

# CSE 120

# Principles of Operating Systems

Spring 2023

Lecture 11: Paging

Amy Ousterhout

# Administrivia

---

- Midterm grades are posted
  - ◆ Bring questions to office hours or ask after class
  - ◆ Statistics are available on Piazza
  - ◆ Submit any regrade requests by 5/17
- Project 2 and Homework #3 are ongoing

# Virtual Memory

---

- The abstraction that the OS provides for managing memory is **virtual memory**
- Two views of memory, called address spaces:
  - ◆ **Virtual address space** (seen by program)
  - ◆ **Physical address space** (actual allocation of memory)

# Recall: Demo of Virtual Memory

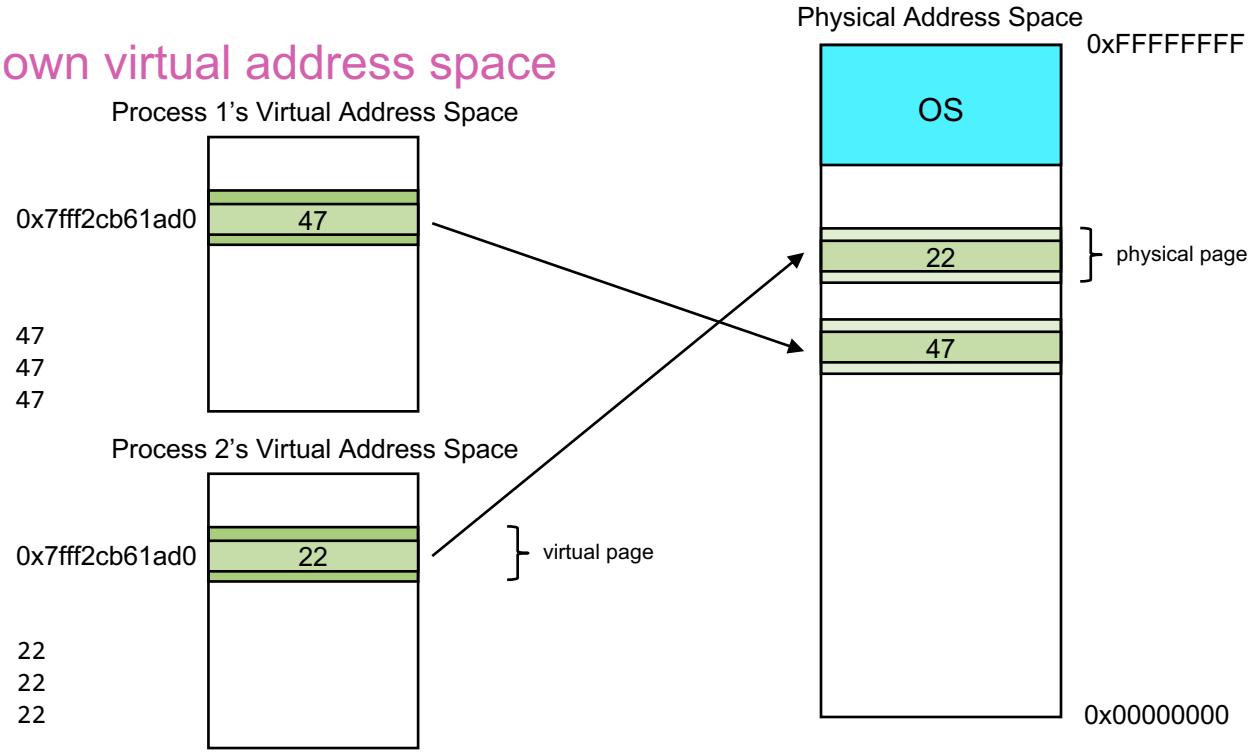
- Each process has its own virtual address space

Process 1

```
$ ./a.out 47
addr of value: 0x7fff2cb61ad0. value: 47
addr of value: 0x7fff2cb61ad0. value: 47
addr of value: 0x7fff2cb61ad0. value: 47
```

