



# Memory Management: Paging

*CS 571: Operating Systems (Spring 2020)*  
Lecture 7a

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpacı-Dusseau.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

# Review: Segmentation

# Virtual Memory Accesses

- Approaches:
  - Static Relocation
  - Dynamic Relocation
    - Base
    - Base-and-Bounds
  - Segmentation

# Virtual Memory Accesses

- Approaches:
  - **Static Relocation**: requires rewrite for the same code
  - **Dynamic Relocation**
    - **Base**: add a base to virtual address to get physical address
    - **Base-and-Bounds**: checks physical address is in range
  - **Segmentation**: many base+bounds pairs

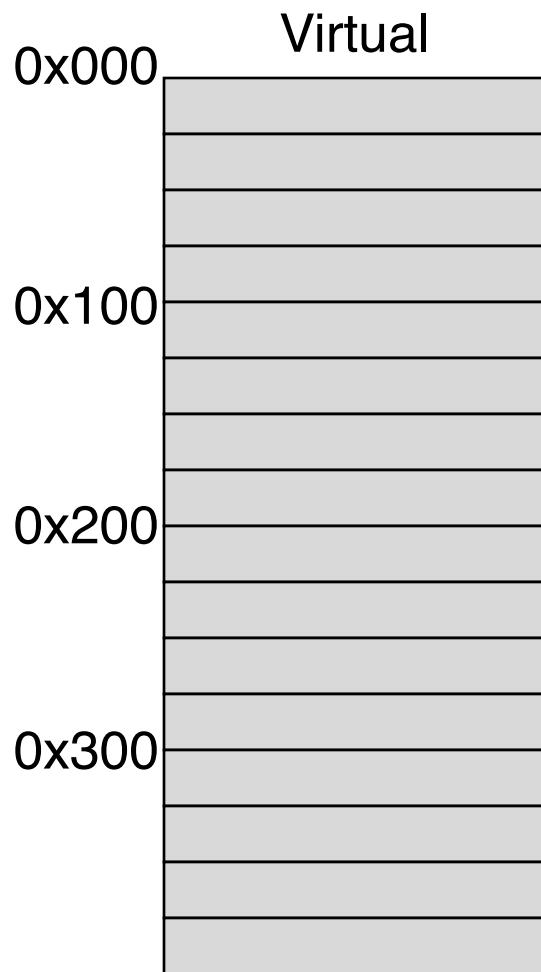
# Virtual Memory Accesses

- Approaches:
  - **Static Relocation**: requires rewrite for the same code
  - **Dynamic Relocation**
    - **Base**: add a base to virtual address to get physical address
    - **Base-and-Bounds**: checks physical address is in range
  - **Segmentation**: many base+bounds pairs

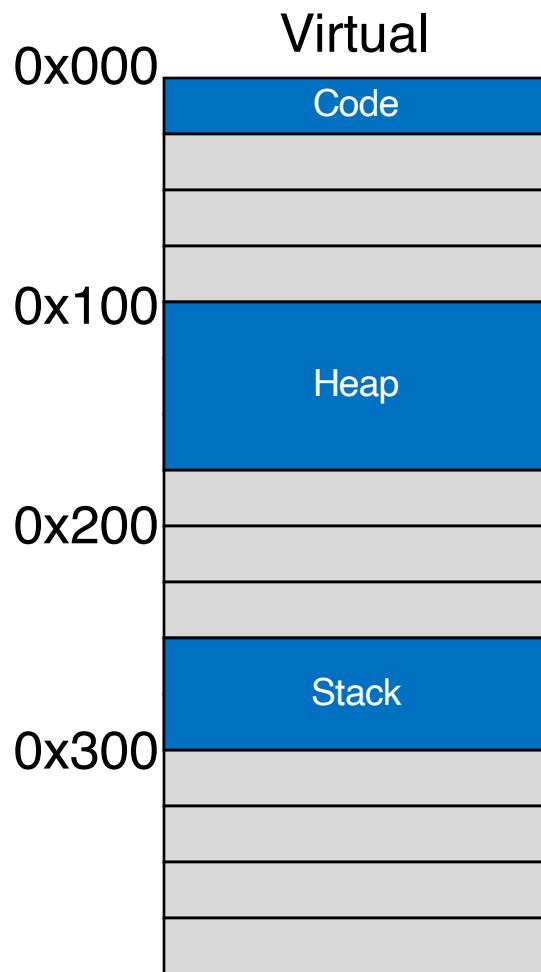
# Segmentation Example

- Assume a **10-bit** virtual address space
  - With the **high 2-bit** indicating the segment
- Assume
  - 0 => code+data
  - 1 => heap
  - 2 => stack

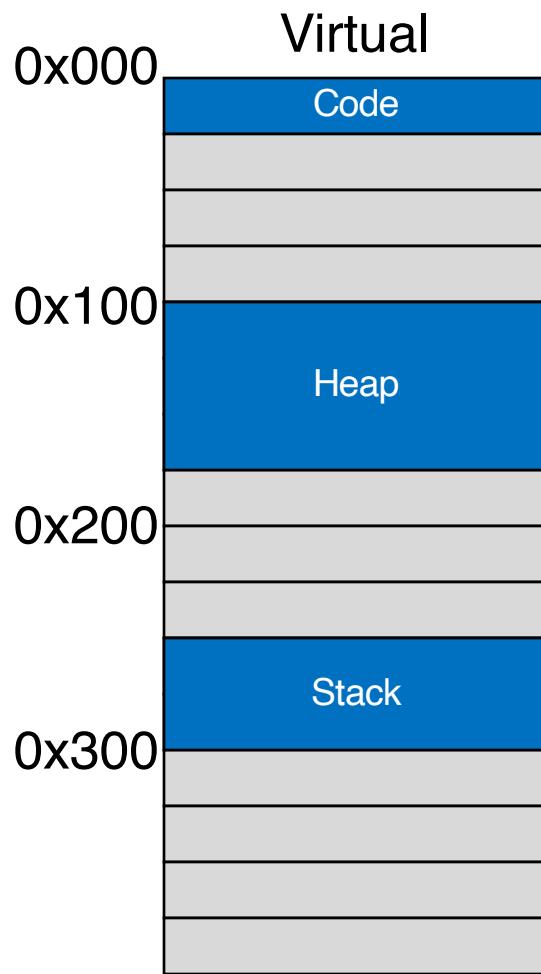
# Segmentation Example



# Segmentation Example

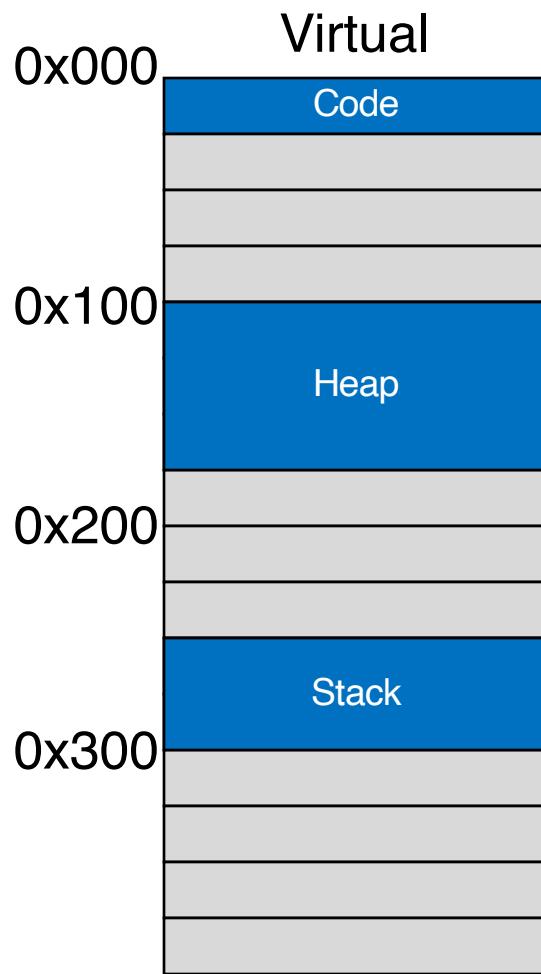


# Segmentation Example



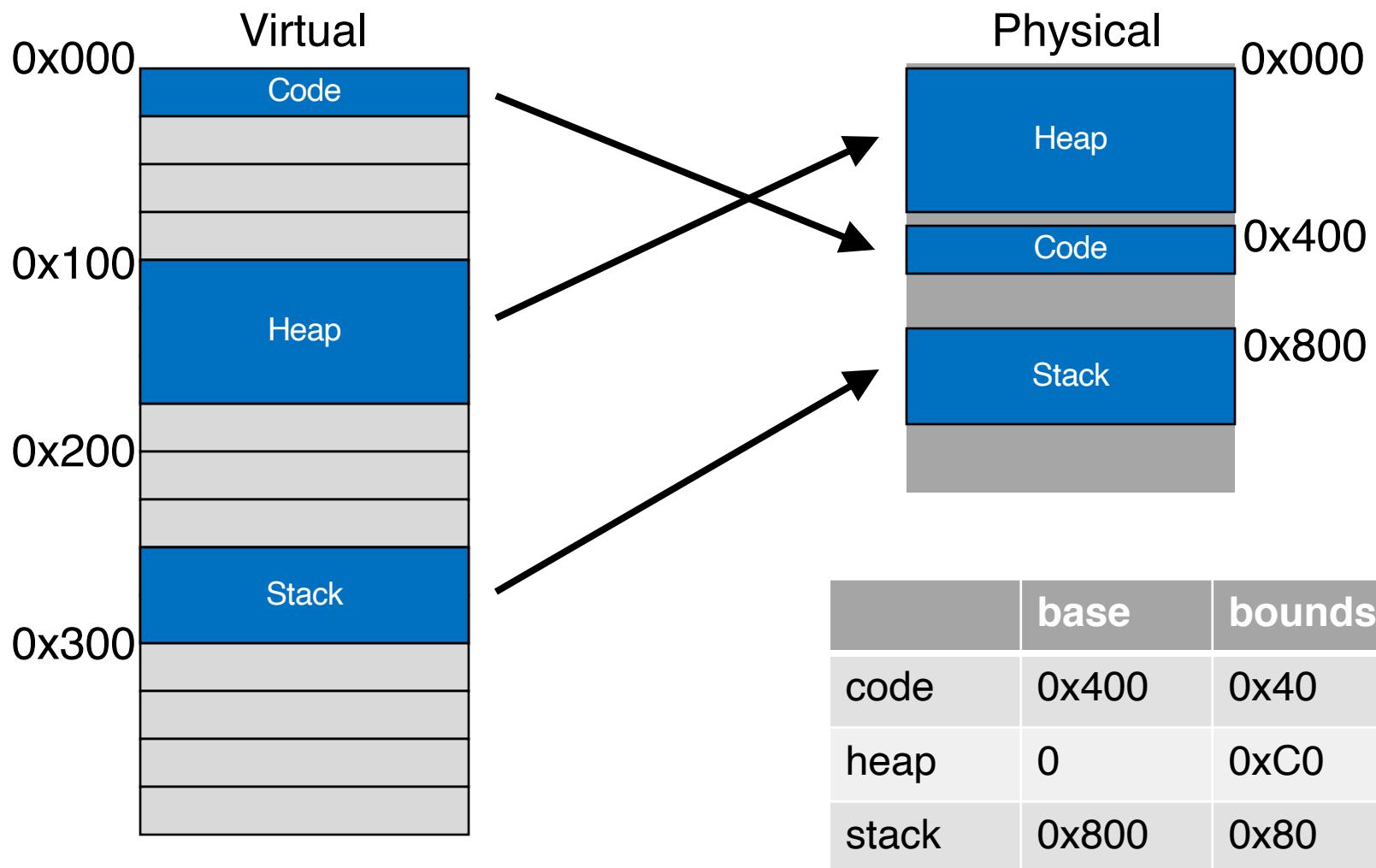
	<b>base</b>	<b>bounds</b>
code	?	?
heap	?	?
stack	?	?

