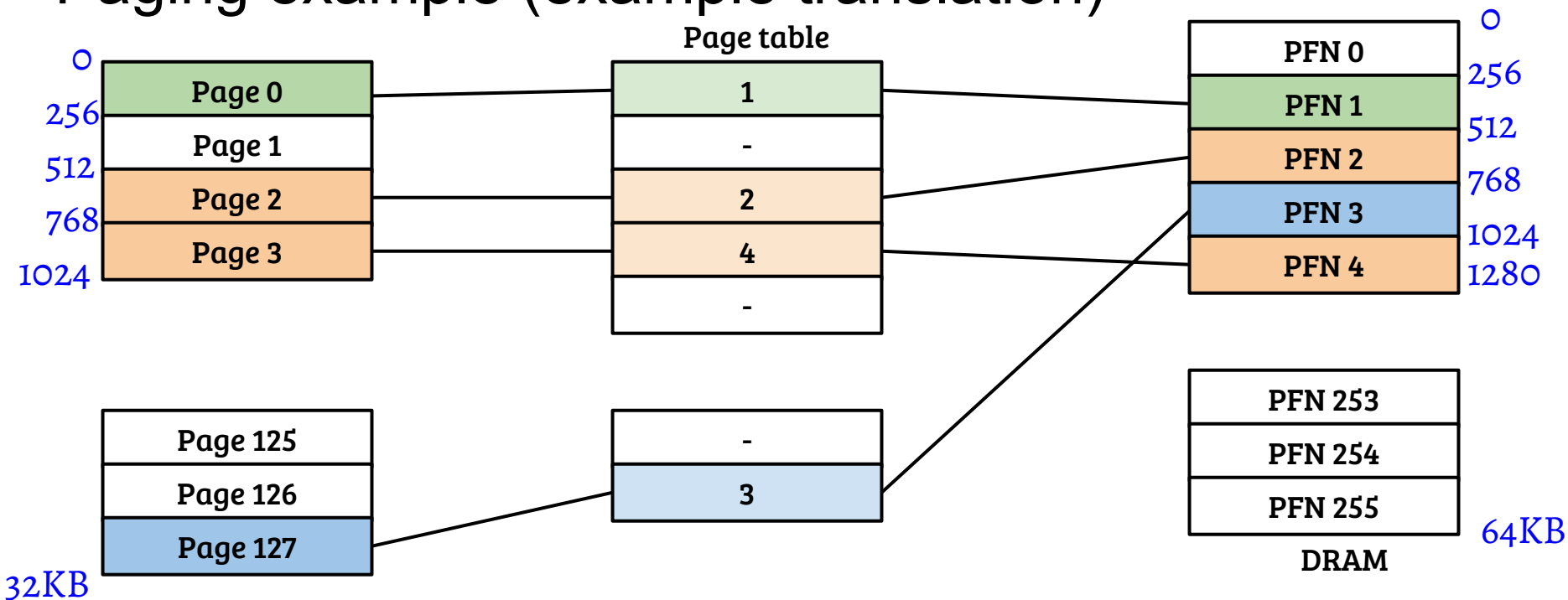
Paging example (page table walk)



```
PTW (vaddr V, PTable P)
// Input: Virtual address, Page table
// Returns physical address
{
    Entry = P[V >> 8];
    if (Entry.present)
        return (Entry.PFN << 8) + (V & 0xFF);
    Raise PageFault;
}
```

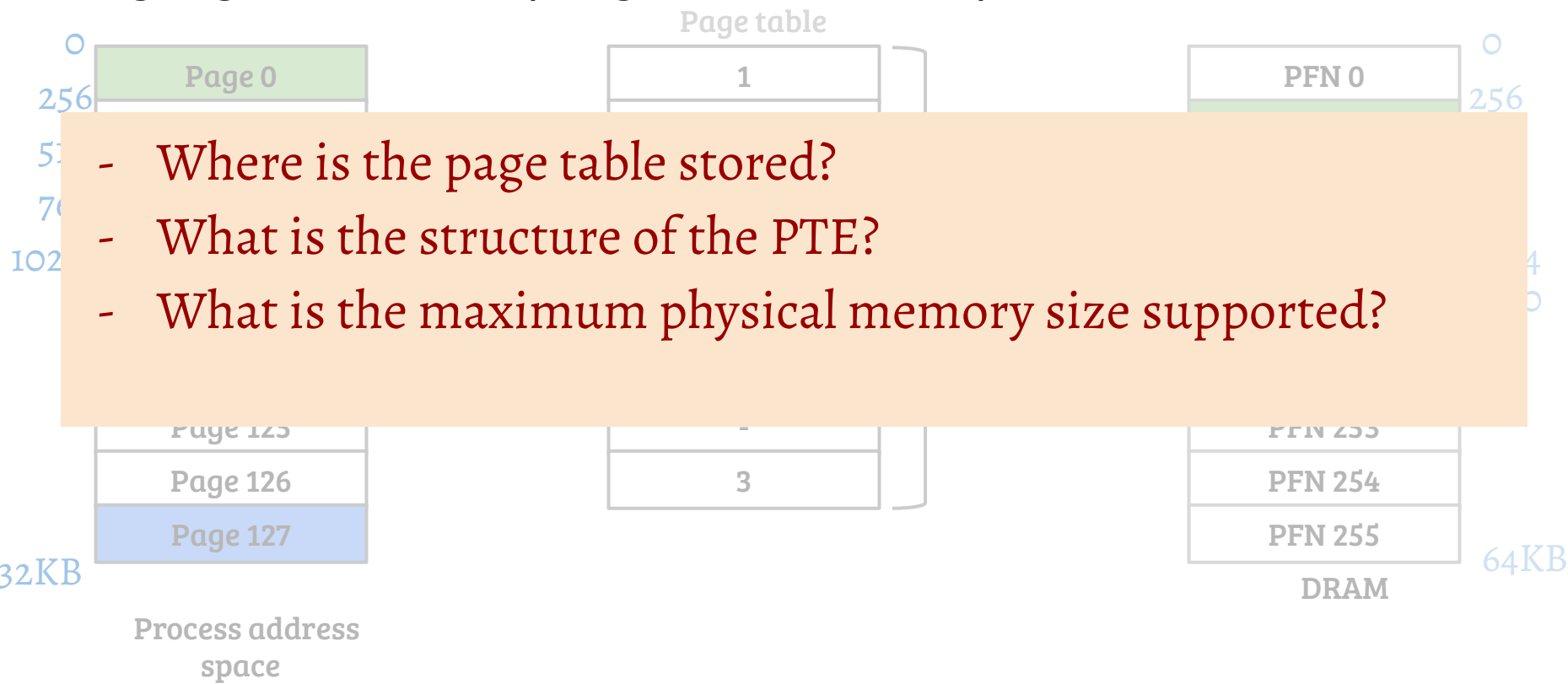


Paging example (example translation)