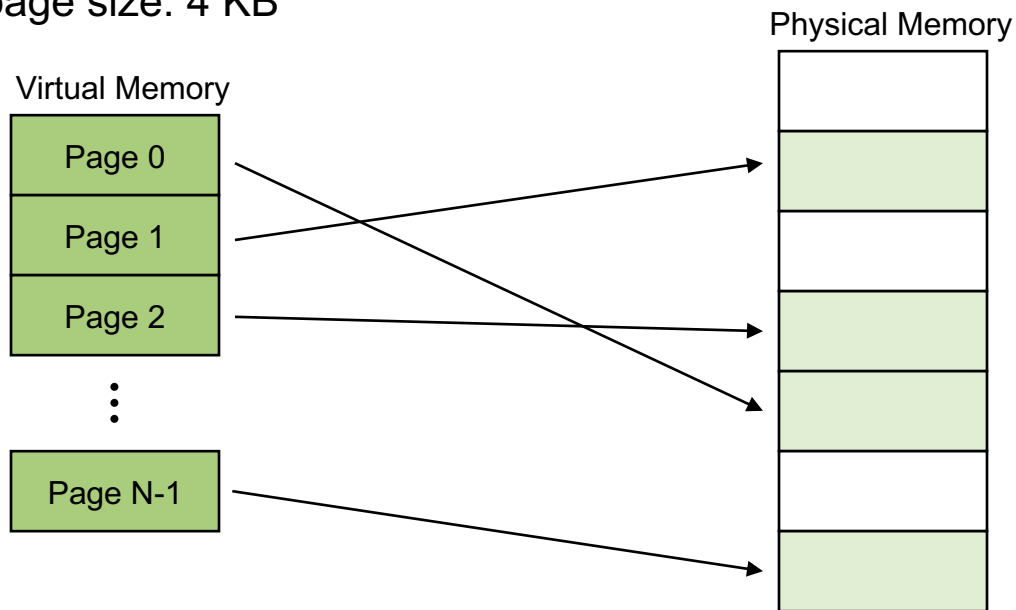
Process 2

```
$ ./a.out 22
addr of value: 0x7fff2cb61ad0. value: 22
addr of value: 0x7fff2cb61ad0. value: 22
addr of value: 0x7fff2cb61ad0. value: 22
```



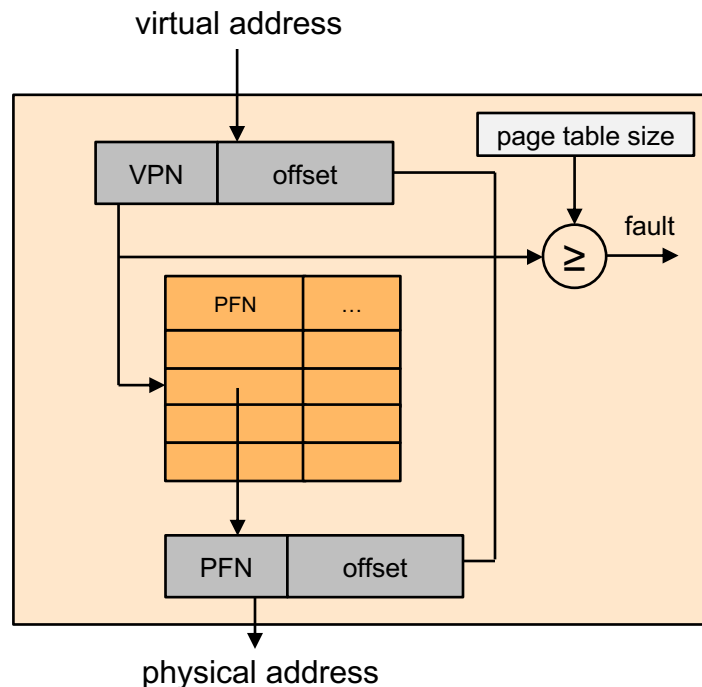
# Paging

- Divide physical and virtual memory into fixed-sized chunks calls **pages**
  - ♦ Common page size: 4 KB



# Paging – Address Translation

- Virtual addresses
  - ♦ Two parts: **virtual page number (VPN)** and **offset**
- Page tables
  - ♦ Map virtual page number to **page frame number (PFN)**
- Concatenate PFN and offset to get the physical address



# Paging - Advantages

---

- Easy to allocate memory
  - ◆ Memory comes from a free list of fixed-size chunks
  - ◆ Allocating a page is just removing it from the list
  - ◆ External fragmentation is not a problem
- Easy to swap out chunks of a program
  - ◆ All chunks are the same size
  - ◆ Use valid bit to detect references to swapped pages