# Segmentation Example

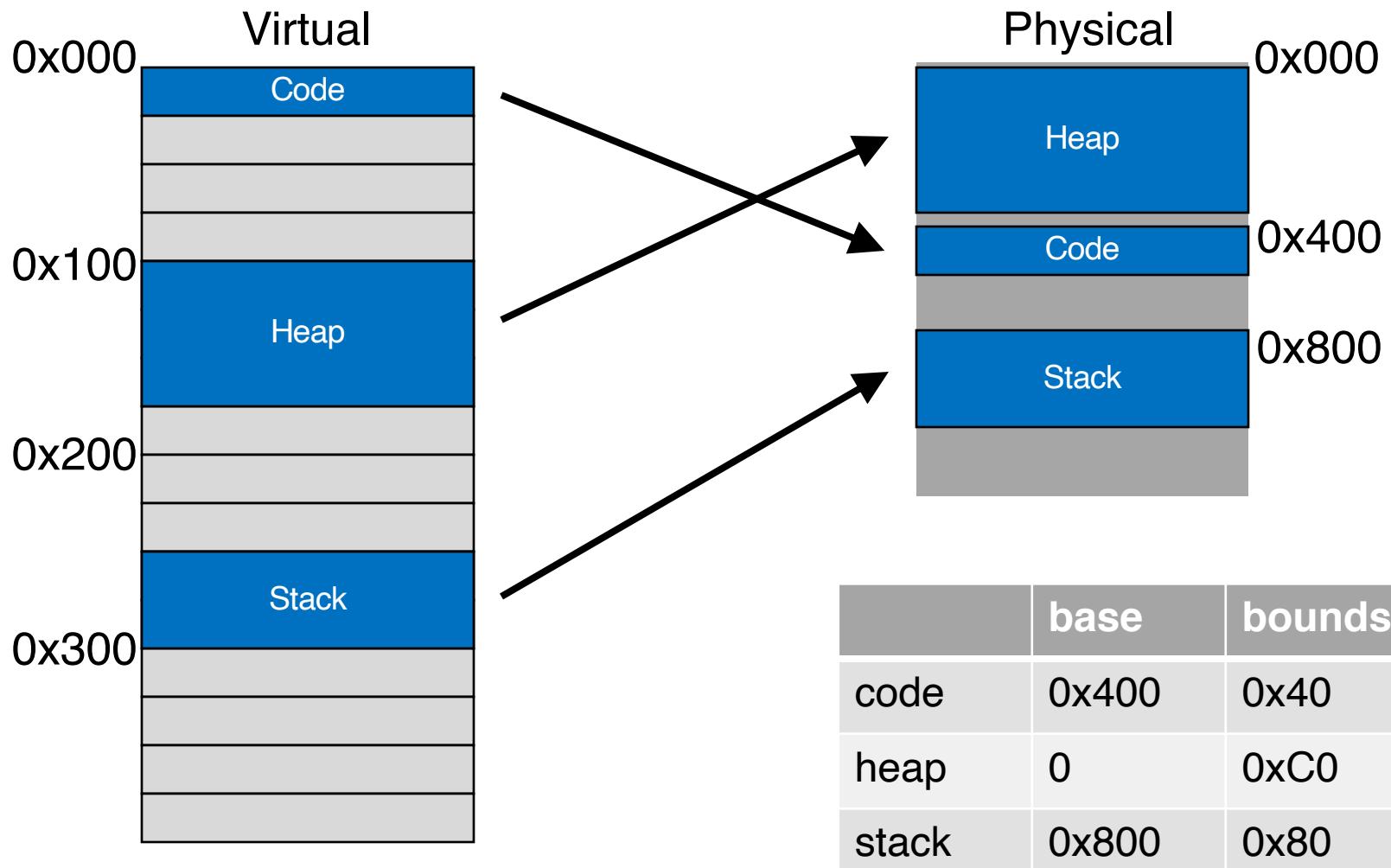


	<b>base</b>	<b>bounds</b>
code	?	0x40
heap	?	0xC0
stack	?	0x80

# Segmentation Example

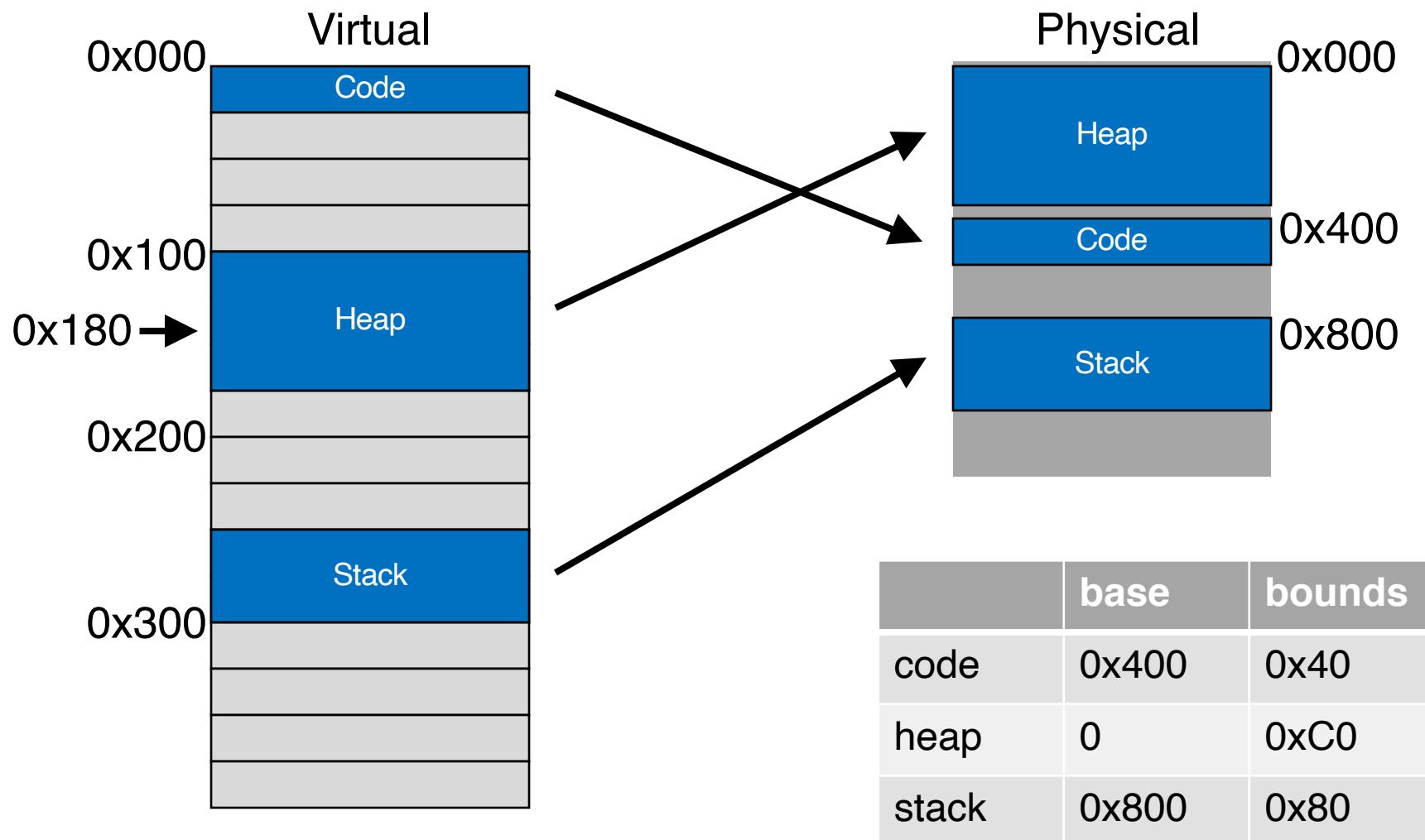


# Segmentation Example



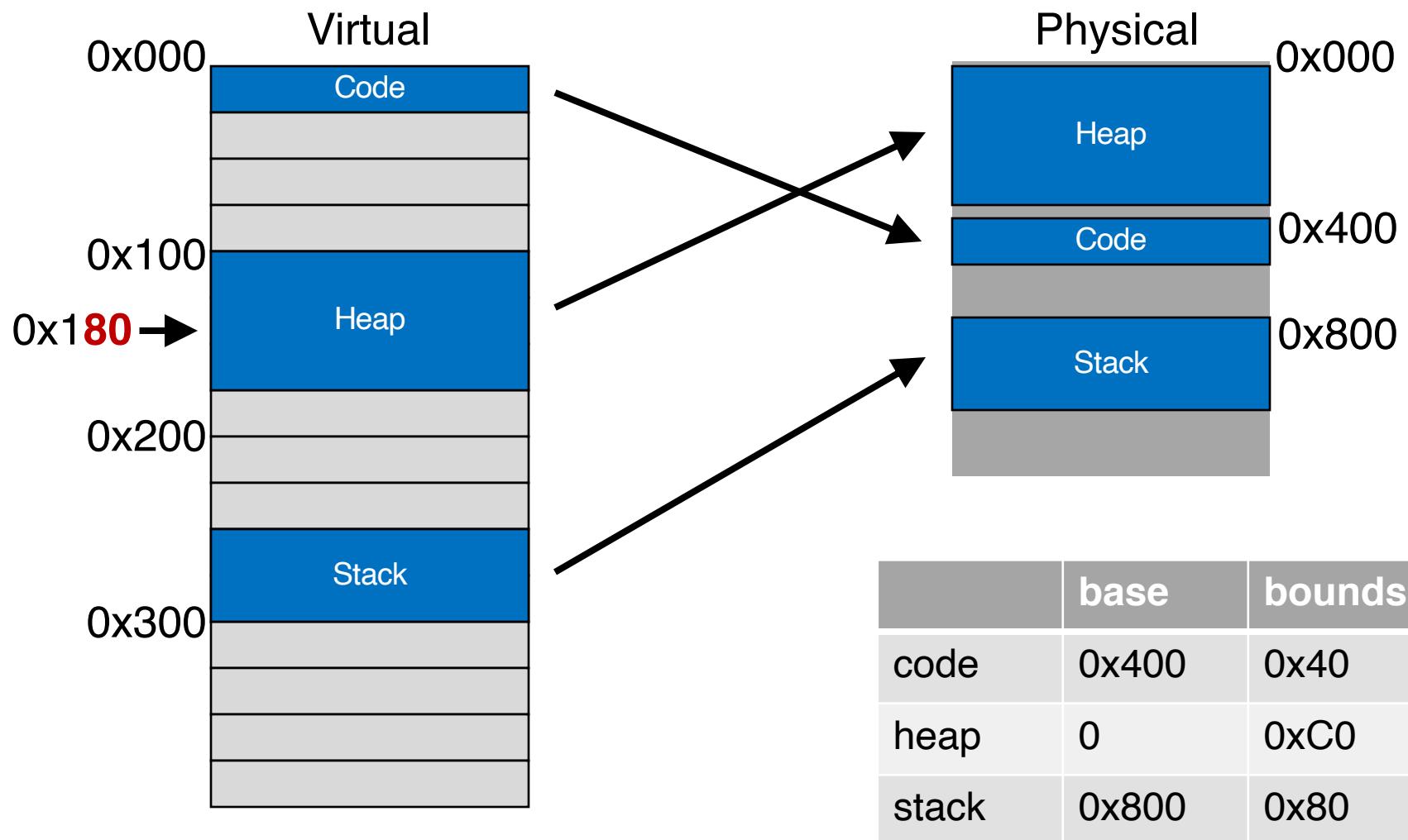
