

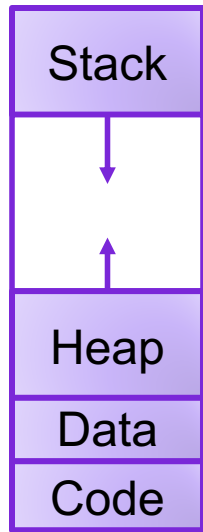
# Lecture 19: Virtual Memory (cont'd)

---

CS 105

Spring 2024

# Review: Address Translation



Virtual Address

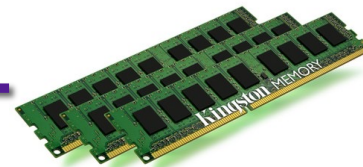


invalid

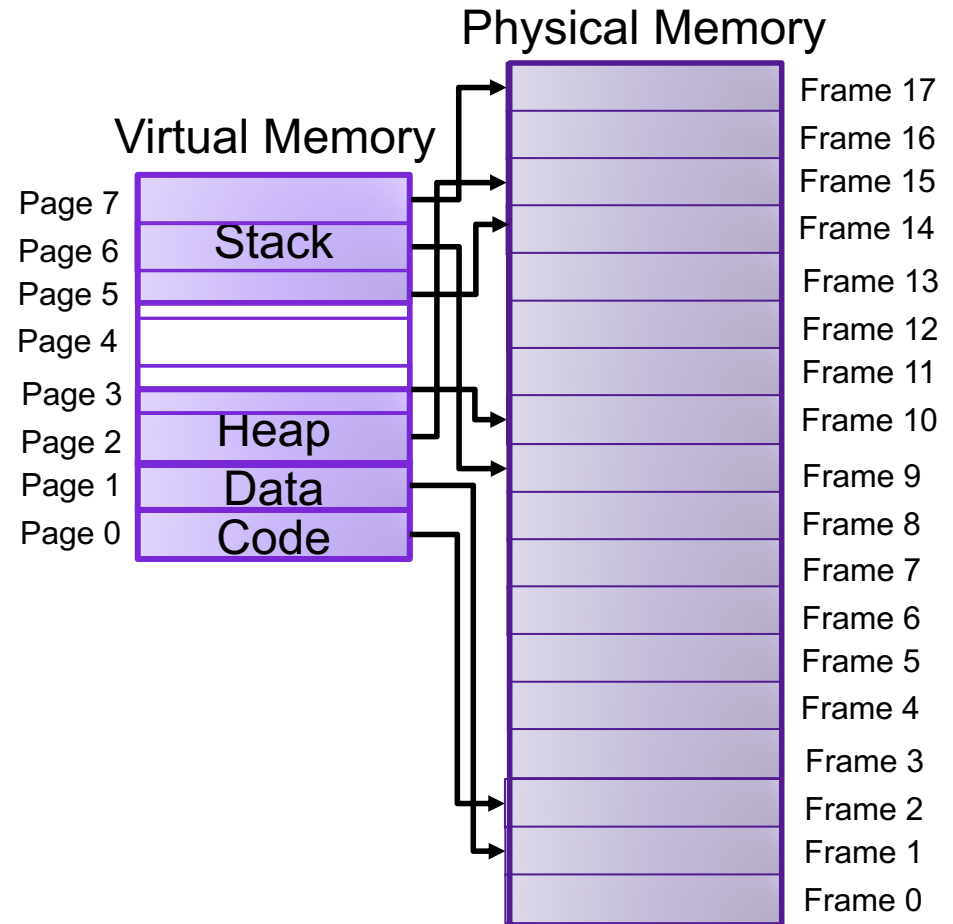
Exception

Physical Address

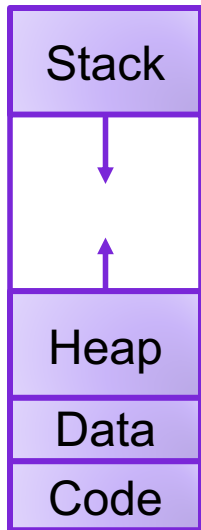
Data



# Review: Paging



# Review: Virtual Pages



page#    offset

vaddr

MMU

page table

v	Frame	Access
1	47	R,W
0	NULL	R,W
0	13	R,W
1	42	R,X
⋮		

NULL page or  
access not allowed

Invalid page

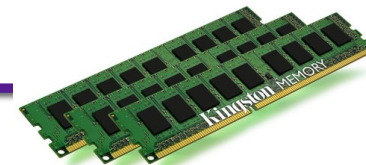
SegFault

Page Fault

paddr =

Frame[page#]    offset

Data



# Review: Paging

Assume that you are currently executing a process P with the following page table on a system with 16 byte pages:

	v	Frame	Access
⋮			
0xEA8B	1	0x47	R,W
0xEA8A	0	NULL	R,W
0xEA89	0	0x13	R,W
0xEA88	1	0x23	R,X
⋮			

- What is the physical address that corresponds to the virtual address 0xEA8B2?
- What is the physical address that corresponds to the virtual address 0xEA8A7?
- What is the physical address that corresponds to the virtual address 0xEA89A?

# Review: Evaluating Paging

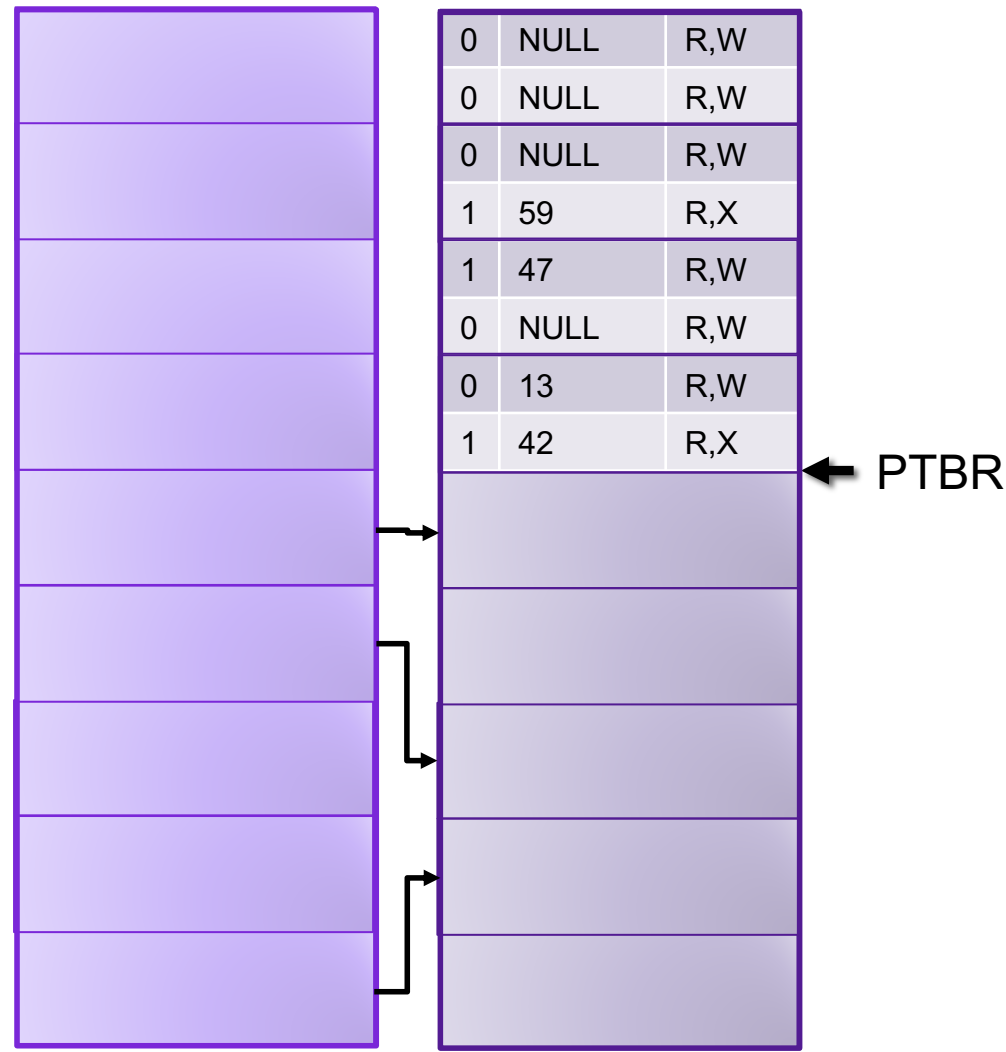


- **Isolation:** don't want different process states collided in physical memory
- **Efficiency:** want fast reads/writes to memory
- **Sharing:** want option to overlap for communication
- **Utilization:** want best use of limited resource
- **Virtualization:** want to create illusion of more resources