# Today's Outline

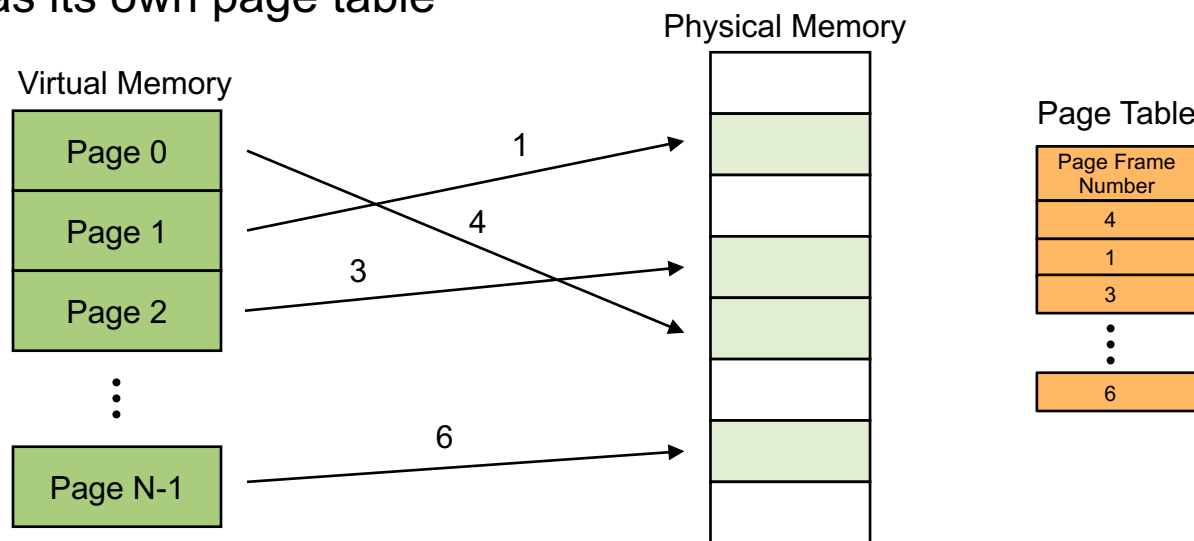
---

- Paging Mechanisms
  - ♦ What information does a page table contain?
  - ♦ How can we represent page tables efficiently?
  - ♦ Page table management



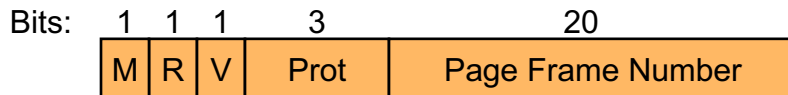
# Page Tables

- **Page tables**: define the mapping between virtual pages and physical pages
- Data structure managed by the OS (and accessed by hardware)
- Each process has its own page table



# Page Table Entries (PTEs)

- Page table consists of **Page Table Entries (PTEs)**



- Modify** bit – whether or not the page has been written (set on write)
- Reference** bit – whether the page has been accessed (set on read or write)
- Valid** bit – whether or not the PTE is valid
  - Checked each time the virtual address is used
- Protection** bits – which operations are allowed on this page
  - Read, write, execute
  - Protections for page 0 often set to no-read, no-write, no-execute
- Page frame number** (PFN) determines physical memory location

# Today's Outline

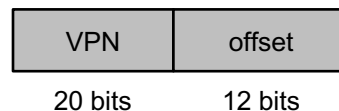
---

- Paging Mechanisms
  - ♦ What information does a page table contain?
  - ♦ How can we represent page tables efficiently?
  - ♦ Page table management

# Memory Requirements for Page Tables

- Assume linear page tables (e.g., an array)
- Assume 4 KB pages
  - ♦ 12 bits for offset within each page
- Assume 4 bytes per PTE
- With a 32-bit address space?
  - ♦ 20 bits for virtual page number, so  $2^{20}$  PTEs per process
  - ♦  $2^{20}$  PTEs x 4 bytes per PTE = 4 MB per process
- With a 64-bit address space?
  - ♦ 52 bits for virtual page number, so  $2^{52}$  PTEs per process
  - ♦  $2^{52}$  PTEs x 4 bytes per PTE = 16 **petabytes** per process
- Linear page tables are not efficient

32-bit virtual address



64-bit virtual address



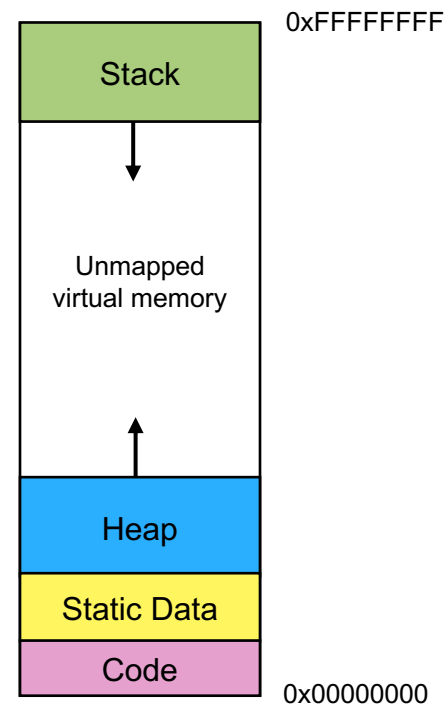
# Managing Page Tables

---

- How can we reduce the space required for page tables?
- Try using larger pages
  - ♦ Huge pages, e.g., 2 MB or 1 GB
- Increases internal fragmentation
- Doesn't solve the problem
- With a 64-bit address space:
  - ♦ 2 MB page size requires 8 TB of page tables
  - ♦ 1 GB page size requires 64 GB of page tables