Most segments: `phys = virt_offset + base`

# Segmentation Example



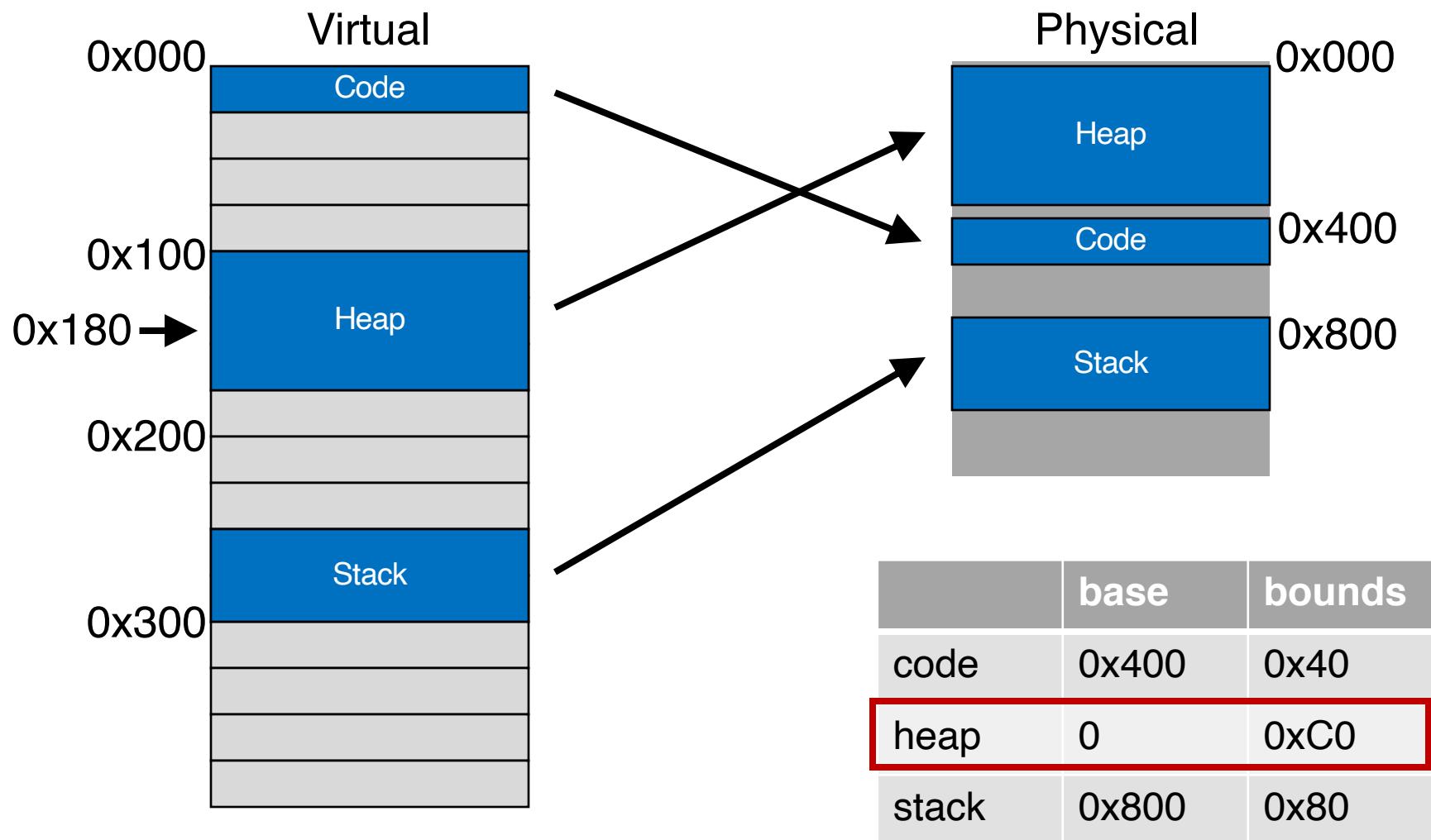
Most segments: `phys = virt_offset + base`

# Segmentation Example



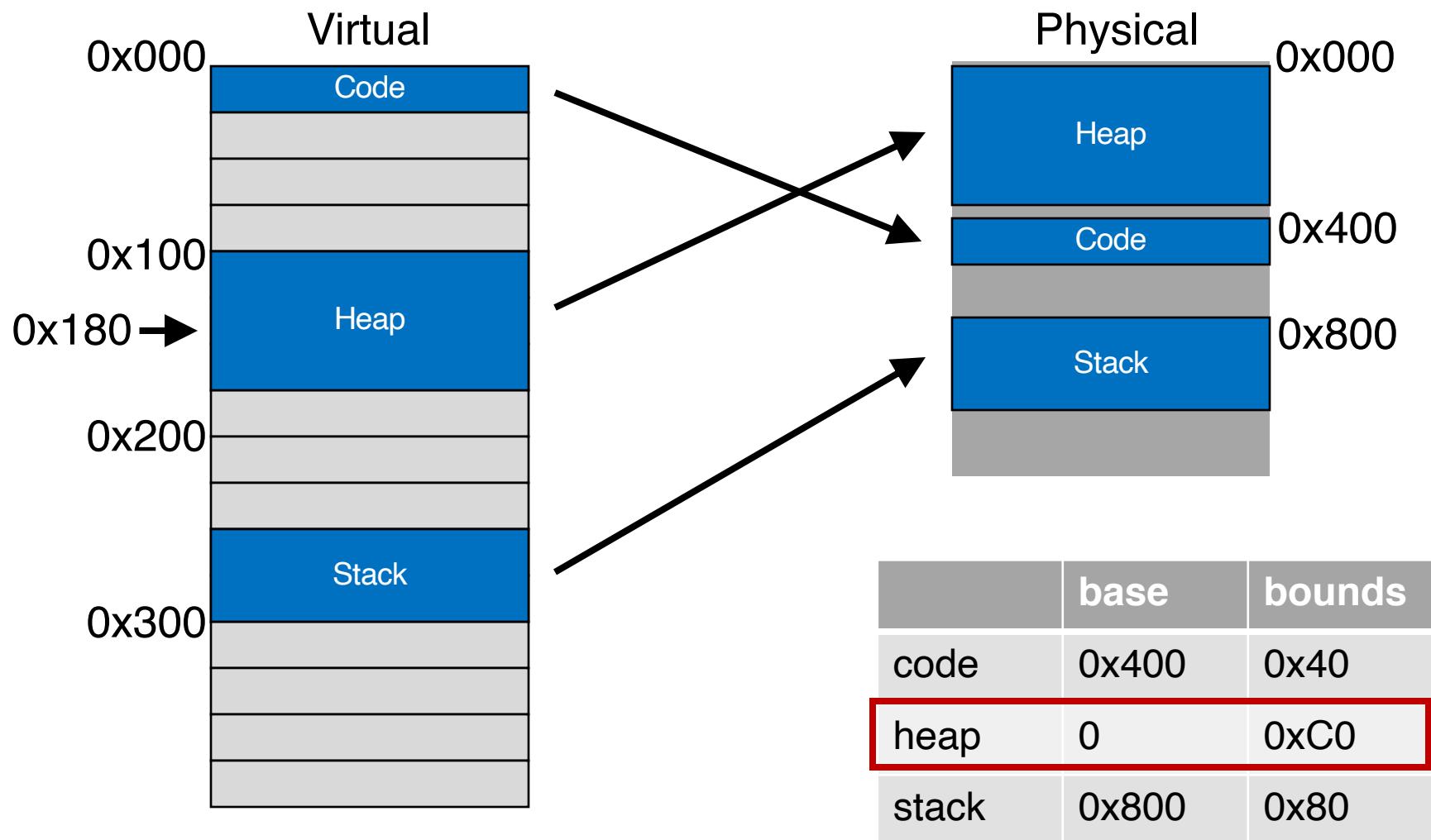
Most segments:  $\text{phys} = 0x80 + \text{base}$