# Traditional Paging

- page table is stored in physical memory
- implemented as array of page table entries
- Page Table Base Register (PTBR) stores physical address of beginning of page table
- Page table entries are accessed by using the page number as the index into the page table



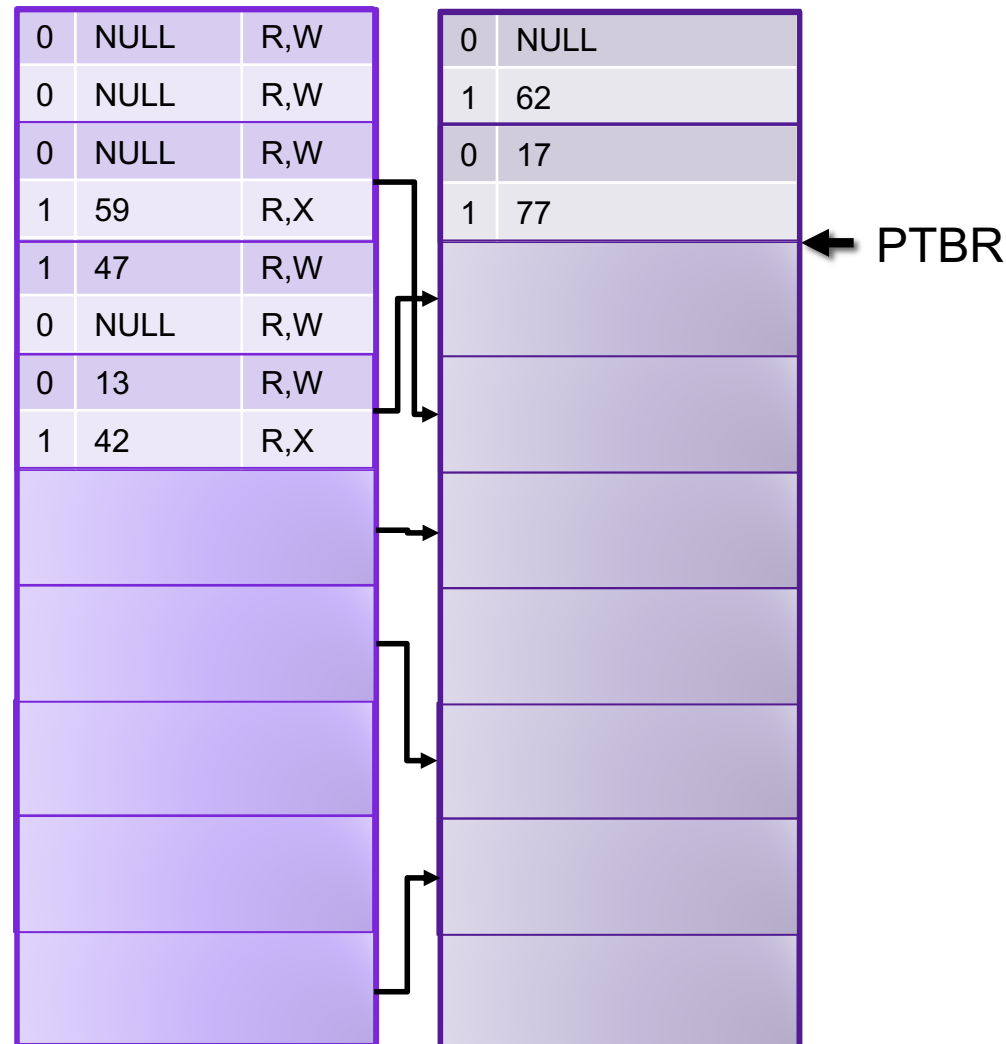
# Problems with Paging

- **Memory Consumption:** page table is really big
  - Example: consider 48-bit address space, 4KB ( $2^{12}$ ) page size, assume each page table entry is 8 bytes.
  - Larger pages increase internal fragmentation
- **Performance:** every data/instruction access requires *two* memory accesses:
  - One for the page table
  - One for the data/instruction