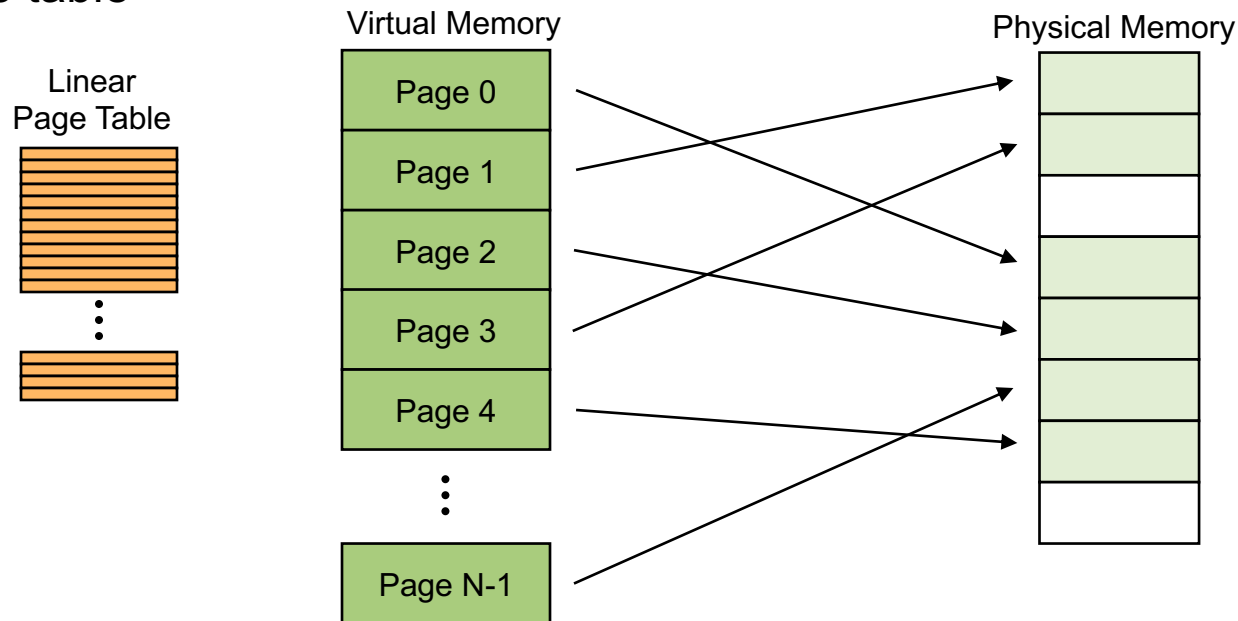
# Managing Page Tables

- How can we reduce the space required for page tables?
- **Observation: we only need to map the portions of the address space that are actually being used**
  - ♦ This may be a tiny fraction of the total address space
- How do we map only the regions that are in use?
  - ♦ We could dynamically extend the page table...
  - ♦ But this won't work if the address space is sparse (internal fragmentation)
- Use another level of indirection: **two-level page tables**