# Segmentation Example



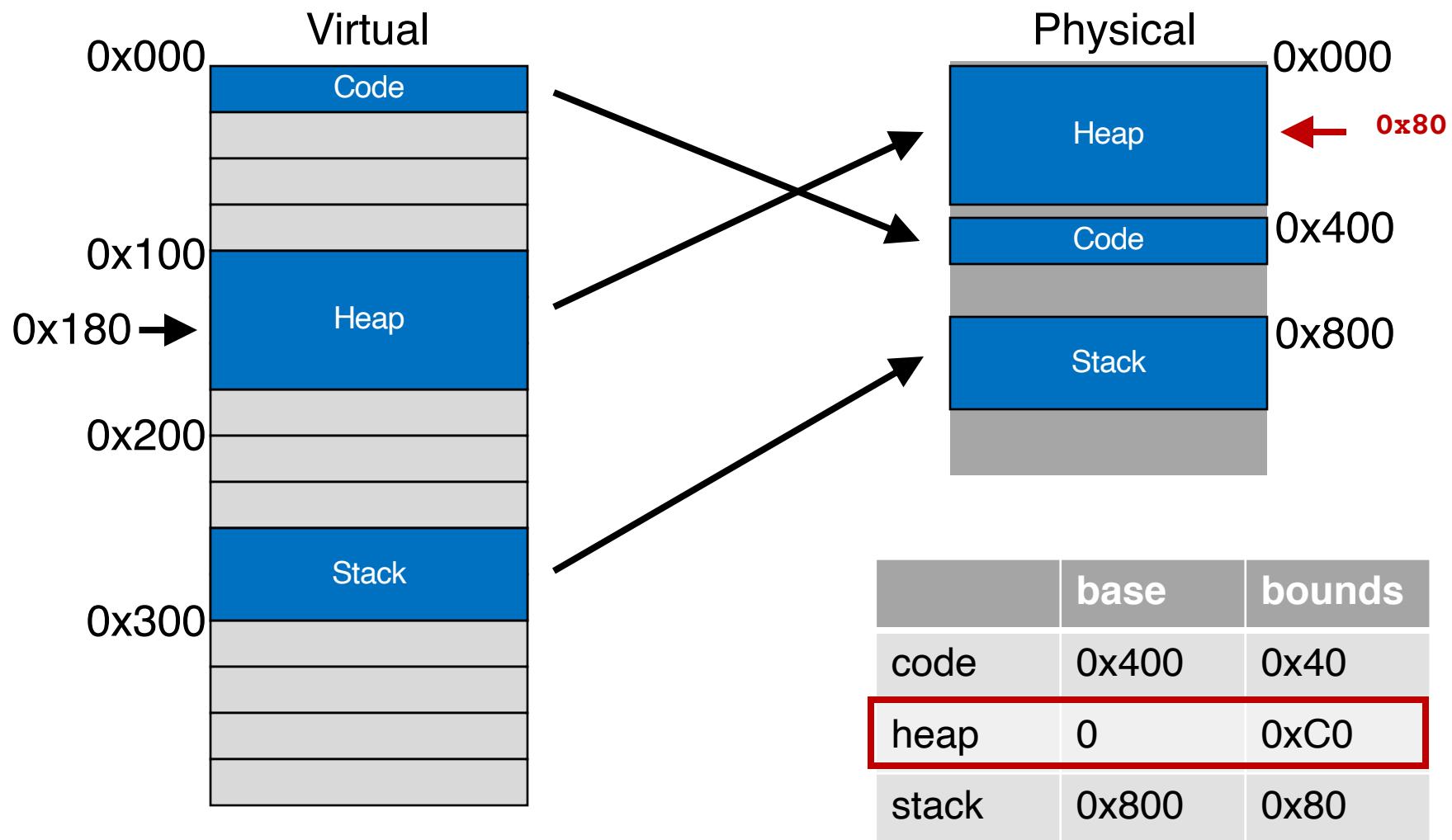
Most segments:  $\text{phys} = 0x80 + \text{base}$

# Segmentation Example



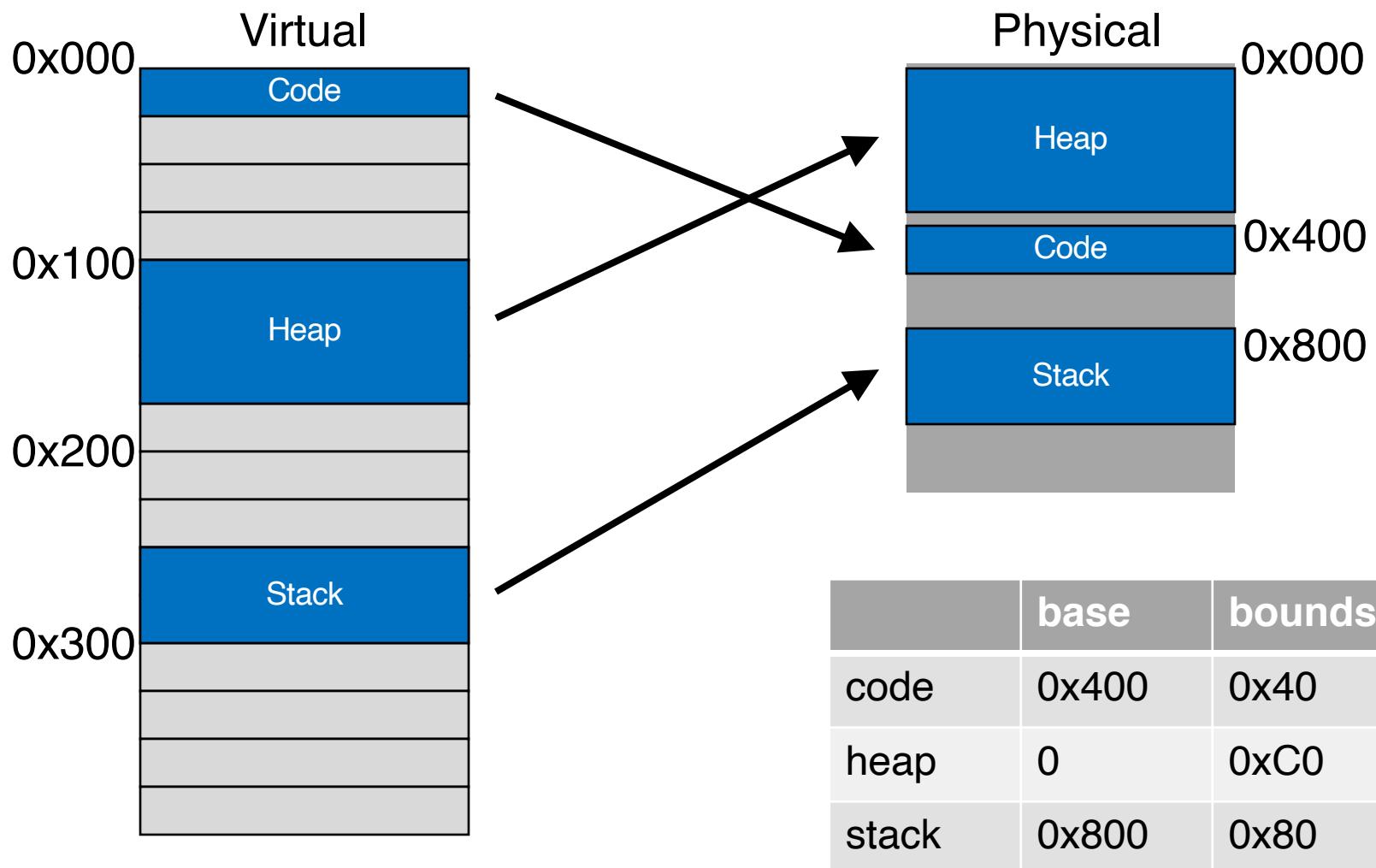
Most segments:  $\text{phys} = 0x80 + 0$

# Segmentation Example



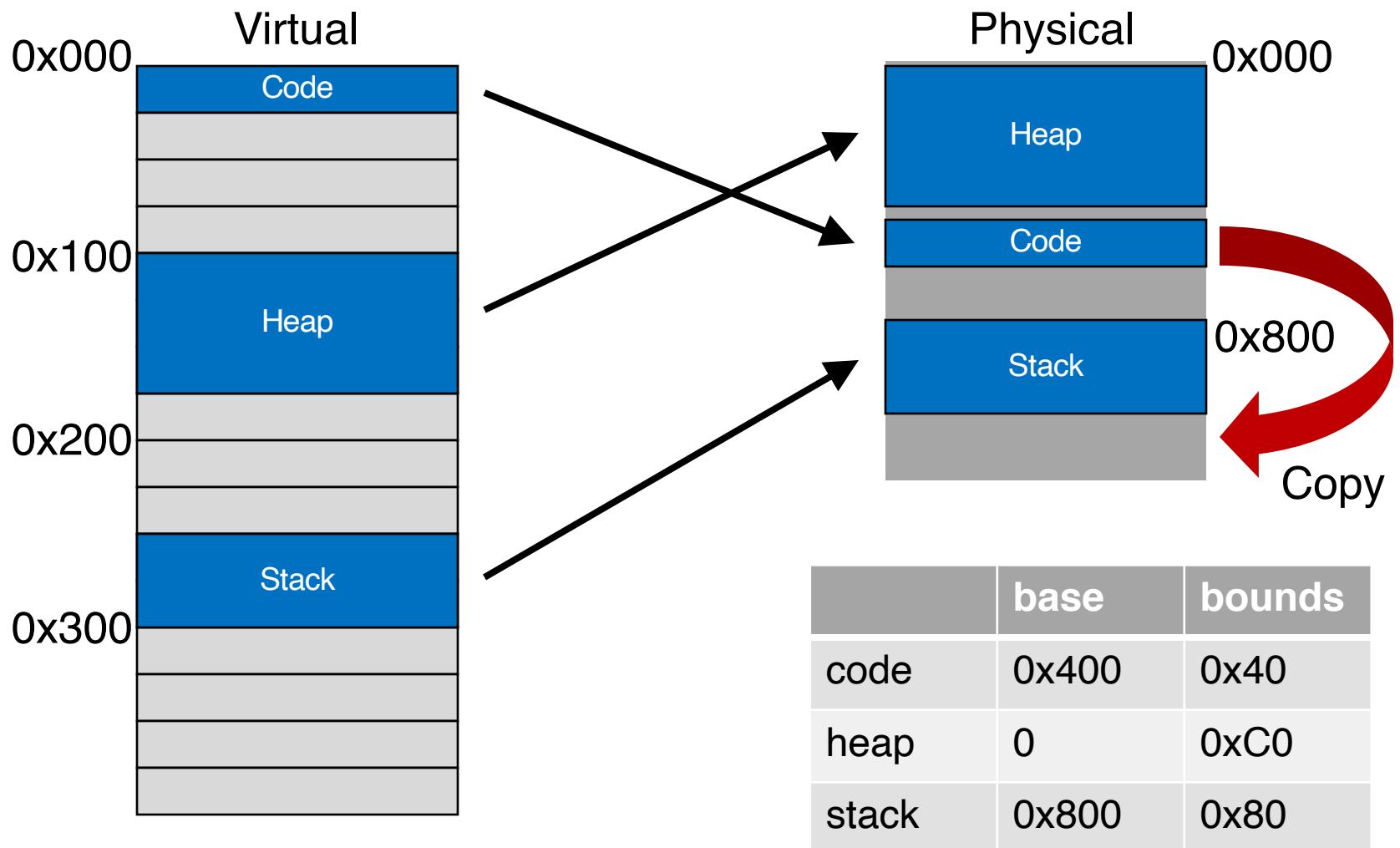
Most segments:  $\text{phys} = 0x80 + 0 = \text{0x80}$