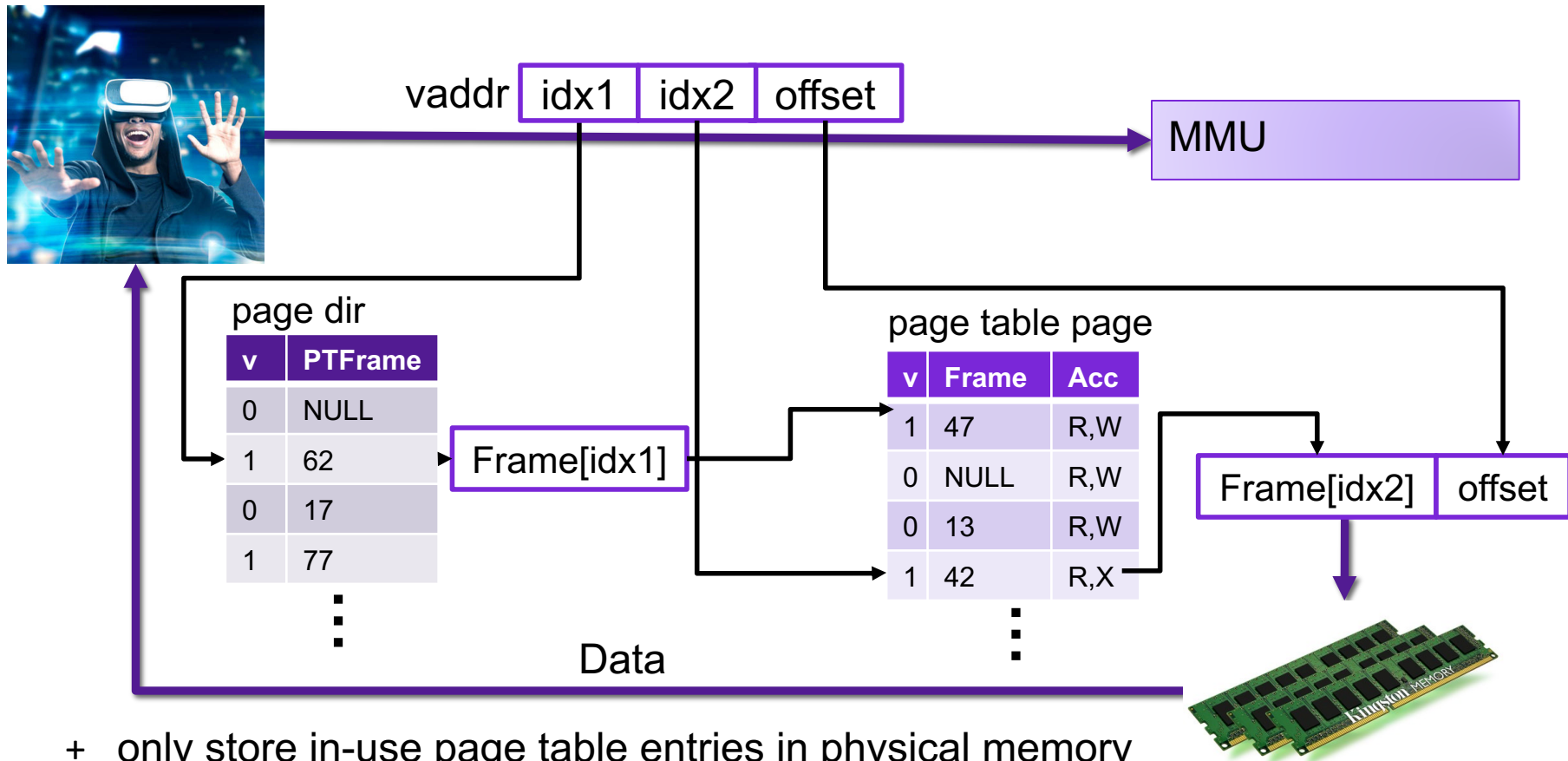


# Two-level Page Tables

- page table is stored in virtual memory pages
- page directory is stored in physical memory (page table for the page table)
- Implemented as array of page directory entries
- Page Table Base Register (PTBR) stores physical address of beginning of page directory



# Two-level Page Tables



- + only store in-use page table entries in physical memory
- + easier to allocate page table
- more memory accesses

# Example: Two-level Page Tables

Assume you are working on an architecture with a 32-bit virtual address space in which idx1 is 4 bits, idx2 is 12 bits, and offset is 16 bits. 