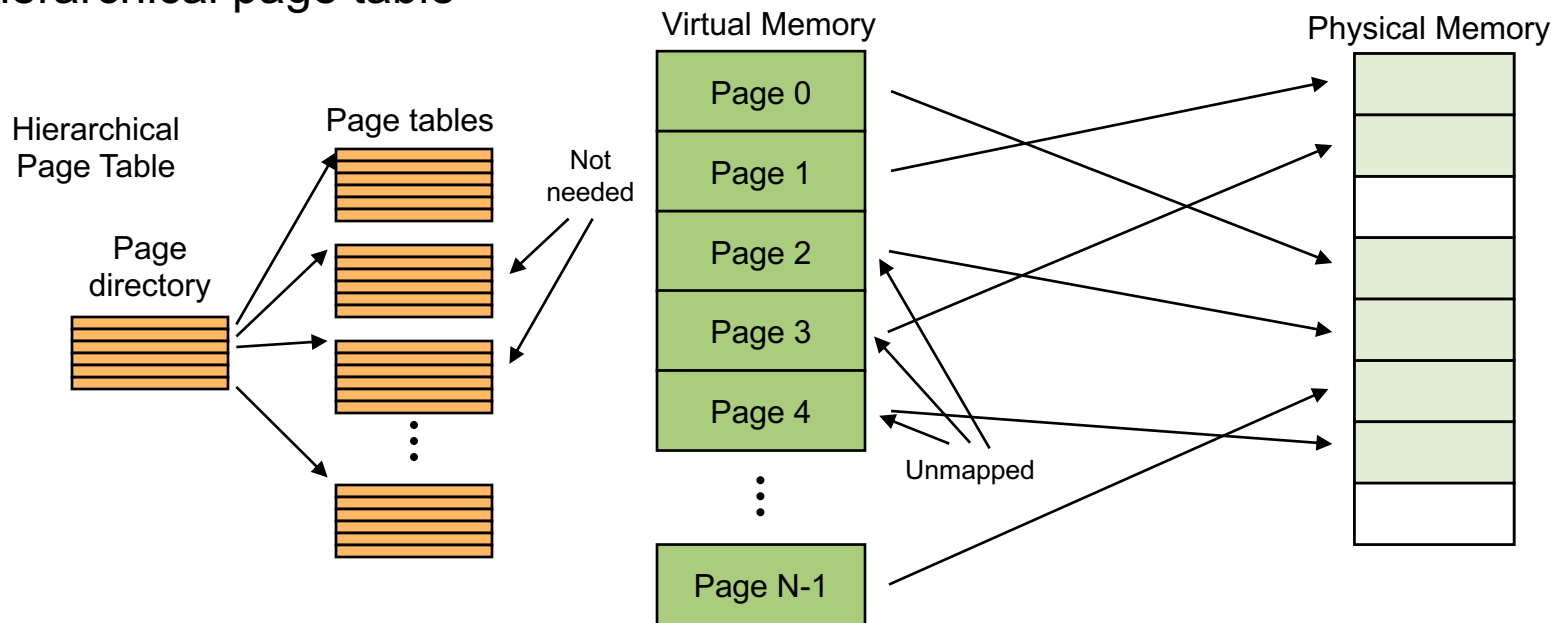
# Single-Level Page Tables

- Linear (flat) page table



# Two-Level Page Tables

- Hierarchical page table

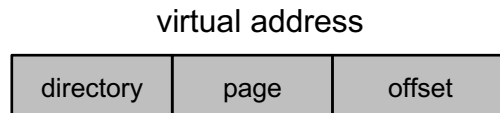




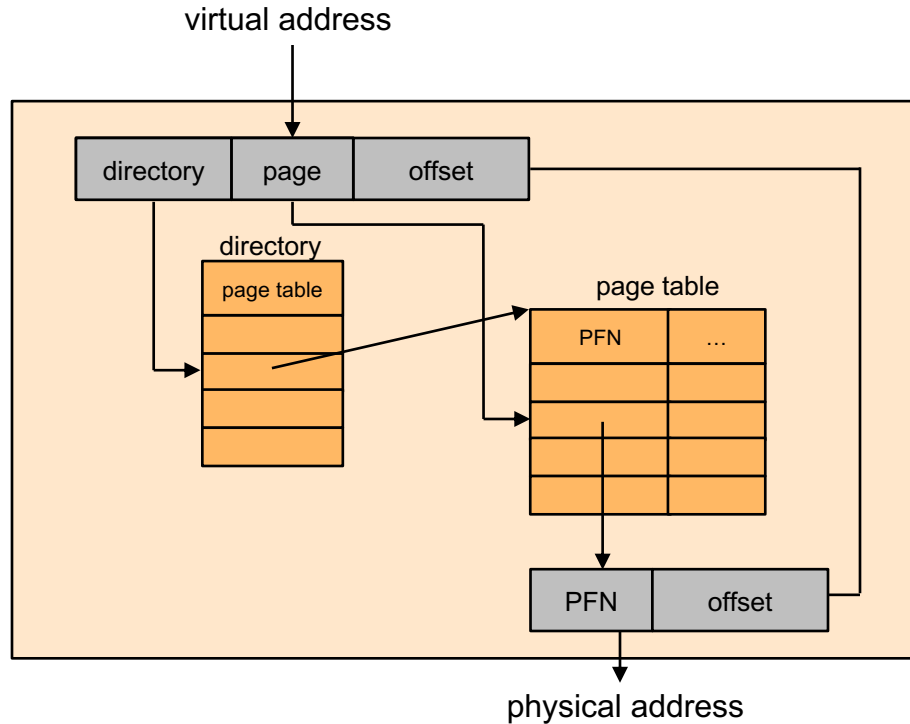
# Two-Level Page Tables

---

- Virtual addresses have three parts:
  - ♦ **Directory** (or root), **secondary page number**, **offset**
  - ♦ Directory page table maps virtual addresses to secondary page table
  - ♦ Secondary page table maps page number to physical page
  - ♦ Offset indicates where in physical page address is located