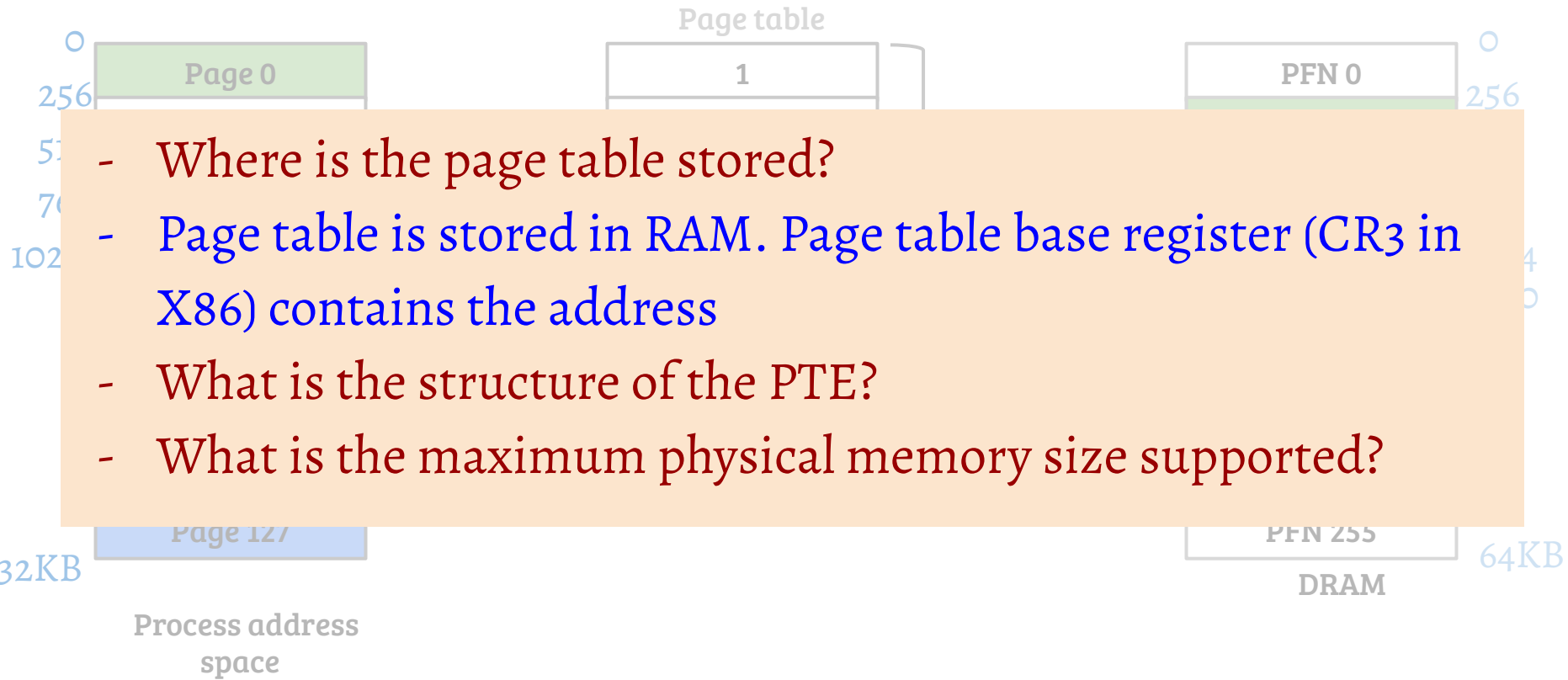


- Process address space
- Virtual address 0x10 translates to physical address 0x110
 - Virtual address 0x7FF0 translates to physical address 0x3F0

Paging example (page table walk)



Paging example (page table walk)



Paging example (structure of an example PTE)



- PFN occupies a significant portion of PTE entry (8 bits in this example)

P

Present bit, 1 $\Rightarrow$ entry is valid

W

Write bit, 1 $\Rightarrow$ Write allowed

S

Privilege bit, 0 $\Rightarrow$ only kernel mode access is allowed

A

Accessed bit, 1 $\Rightarrow$ Address accessed (set by H/W during walk)

D

Dirty bit, 1 $\Rightarrow$ Address written (set by H/W during walk)

X

Execute bit, 1 $\Rightarrow$ Instruction fetch allowed for this page

Reserved/unused bits

Paging example (Page table entries)