# Segmentation Example



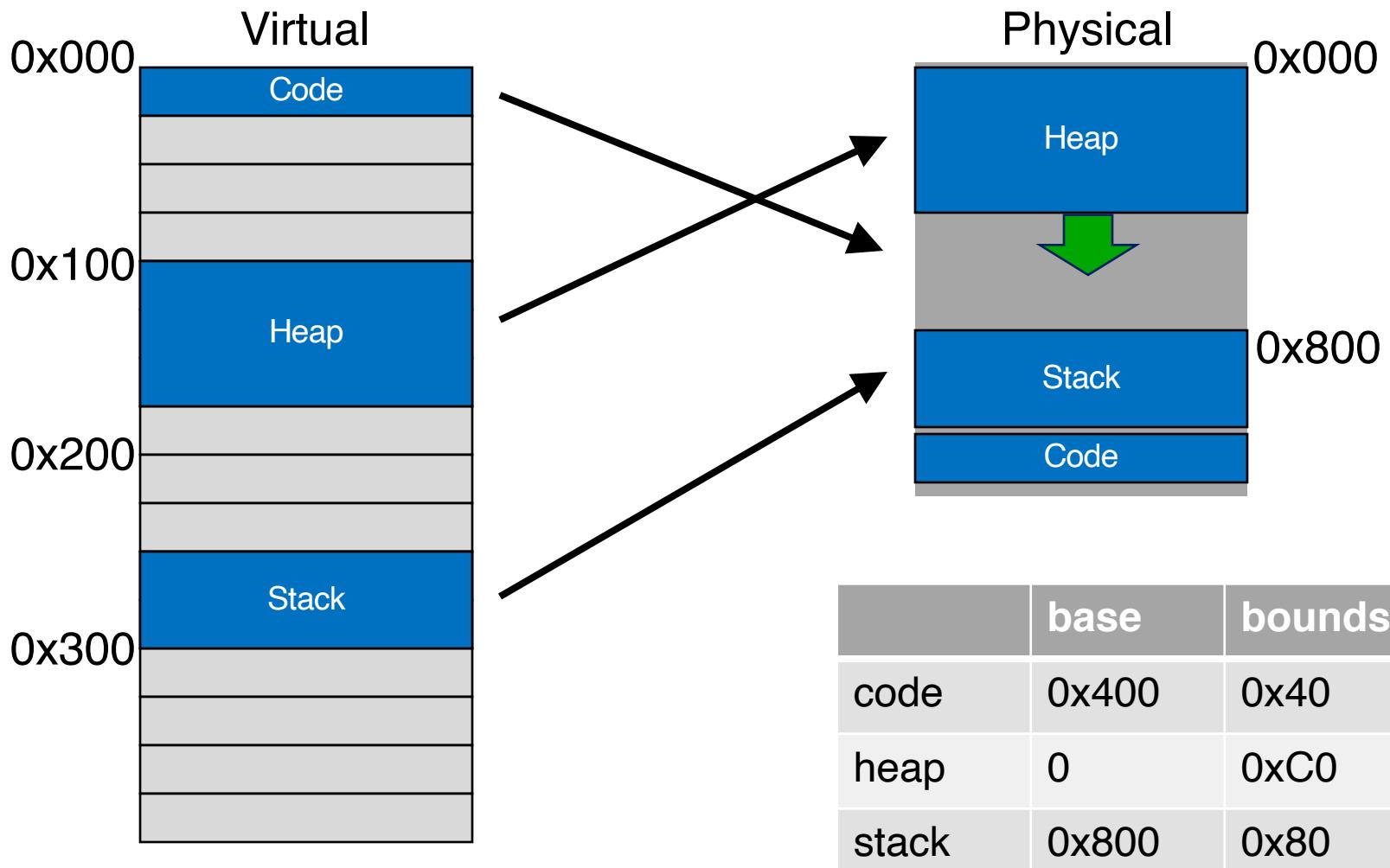
What if heap needs to grow?

# Segmentation Example



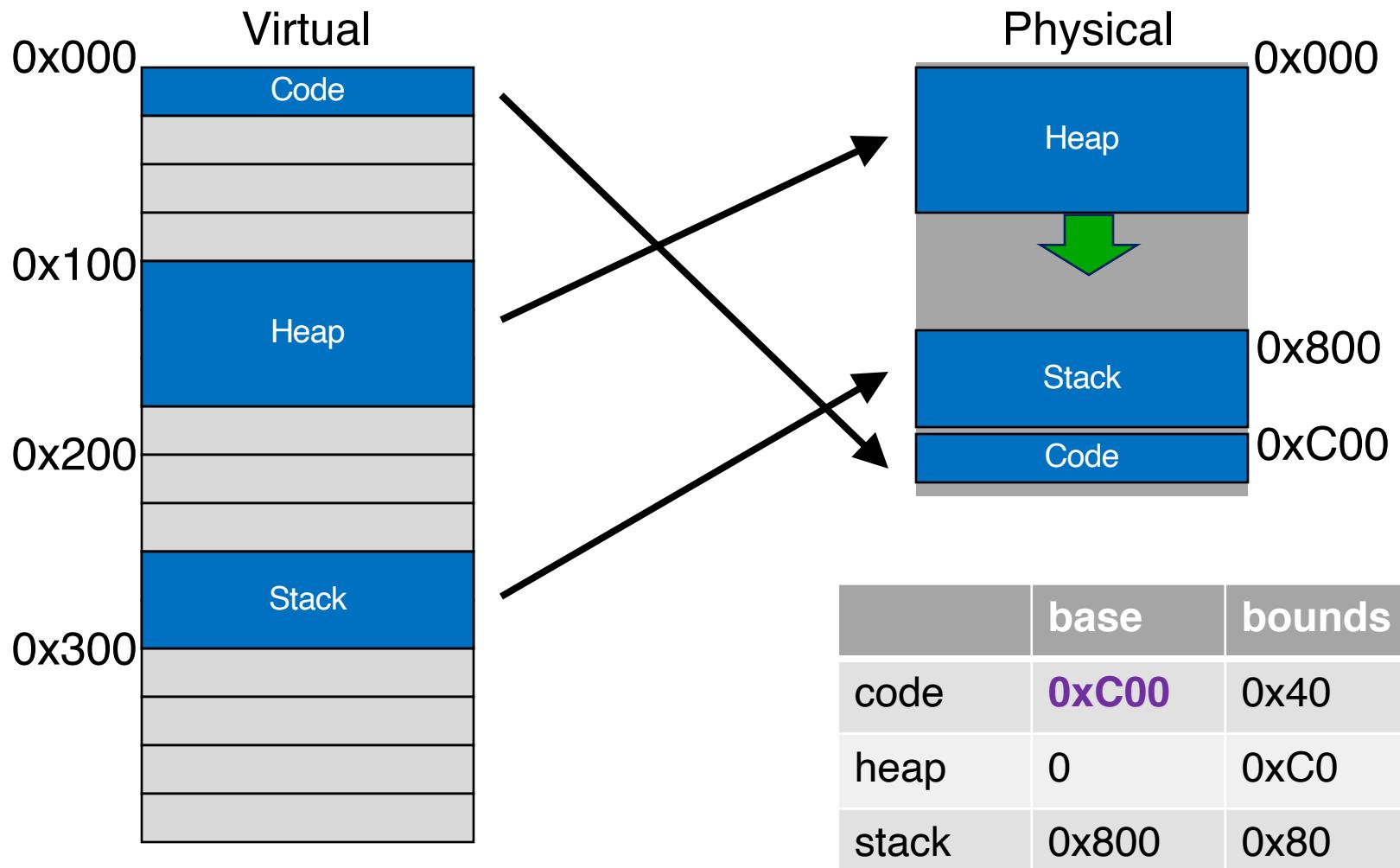
What if heap needs to grow?