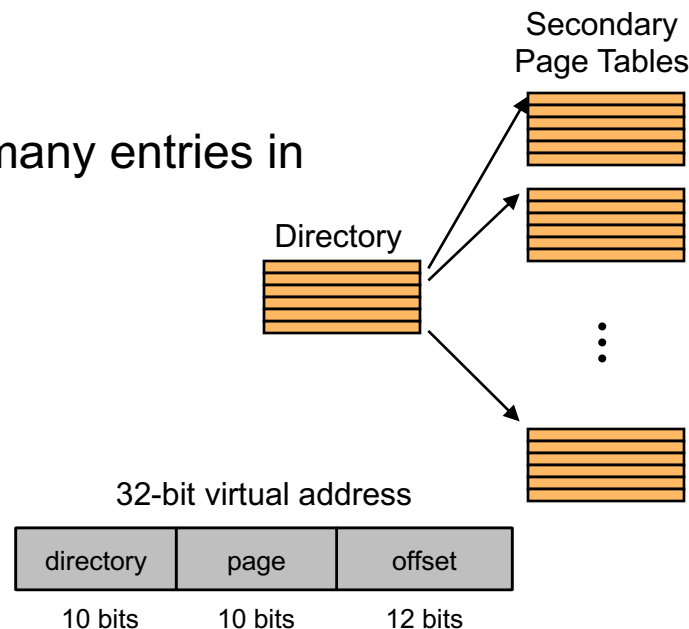


# Two-Level Page Tables – Address Translation



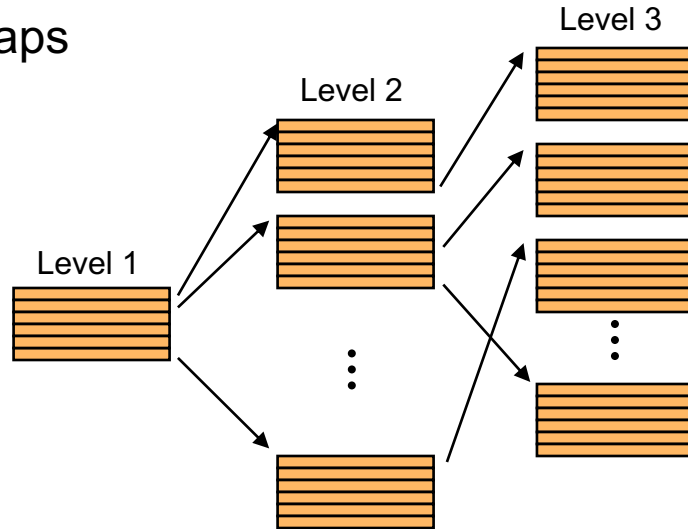
# Two-Level Page Tables - Example

- Assume 4 KB pages, 4 bytes per PTE, 32-bit address space
- How many bits in the offset?
  - ◆ 4 KB page =  $2^{12}$  so 12 bits
- Want directory page table in one page, how many entries in directory page table?
  - ◆ 4 KB / 4 bytes = 1024 entries
- How many bits in the directory index?
  - ◆ 1024 entries =  $2^{10}$  so 10 bits
- $32 - 12 - 10 = 10$  bits left for the page index
  - ◆  $2^{10}$  4-byte entries in each secondary page table