4 bit idx1	12 bit idx2	16 bit offset
------------	-------------	---------------

- How big is a page in this architecture?  **$2^{16}$  bytes = 64 KB**
- How big is a page table entry in this architecture? **16 bytes**

# Exercise: Two-level Page Tables


Assume you are still working on that architecture.

4 bit idx1 | 12 bit idx2 | 16 bit offset

Compute the physical address corresponding to each of the virtual address (or answer "invalid"):

- a) 0x00000013
- b) 0x20022002
- c) 0x10015555

page directory	
v	PTFrame
0x0	1
0x1	1
0x2	0
0x3	0
	⋮
0xF	0

		page table	
Frame 0	v	Frame	Acc
	0x0	1	0x0047 R,W
	0x1	0	NULL R,W
	0x2	0	0x0013 R,W
	0x3	1	0x0042 R,X
			⋮
Frame 1			
Frame 2	v	Frame	Acc
	0x0	0	0x002A R
	0x1	1	0xCAFE R,W
	0x2	0	NULL R,W
	0x3	0	13 R,W
			⋮

# Multi-level Page Tables

- Problem: How big does the page directory get? **1 GB**
  - Assume you have a 48-bit address space
  - Assume you have 4KiB pages
  - Assume you have 8 byte page table entries/page directory entries



- Goal: Page Table Directory should fit in one frame
- **Multi-level page tables:** add additional level(s) to tree

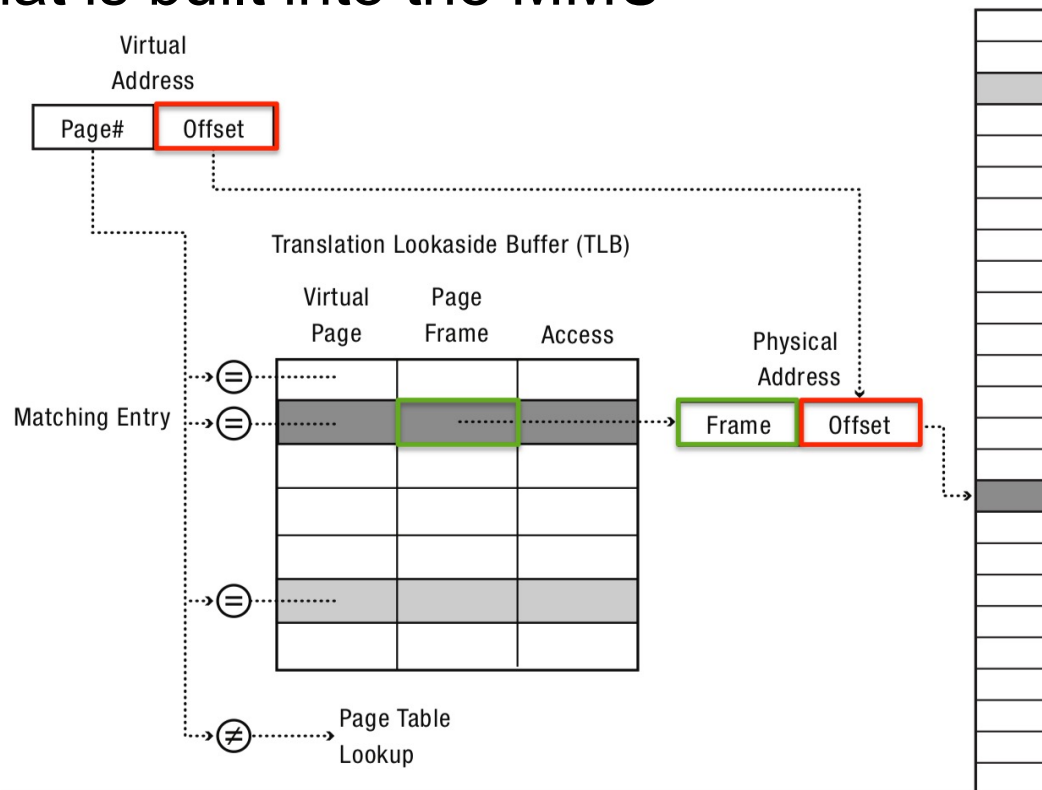


# Review: Problems with Paging

- **Memory Consumption:** page table is really big
  - Example: consider 64-bit address space, 4KB ( $2^{12}$ ) page size, assume each page table entry is 8 bytes.
  - Larger pages increase internal fragmentation
- **Performance:** every data/instruction access requires ~~two~~<sup>five</sup> memory accesses:
  - One for ~~the page table~~ each of the four levels of page table
  - One for the data/instruction