0	Page 0
256	Page 1
512	Page 2
768	Page 3
1024	

Page table	
0x125	
0x0	
0x207	
0x407	
0x0	

0	PFN 0
256	PFN 1
512	PFN 2
768	PFN 3
1024	PFN 4
1280	

Page 125
Page 126
Page 127

0x0
0x307

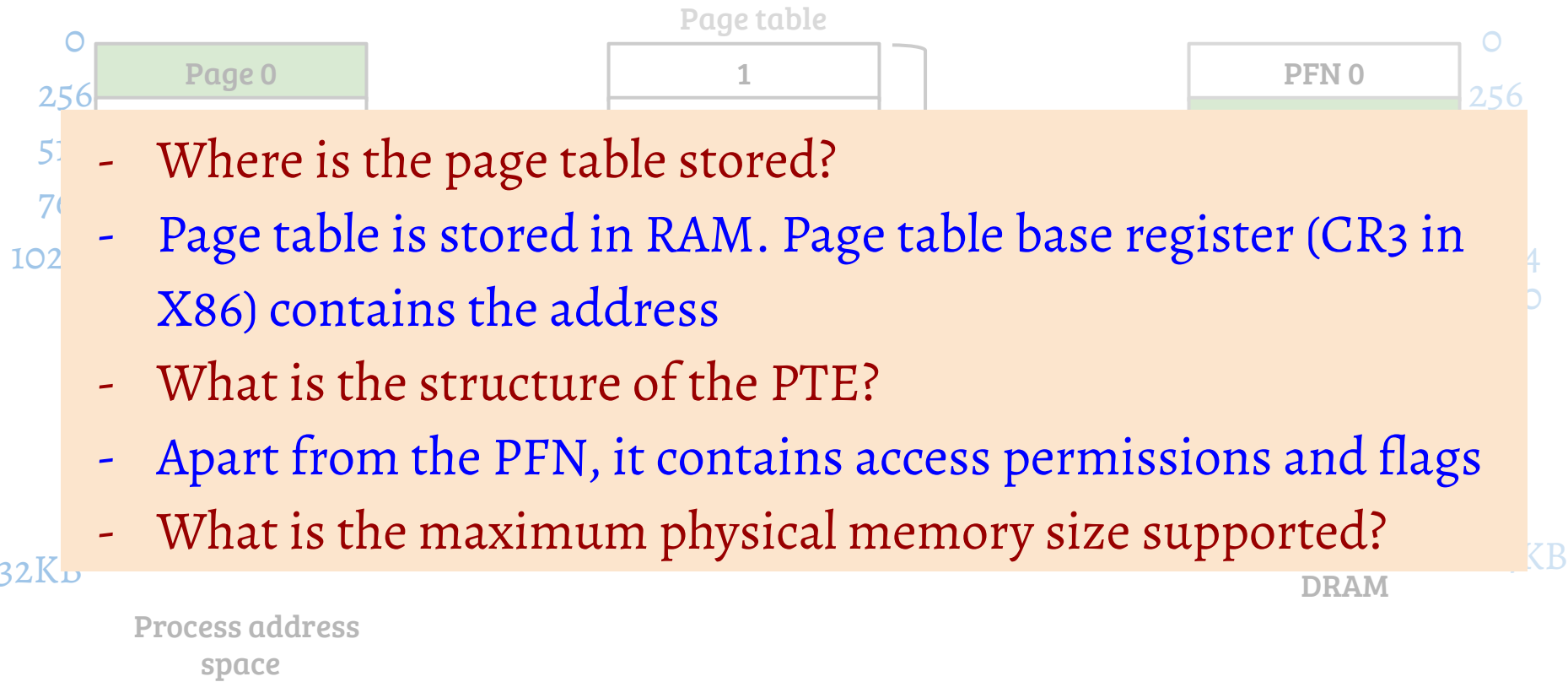
PFN 253
PFN 254
PFN 255

64KB

Process address
space

- Code: Page 0 (Read and Execute)
- Data: Page 2 and Page 3 (Read and Write)
- Stack: Page 127 (Read and Write)

Paging example (page table walk)



Paging example (page table walk)

- Where is the page table stored?
- Page table is stored in RAM. Page table base register (CR3 in X86) contains the address
- What is the structure of the PTE?
- Apart from the PFN, it contains access permissions and flags
- What is the maximum physical memory size supported?
- For this example, 8-bits can be used to specify 256 page frames. Maximum RAM size = $256 * 256 = 64\text{KB}$

Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?

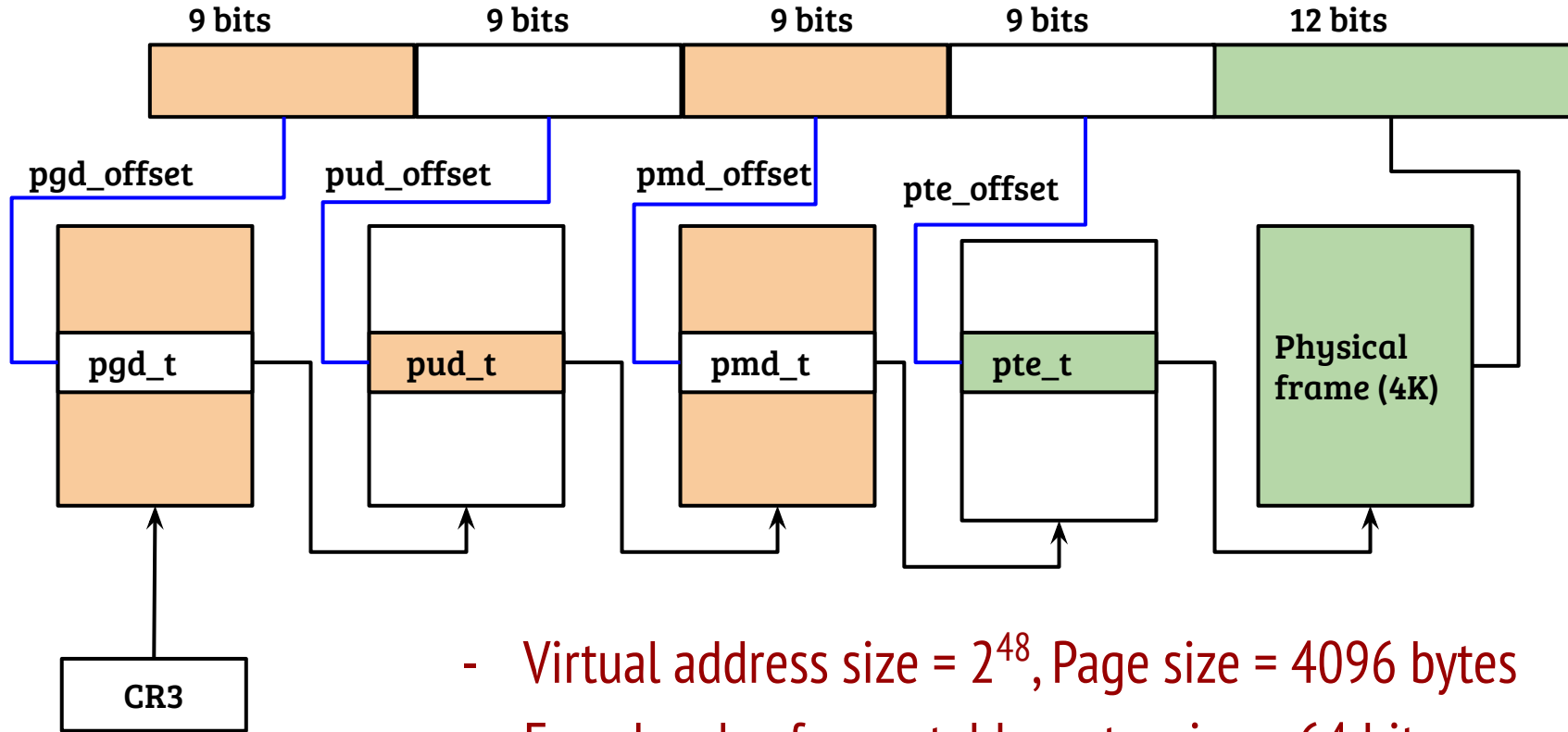
Paging: one level of page table may not be feasible!

- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?
- Large page size results in *internal fragmentation*
- Assuming page size = 4KB, How many entries are required in a one-level paging system?

Paging: one level of page table may not be feasible!

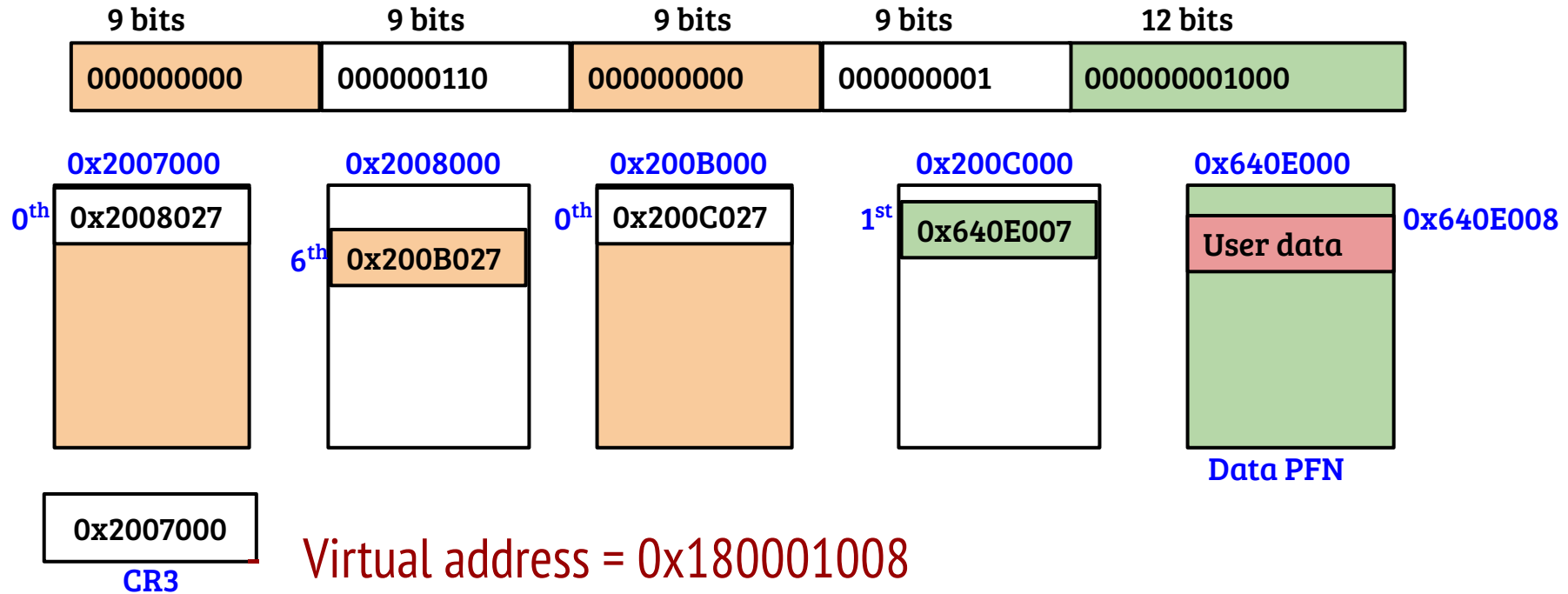
- Consider a 32-bit address space (=4GB)
- What should be the page size for this system?
- Large page size results in *internal fragmentation*
- Assuming page size = 4KB, How many entries are required in a one-level paging system? (2^{20} entries)
- Not possible to hold 2^{20} entries in a single page
- Therefore, multi-level page tables are used in modern systems

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



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

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

Paging: translation efficiency

```
0x20100: mov $0, %rax;
```

```
0x20102: mov %rax, (%rbp); // sum=0
```

- 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

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	
Page	PTE
0x20	0x750
0x7FFF	0x890