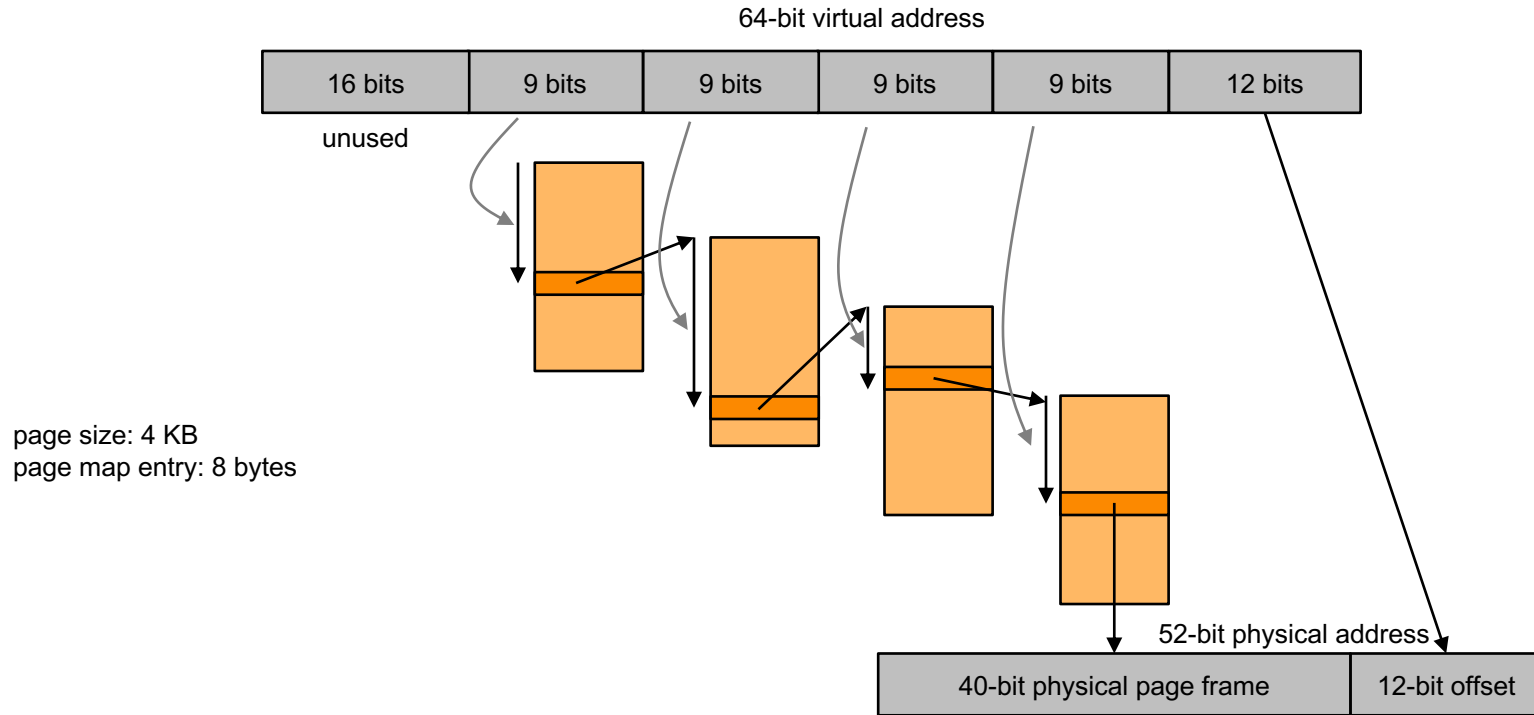


# Multi-Level Page Tables

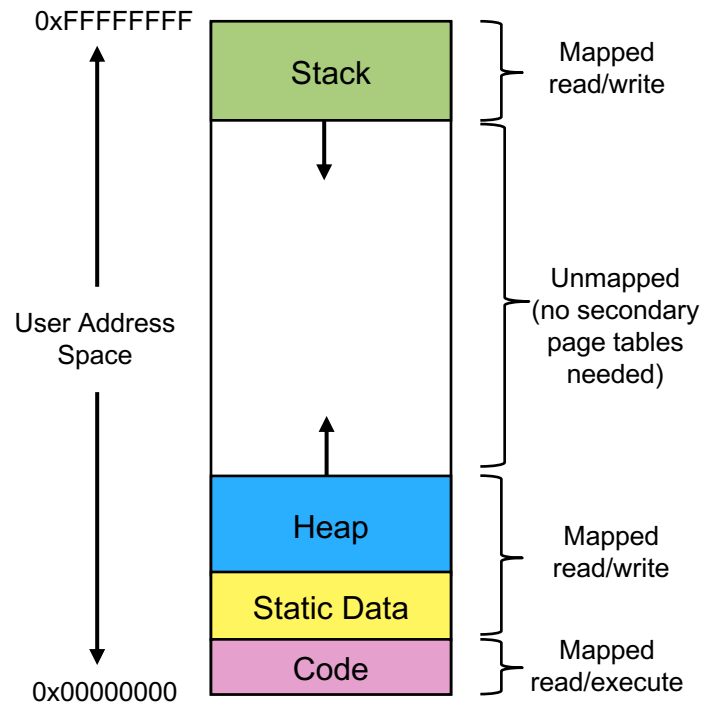
- We can generalize to many levels of page tables
- Each page map fits in one page
- Omit empty page maps



# x86-64 Address Translation



# Simple Address Space



# Today's Outline

---

- Paging Mechanisms
  - ◆ What information does a page table contain?
  - ◆ How can we represent page tables efficiently?
  - ◆ Page table management

# Storing Page Tables

---

- Where do we store page tables?
- Page tables are too large to store in the MMU
  - ◆ Store in memory instead
  - ◆ A special register holds the address of the top-level page map



# Addressing Page Tables

---

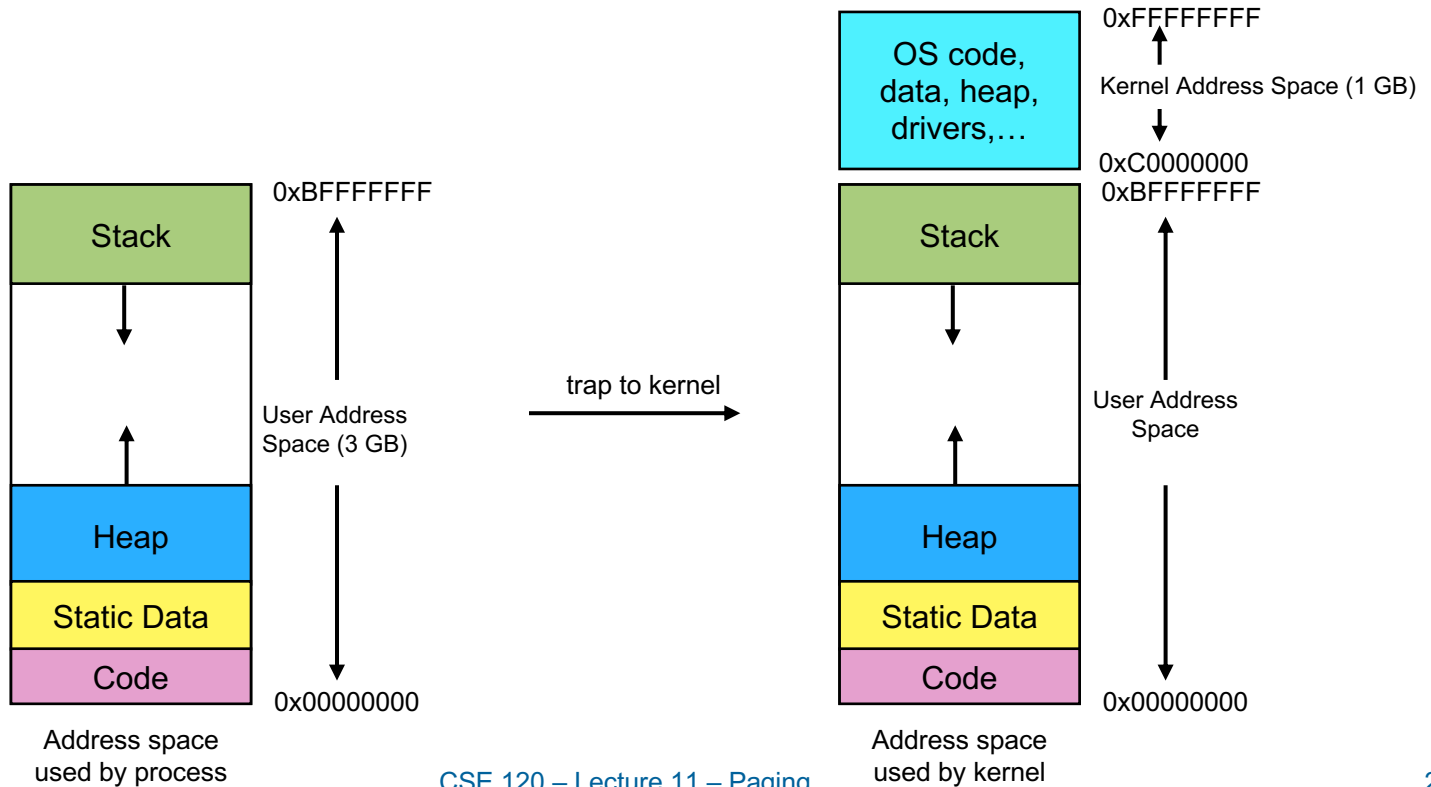
- Which address space should we store page tables in?
- Physical memory
  - ◆ Easy to address, no translation required
  - ◆ But, allocated page tables consume memory for the lifetime of the virtual address space
- Virtual memory (OS virtual address space)
  - ◆ Cold (unused) page table pages can be paged out to disk
  - ◆ But, addressing page tables requires translation!
  - ◆ How do we stop the recursion?
    - » Do not page the outermost page table