# Segmentation Example



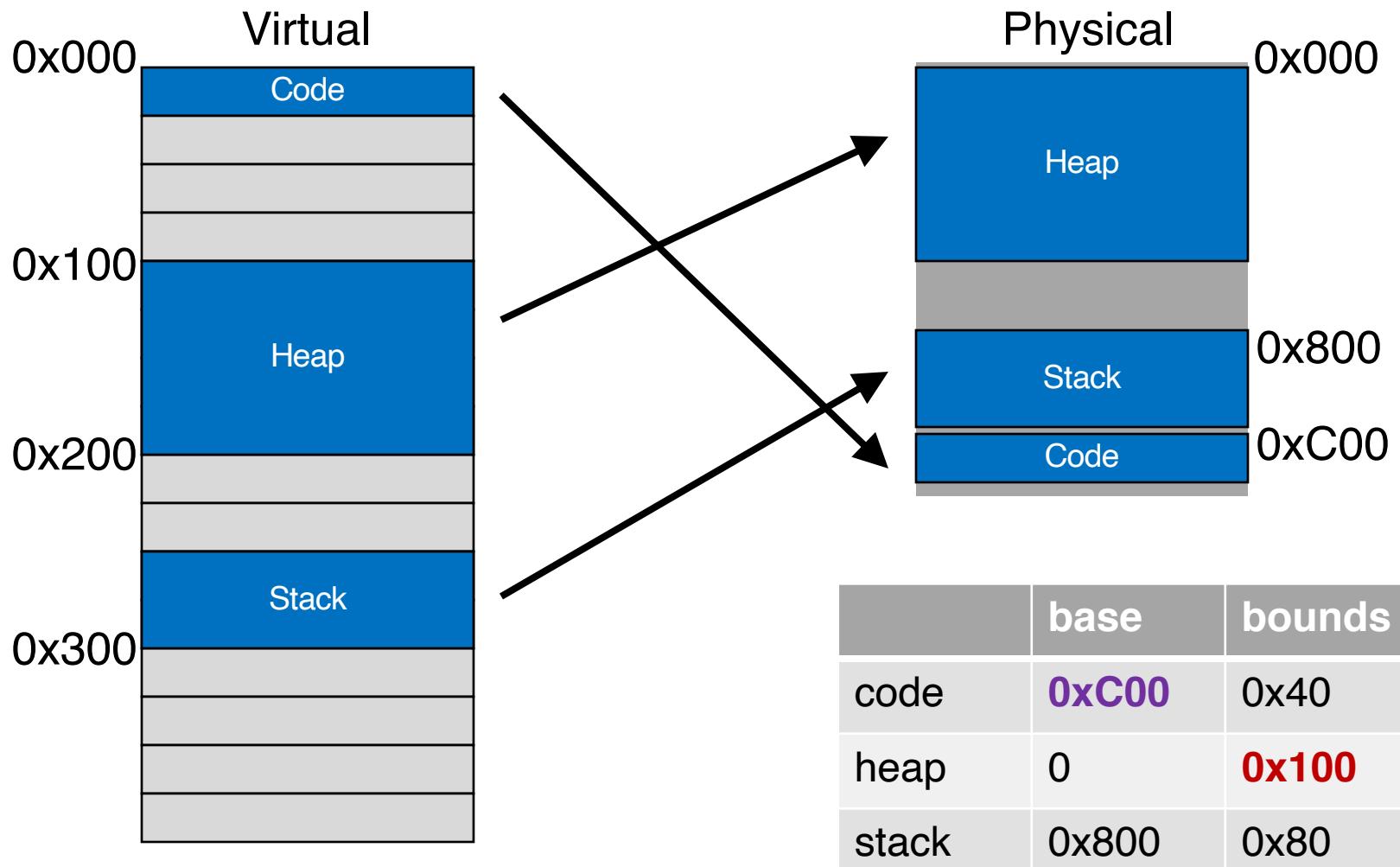
What if heap needs to grow?

# Segmentation Example



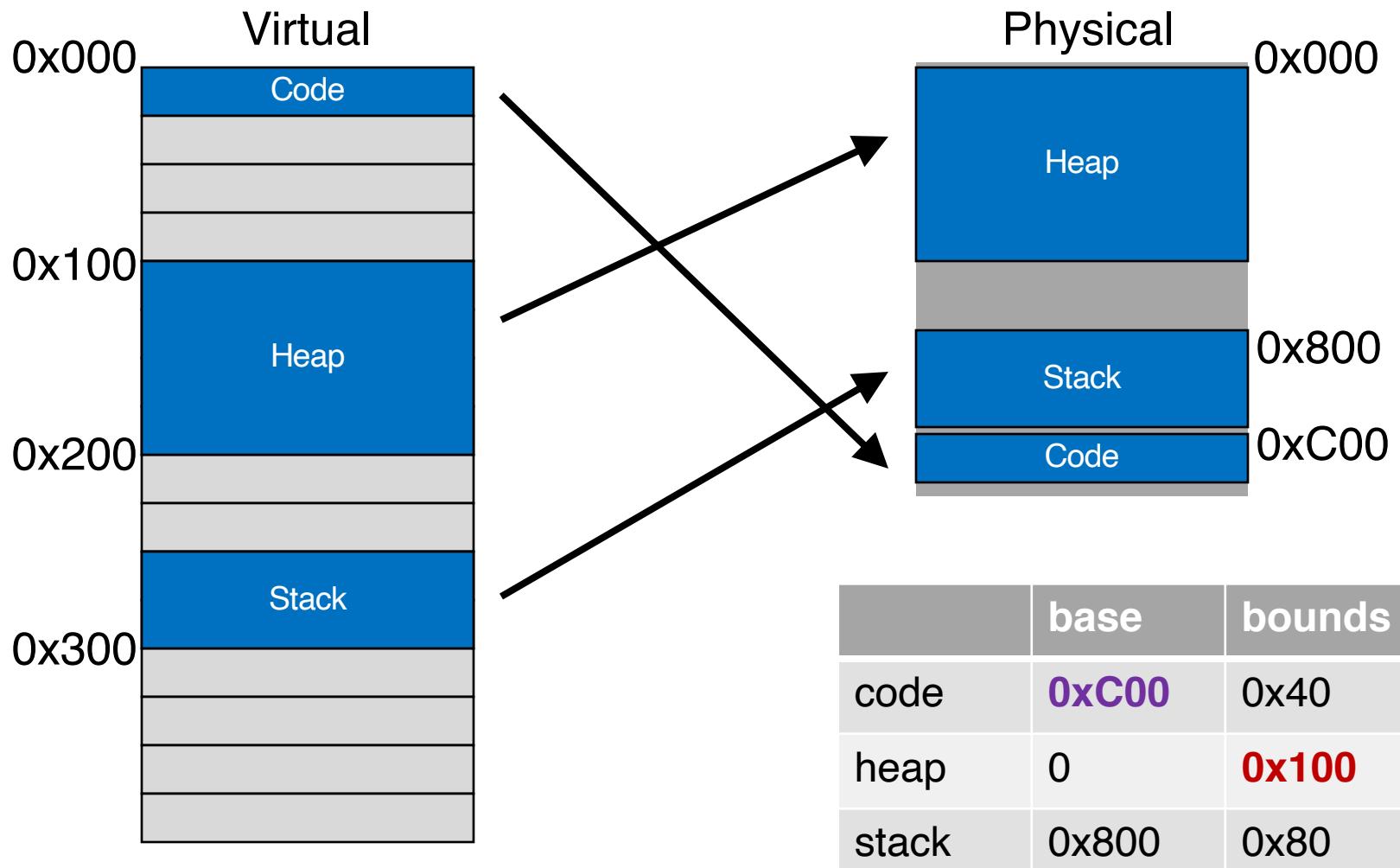
What if heap needs to grow?

# Segmentation Example



What if heap needs to grow?