- 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

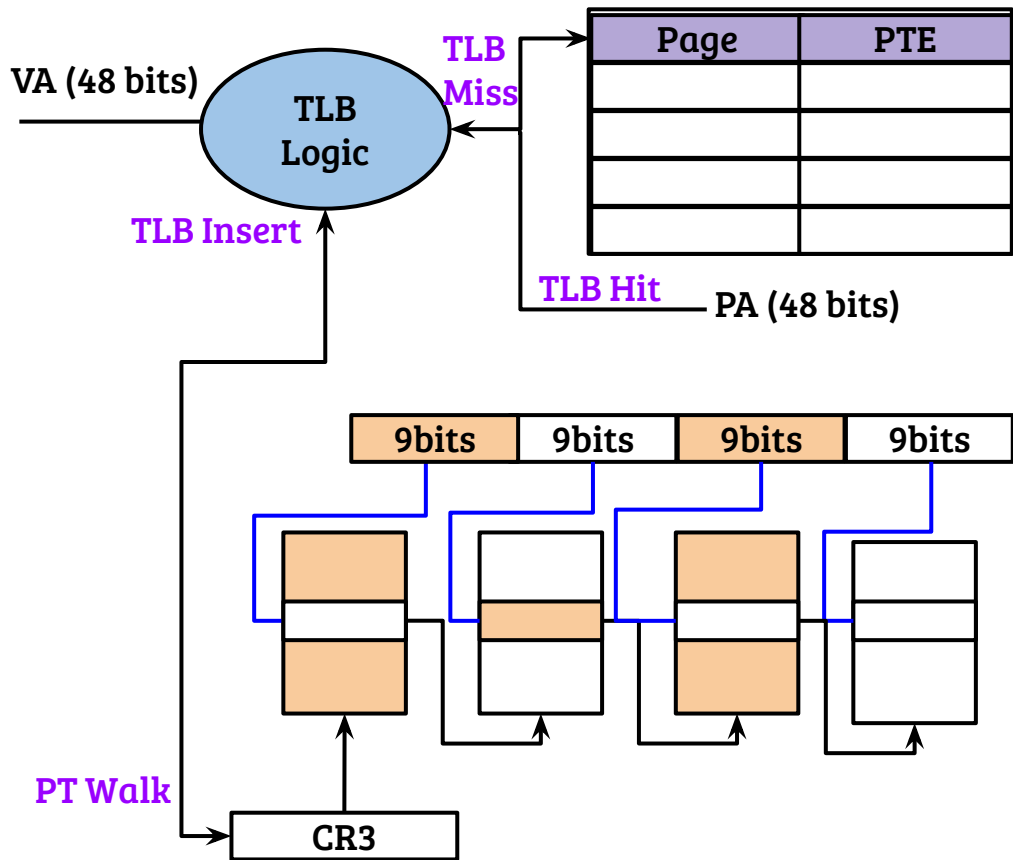
Paging: translation efficiency

```
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.

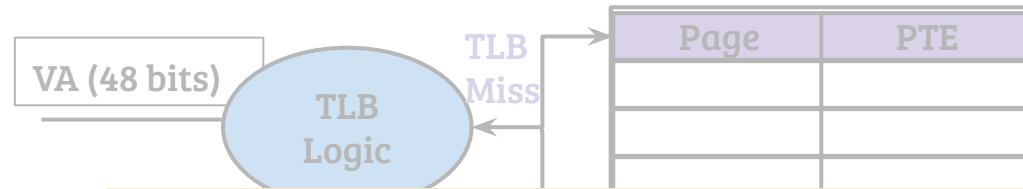
required (for translation) during the execution of the above code?

Address translation (TLB + PTW)



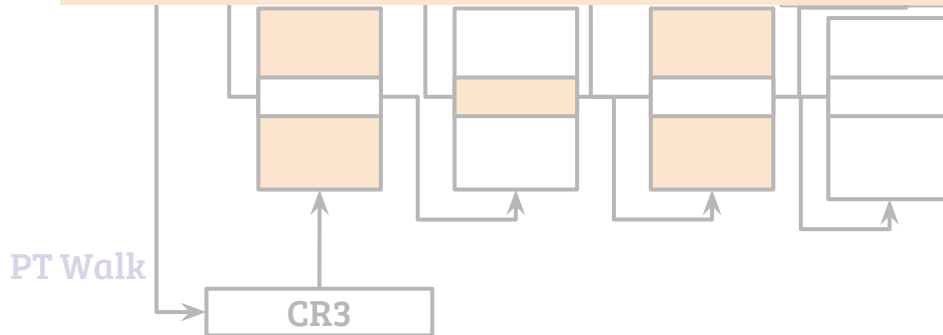
- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



- 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

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

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
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

Process (A)

Process (B)

- Assume that, process A is currently executing. What happens when process B is scheduled?

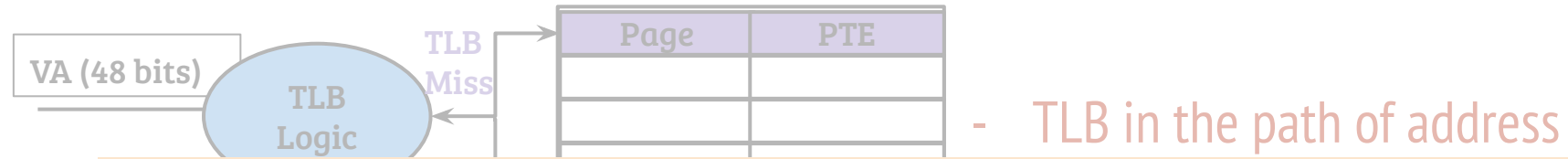
- A) Do nothing
- B) Flush the whole TLB
- C) Some other solution

- Address space identified (ASID) along with each TLB entry to identify the process

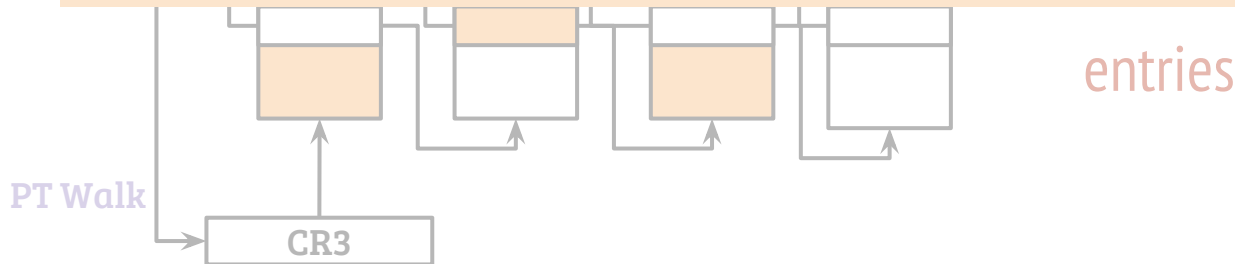
ASID	Page	PTE
A	0x100	0x200007
A	0x101	0x205007
B	0x100	0x301007
B	0x101	0x302007

TLB

Address translation (TLB + PTW)



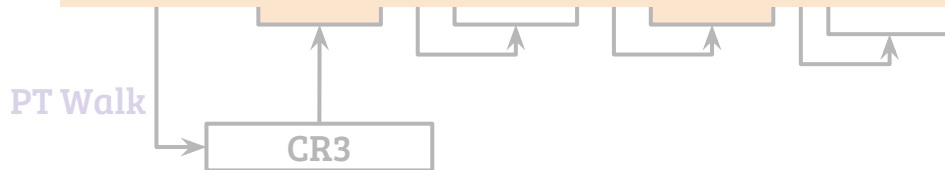
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?



Address translation (TLB + PTW)



- 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?



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;  
    push [kstack , errorCode]  
    Raise pageFault;  
} // Simplified
```