# Kernel Address Space

---

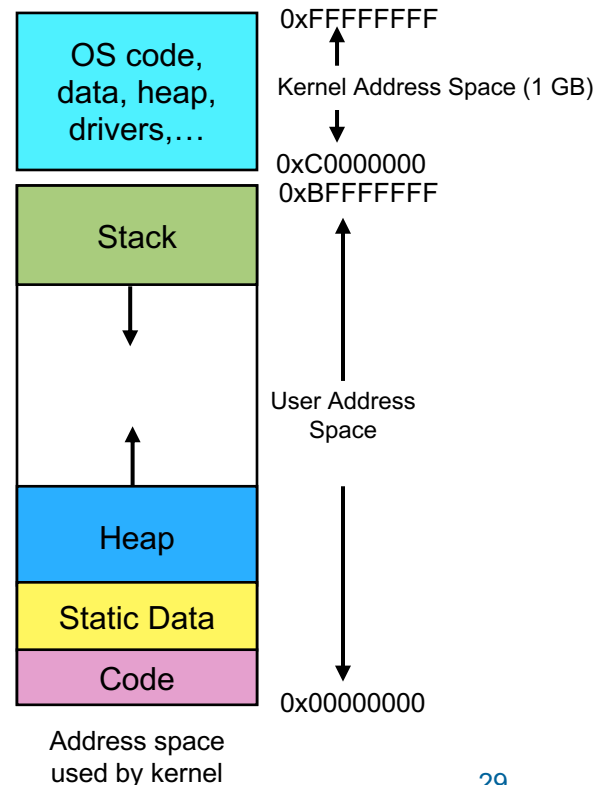
- How does the OS virtual address space work?
- The OS has its own separate address space
- Typically implemented as an extension of the user-level process address space
  - ◆ Bottom portion is for the user-level process
  - ◆ Top portion is for the OS/kernel
    - » Early Unix: user 2 GB, kernel 2 GB (32-bit)
    - » Linux, Windows: user 3 GB, kernel 1 GB (32-bit)

# Kernel Address Space



# Kernel Address Space

- In **user mode**, a process can only access the user-level portion
- In **kernel mode**, the OS can access the entire address space
- OS is mapped into every process
  - ◆ The upper portion of every process address space is the OS
  - ◆ Context switching switches the bottom portion
- Can use the same page table or an extended copy (KPTI)



# Paging Summary

---

- Paging Mechanisms
  - ♦ Page tables and page table entries (PTEs)
  - ♦ Multi-level page tables
  - ♦ Storing and addressing page tables

# More Memory Management

---

- Efficient translations
  - ♦ With multi-level page tables, many memory accesses in order to access one location in memory!
  - ♦ TLBs
- Efficient use of memory
  - ♦ What if our data doesn't fit in physical memory at once?
  - ♦ Demand-paged virtual memory
- Advanced functionality

# Two-Level Page Table Example

- Translate address 0x3F80, the 0<sup>th</sup> byte of VPN 254
  - ♦ b11 1111 1000 0000
- Assume 14-bit virtual address space
  - ♦ 8 bits for VPN, 6 bits for offset
- What is the physical address?

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figure 20.5: A Page Directory, And Pieces Of Page Table

From Operating Systems [Version 1.01], Figure 20.5





# For next class...

---

- Read chapter 21