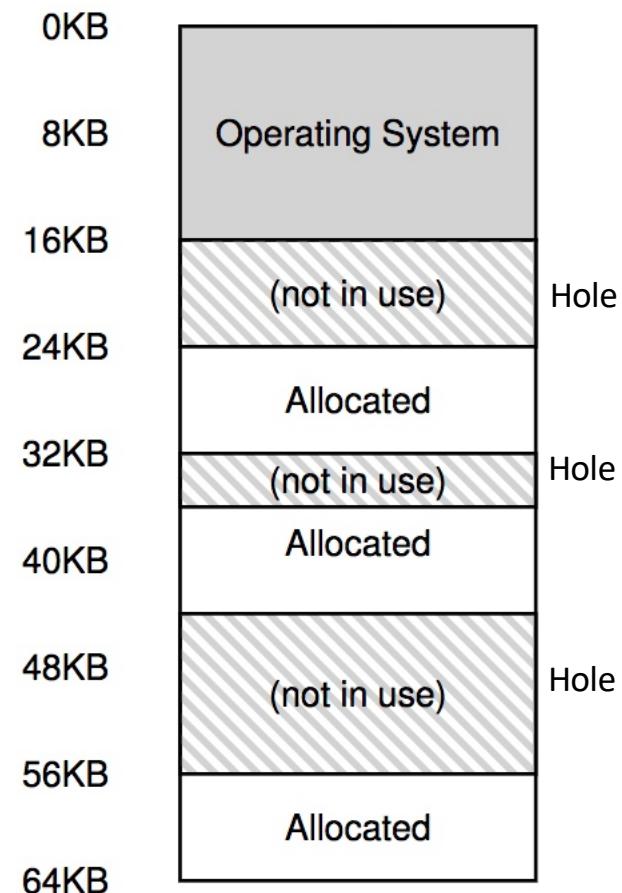
# Segmentation Example

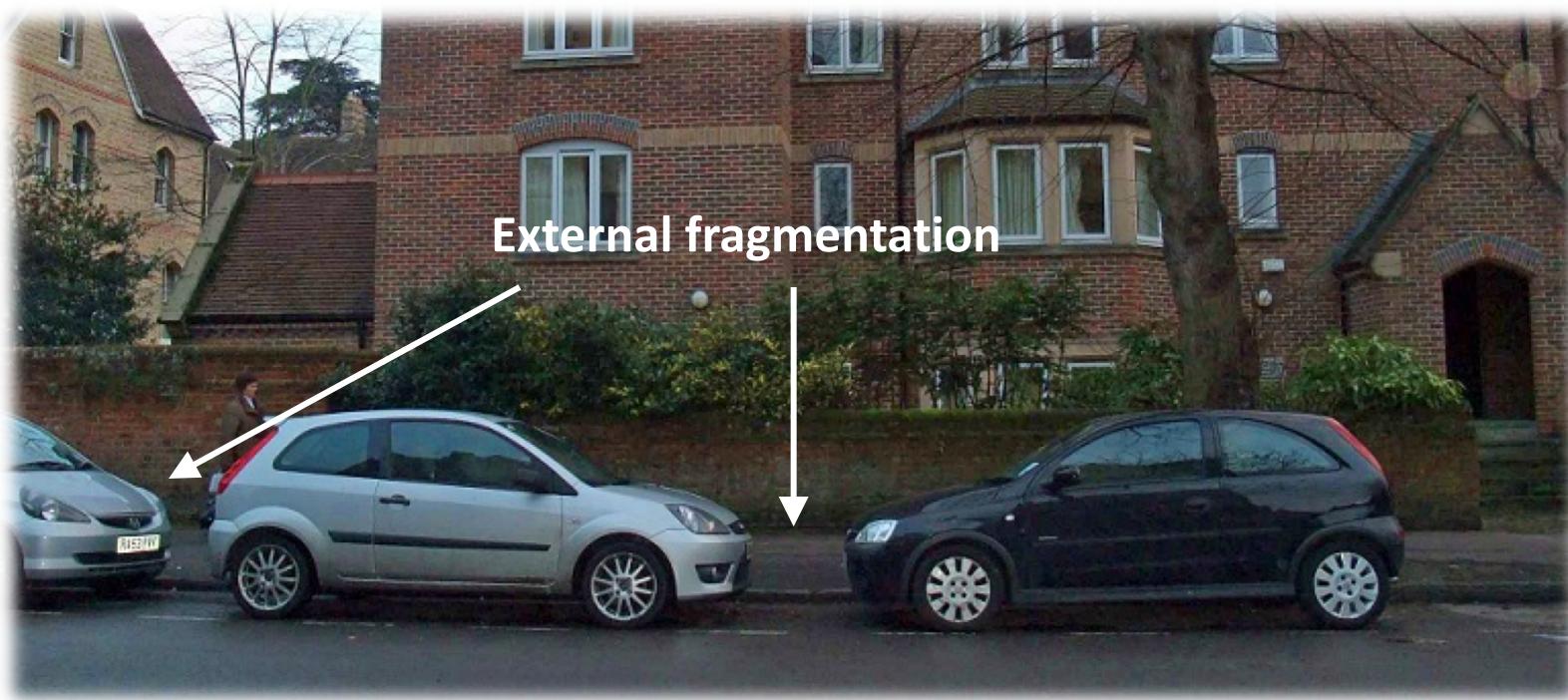


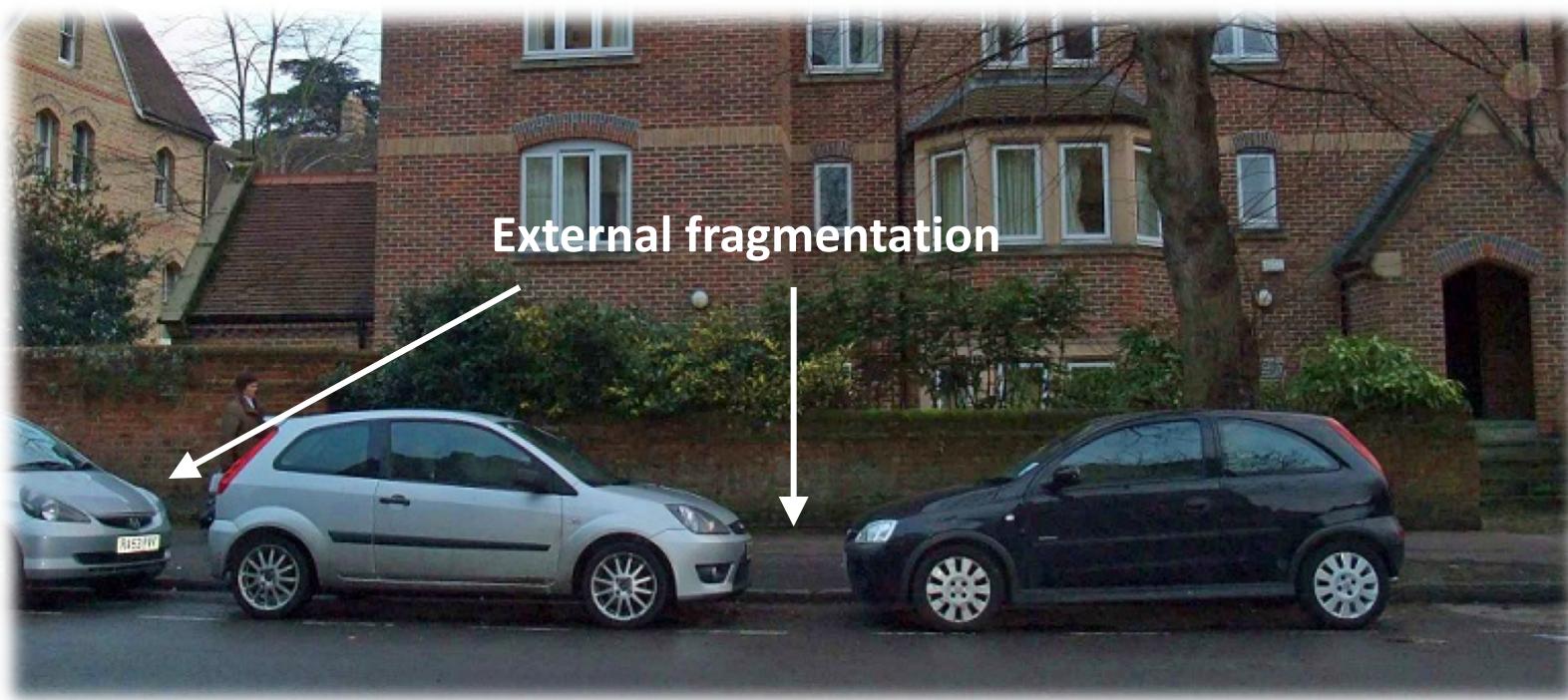
dilemma: must **(a) waste space or (b) waste time**

# Issues: External Fragmentation

- As processes are loaded and removed from the main memory, the free memory is broken into small pieces
  - Hole:** block of available memory; holes of various size are scattered throughout memory
- A new allocation request may have to be denied
  - When there is no contiguous free memory with requested size
  - The total free memory space may be much larger than the requested size!**



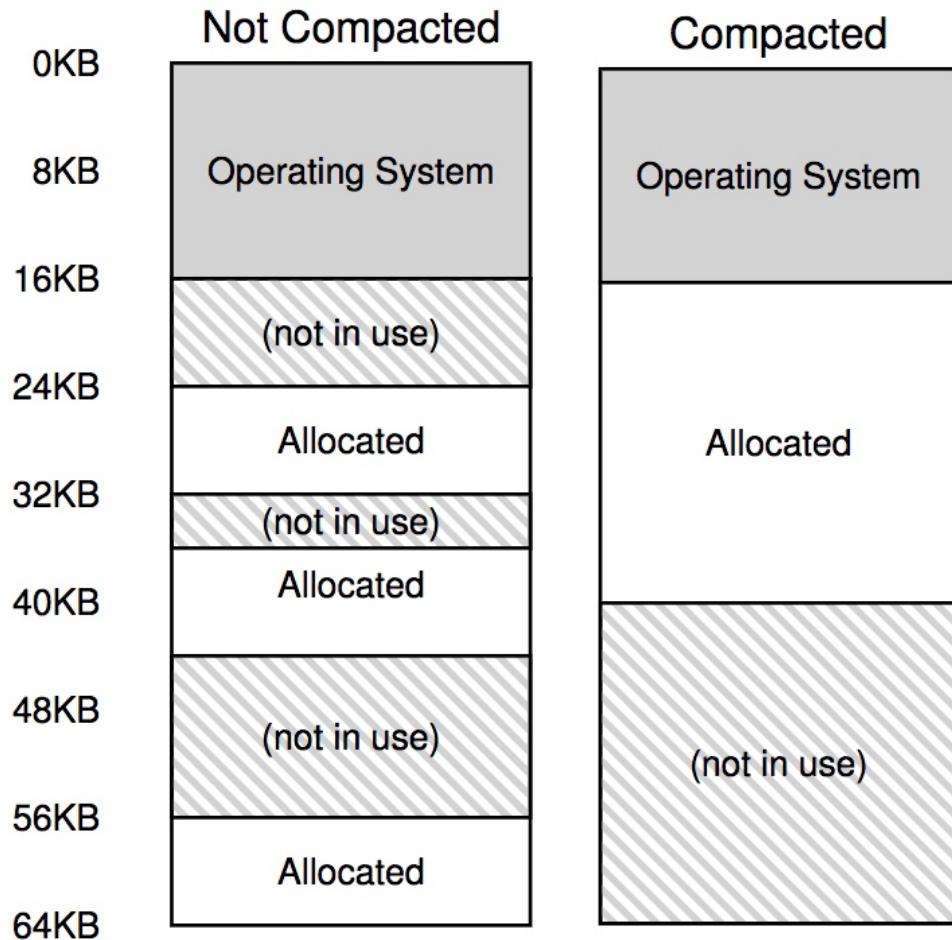




**Ideally, what we want...**

# Memory Compaction

- Reduce external fragmentation by **copy+compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only if relocation is dynamic, and is done at execution time*
  - Must be careful about pending I/O before initiating compaction
  - **Problems**
    - Too much perf overhead