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;  
    push [kstack, errorCode]  
    Raise pageFault;  
} // Simplified
```

Error code

Other and unused	I	R	U	W	P
------------------	---	---	---	---	---

P

Present bit, 1 $\Rightarrow$ fault is due to protection

W

Write bit, 1 $\Rightarrow$ Access is write

U

Privilege bit, 1 $\Rightarrow$ Access is from user mode

R

Reserved bit, 1 $\Rightarrow$ Reserved bit violation

I

Fetch bit, 1 $\Rightarrow$ Access is Instruction Fetch

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

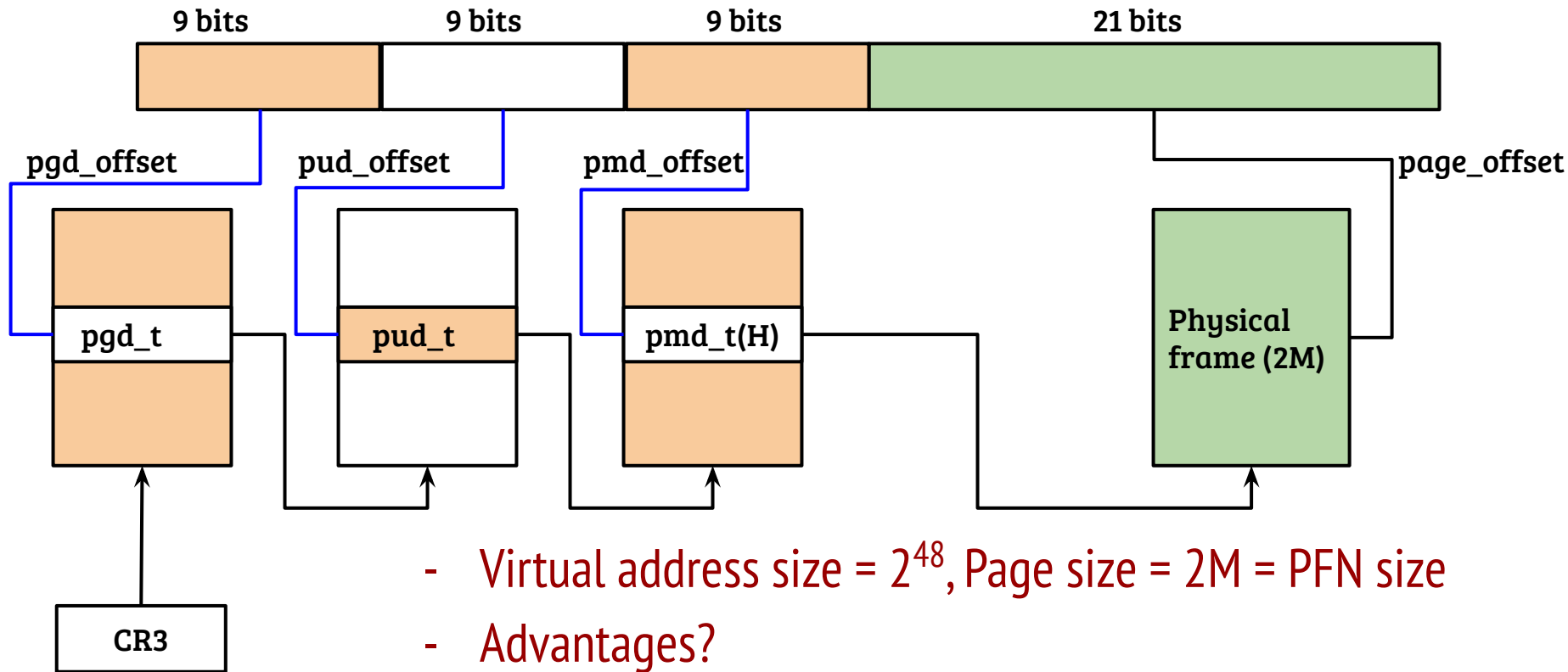
Page fault handling in X86: OS fault handler

```
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)

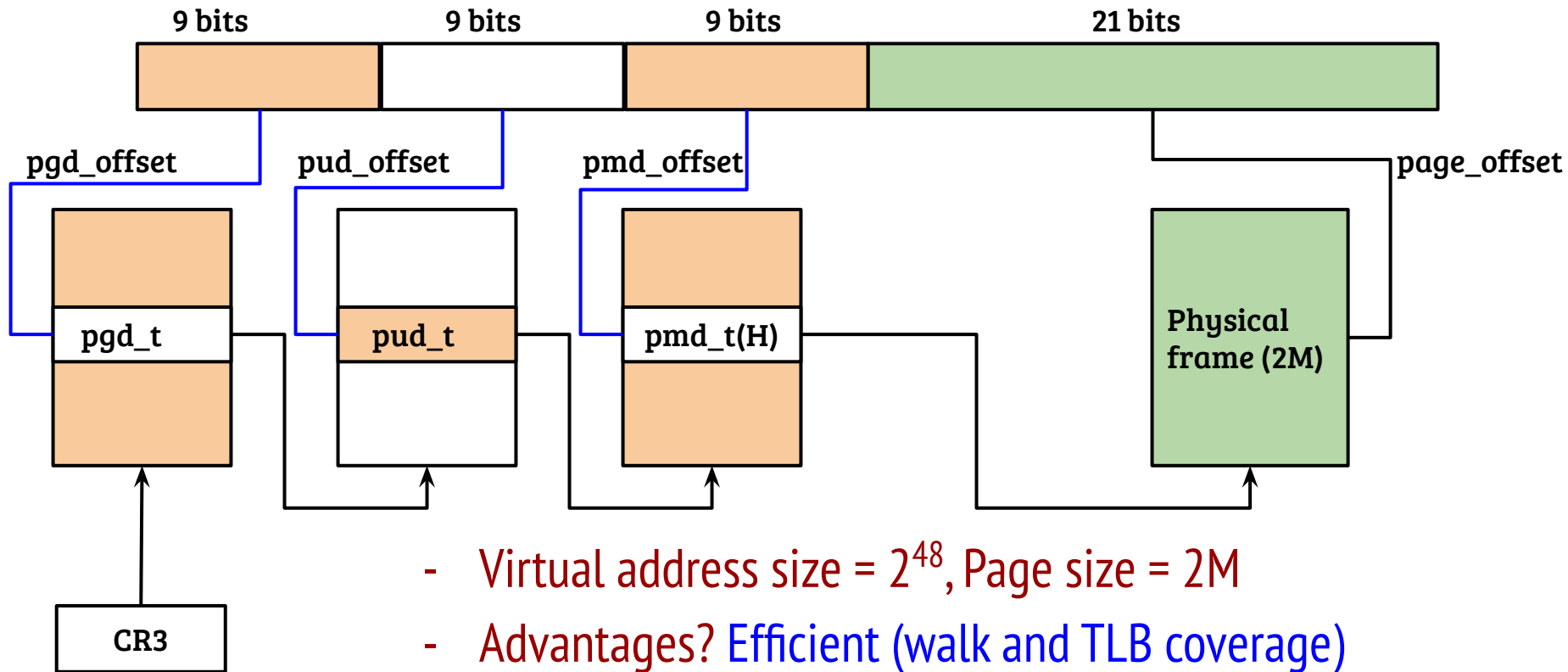
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler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