# Translation-Lookaside Buffer (TLB)

- General idea: if address translation is slow, cache some of the answers
- **Translation-lookaside buffer** is an address translation cache that is built into the MMU



# Exercise: TLB

TLB												
idx	v	tag	PPN	v	tag	PPN	v	tag	PPN	v	tag	PPN
0	1	03	B	0	07	6	1	28	3	0	01	F
1	1	31	0	0	12	3	1	3E	4	1	0B	1
2	0	2A	A	0	11	1	1	1F	8	1	07	5
3	1	07	3	0	2A	A	0	1E	2	0	21	B

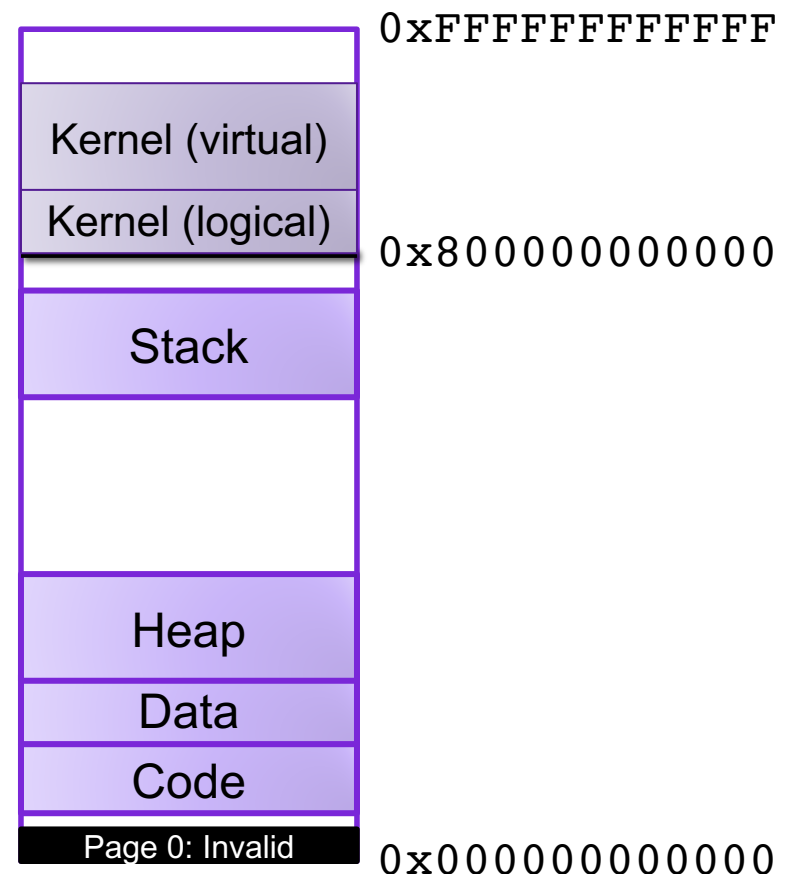
Assume you are running on an architecture with a one-level page table with 4096 byte pages. For each of the following virtual addresses, determine whether the address translation is stored in the TLB. If so, give the corresponding physical address

- 0x7E37C
- 0x16A48



# Example: The Linux x86 Address Space

- Use "only" 48-bit addresses (top 16 bits not used)
- 4KiB pages by default
  - supports larger "superpages"
- Four-level page table
- Physical memory stores memory pages, memory-mapped files, cached file pages
- Updates are periodically written to disk by background processes
- Page eviction algorithm uses variant of LRU called 2Q
  - approximates LRU with clock
  - maintains two lists (active/inactive)
- Stack is marked non-executable
- Virtual address of stack/heap start are randomized each time process is initialized



# Example: Core i7 Memory Accessing