Efficient translation: Huge page support



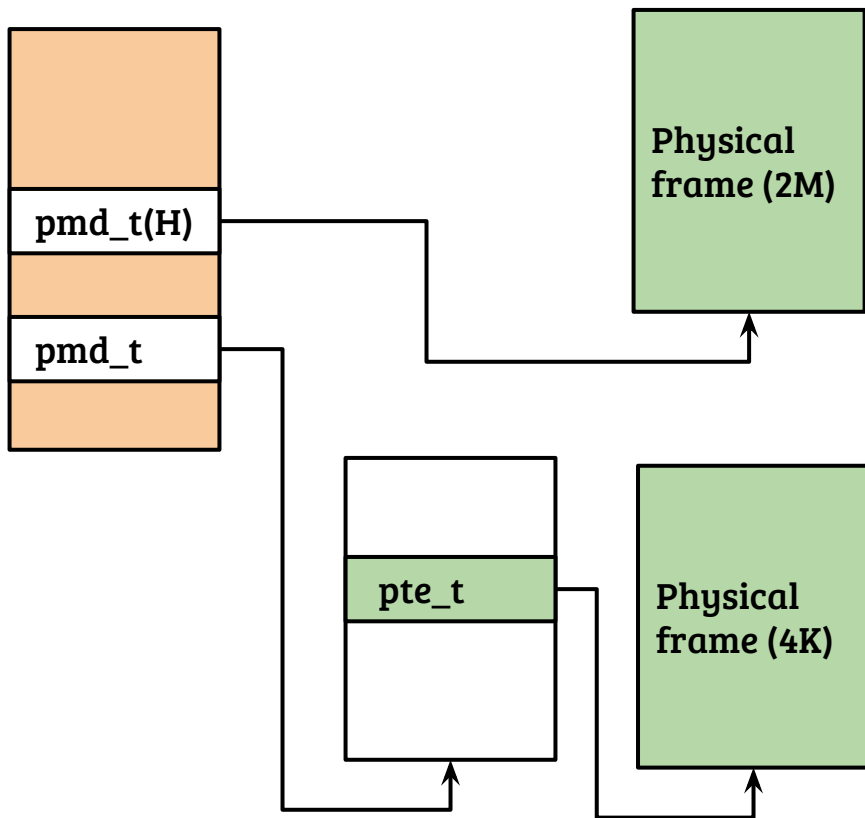
- Virtual address size = 2^{48} , Page size = 2M = PFN size
- Advantages?
- Disadvantages?

Efficient translation: Huge page support



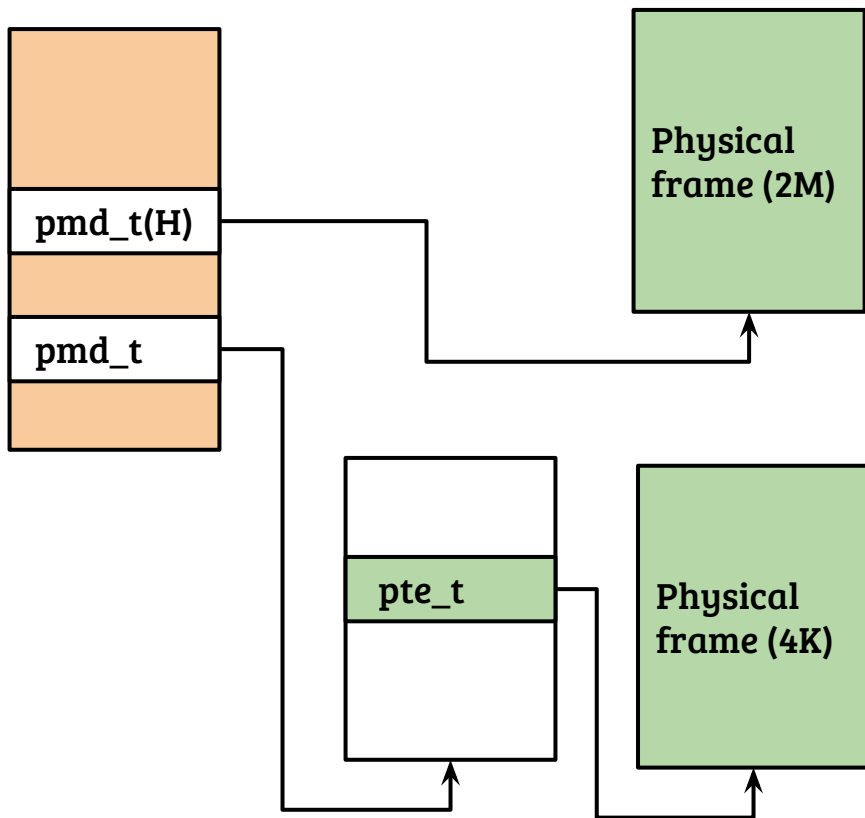
- Virtual address size = 2^{48} , Page size = 2M
- Advantages? Efficient (walk and TLB coverage)
- Disadvantages? Inefficient management

Mixed page size support



```
walk_pmd(pmd, vaddr) {  
    if(pmd.H)  
        paddr = pmd.nextL() + (vaddr & pmask);  
    else  
        pte = pmd.nextL() + pte_offset(vaddr)  
    } // Simplified H/W logic
```

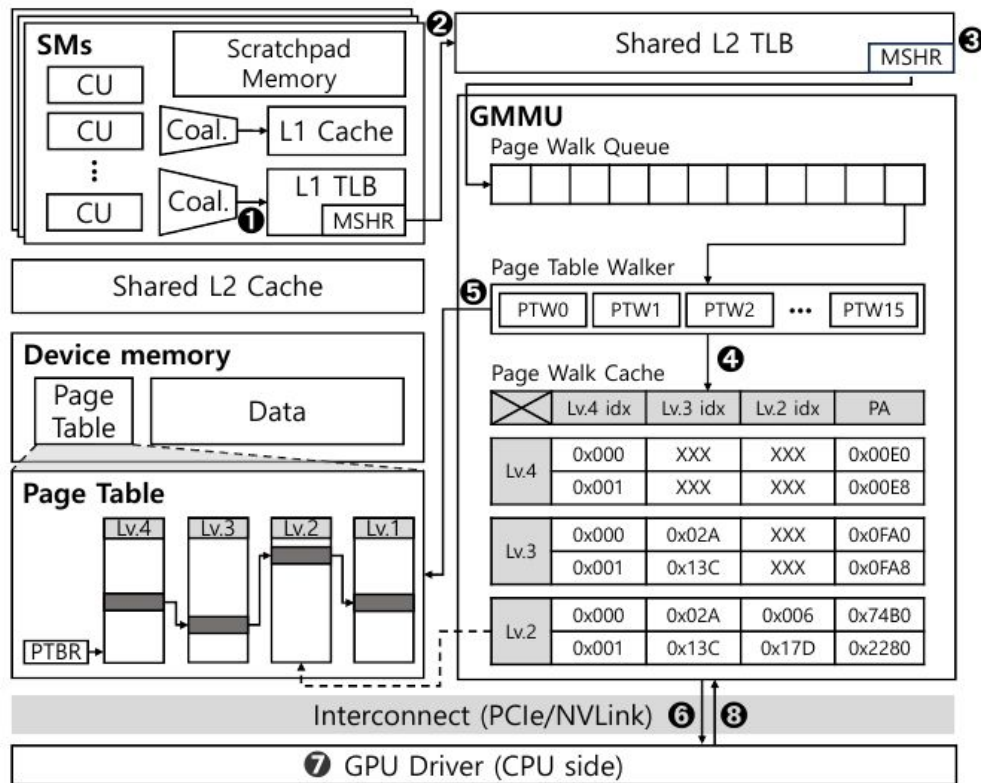
Mixed page size support



```
walk_pmd(pmd, vaddr) {  
    if(pmd.H)  
        paddr = pmd.nextL() + (vaddr & pmask);  
    else  
        pte = pmd.nextL() + pte_offset(vaddr)  
} // Simplified H/W logic
```

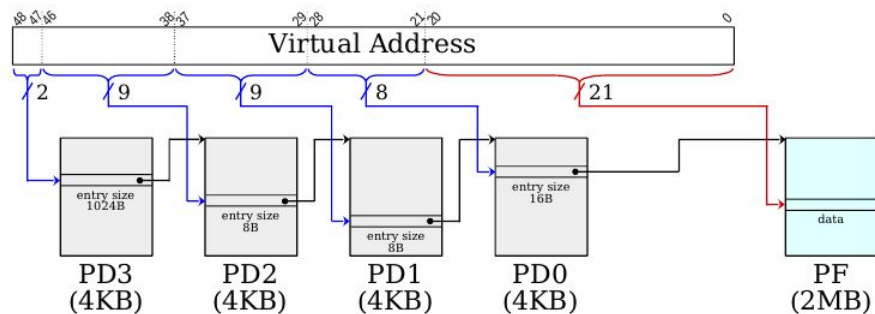
- The OS may use the hardware support to implement any policy
- Transparent hugepage (THP) in Linux tries to create huge page mapping in w/o explicit user space assistance
- Policy knobs through sysfs

NVIDIA GPU MMU (GMMU)¹

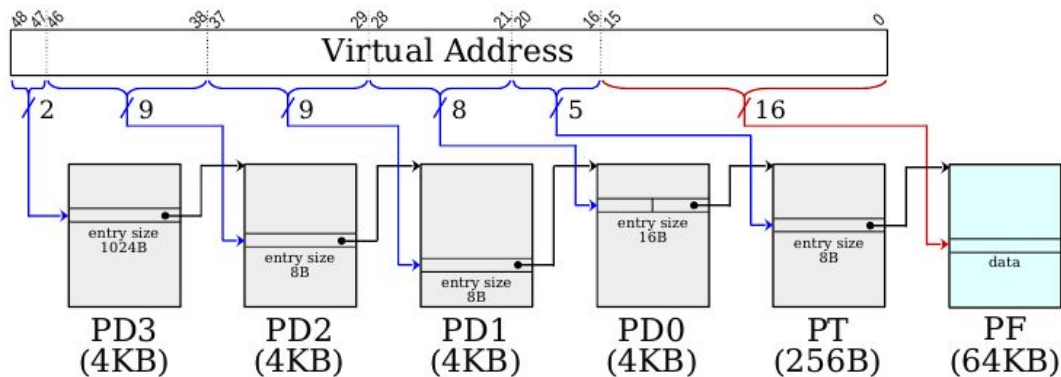


- Per-SM L1 TLB and a shared L2 TLB
- Multi-threaded page table walker. What is the issue with per-SM page table walker?
 - Inefficient resource management
- Scheduling PTW is an interesting problem

NVIDIA GPU Page Table²



(a) 2MB-page translation.



(b) 64KB-page translation.

- NVIDIA GPUS support three page sizes—4KB, 64KB and 2MB
- Different indexing is used by the hardware to walk the page table
- Mapping is performed by the device driver
 - Reverse engineering shows page sizes increase with contig. access