# Paging

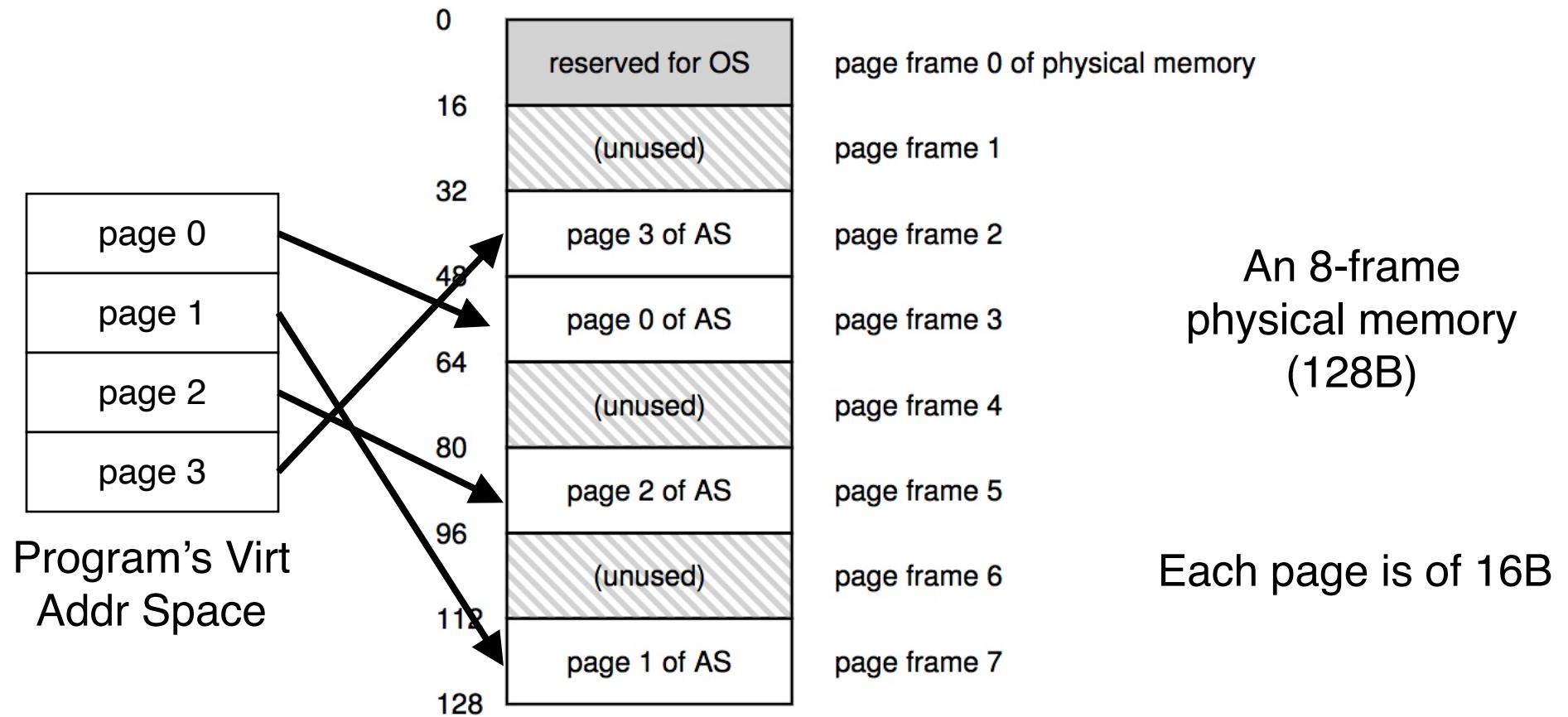
# Paging

- Motivation: Segmentation is too **coarse-grained**
  - Either waste space (**external fragmentation**) or
  - copy memory often (**compaction**)
- We need a **finer-grained** alternative!

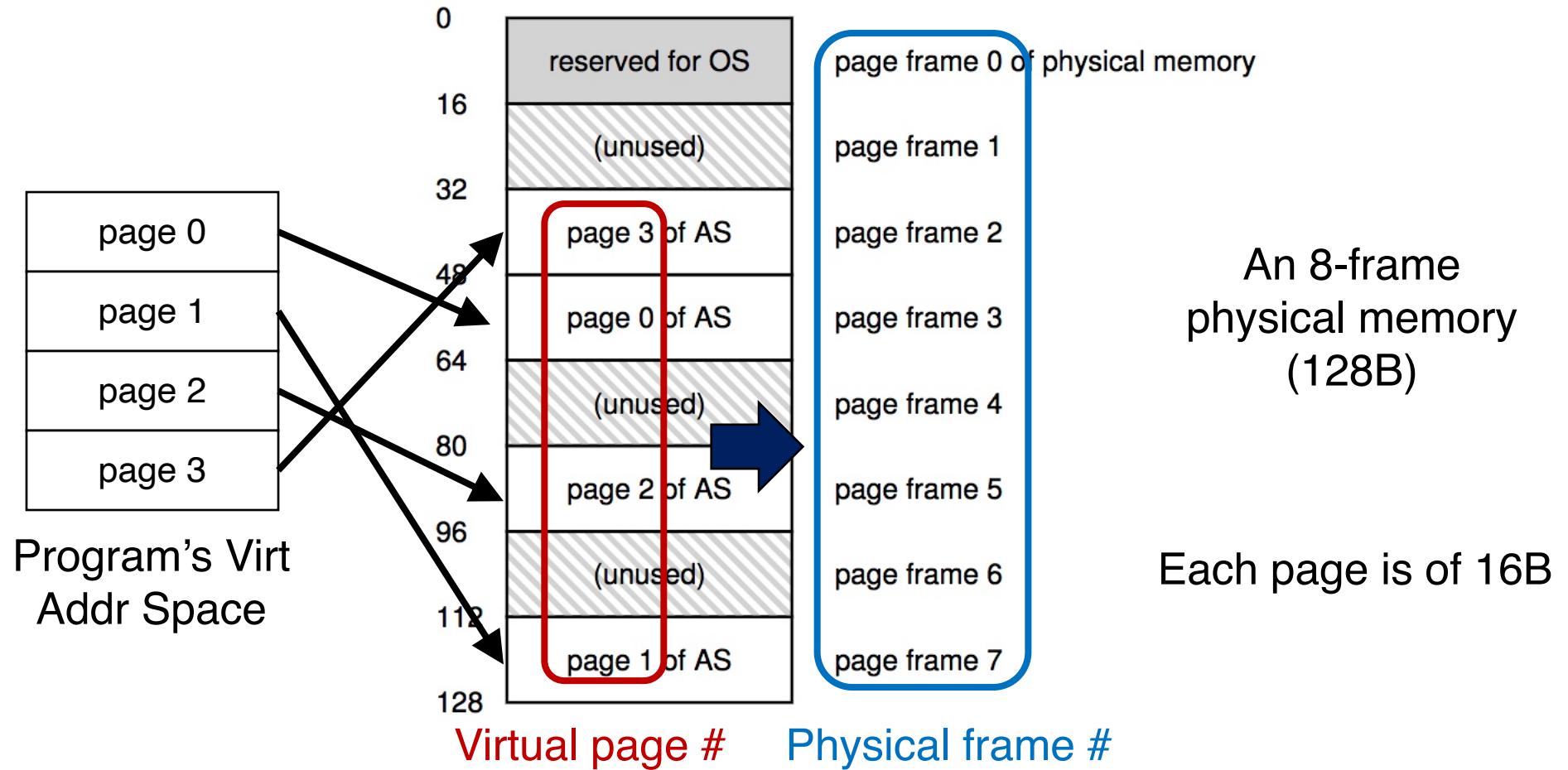
# Paging Scheme

- A memory management scheme that allows the physical address space of a process to be **non-contiguous**
- Divide **physical memory** into fixed-sized blocks called **frames**
- Divide **logical memory** into blocks of same size called **pages**
- Flexible mapping: Any page can go to any free frame
- Scalability: To run a program of size  $n$  pages, need to find  $n$  free frames and load program
  - **Grow memory segments wherever we please!**

# A Simple Example

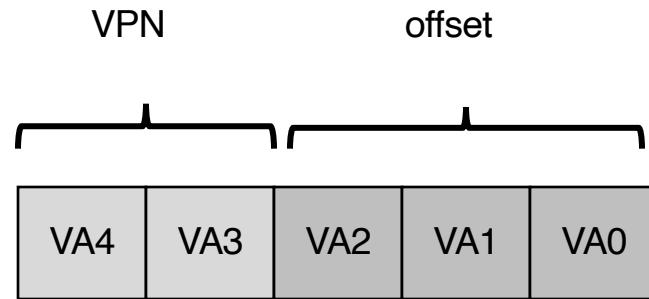


# A Simple Example



# Addressing Basics

- For segmentation
  - High bits => segment #
  - Low bits => offset
- For paging
  - High bits => page #
  - Low bits => offset



Q: How many offset bits do we need?

A: **log(page\_size)**

# Address Examples

Page size	Low bits (offset)
16 Bytes	4
2 KB	11
4 MB	22
256 Bytes	8
16 KB	14

# Address Examples

Page size	Low bits (offset)	Virt Addr bits	High bits (vpn)
16 Bytes	4	10	
2 KB	11	20	
4 MB	22	32	
256 Bytes	8	16	
16 KB	14	64	

# Address Examples

Page size	Low bits (offset)	Virt Addr bits	High bits (vpn)
16 Bytes	4	10	6
2 KB	11	20	9
4 MB	22	32	10
256 Bytes	8	16	8
16 KB	14	64	50

# Address Examples

Page size	Low bits (offset)	Virt Addr bits	High bits (vpn)	Virt pages
16 Bytes	4	10	6	
2 KB	11	20	9	
4 MB	22	32	10	
256 Bytes	8	16	8	
16 KB	14	64	50	

# Address Examples

Page size	Low bits (offset)	Virt Addr bits	High bits (vpn)	Virt pages
16 Bytes	4	10	6	64
2 KB	11	20	9	512
4 MB	22	32	10	1K
256 Bytes	8	16	8	256
16 KB	14	64	50	$2^{50}$

# Address Examples

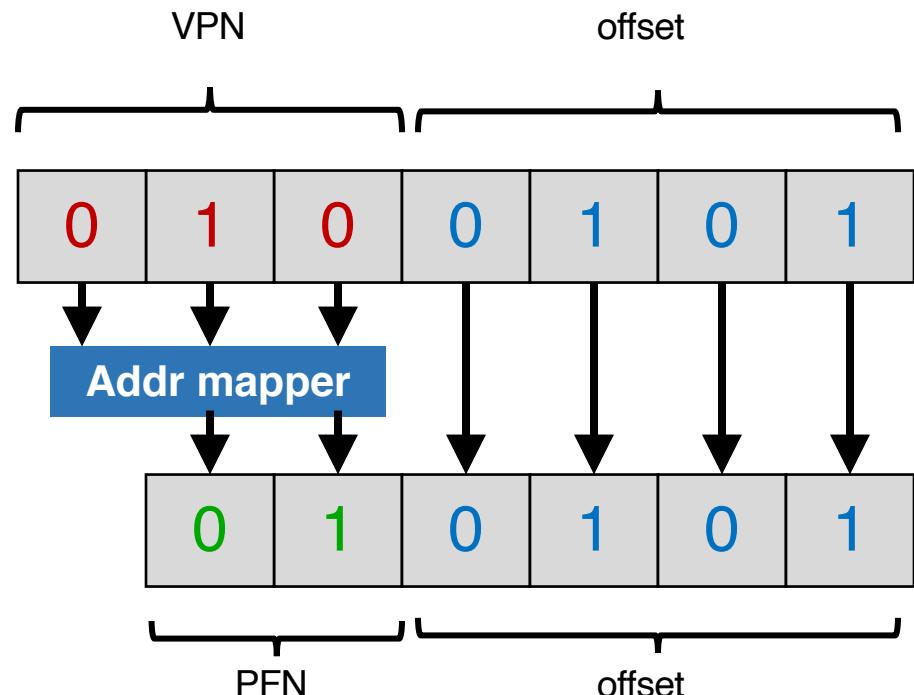
Page size	Low bits (offset)	Virt Addr bits	High bits (vpn)	Virt pages
16 Bytes	4	10	6	64
2 KB	11	20	9	512
4 MB	22	32	10	1K
256 Bytes	8	16	8	256
16 KB	14	64	50	$2^{50}$

Note: high bits for physical frames may be different!

Question: An x86\_64 Linux OS with 4KB page size. How many pages can we have assuming the maximum memory limit?

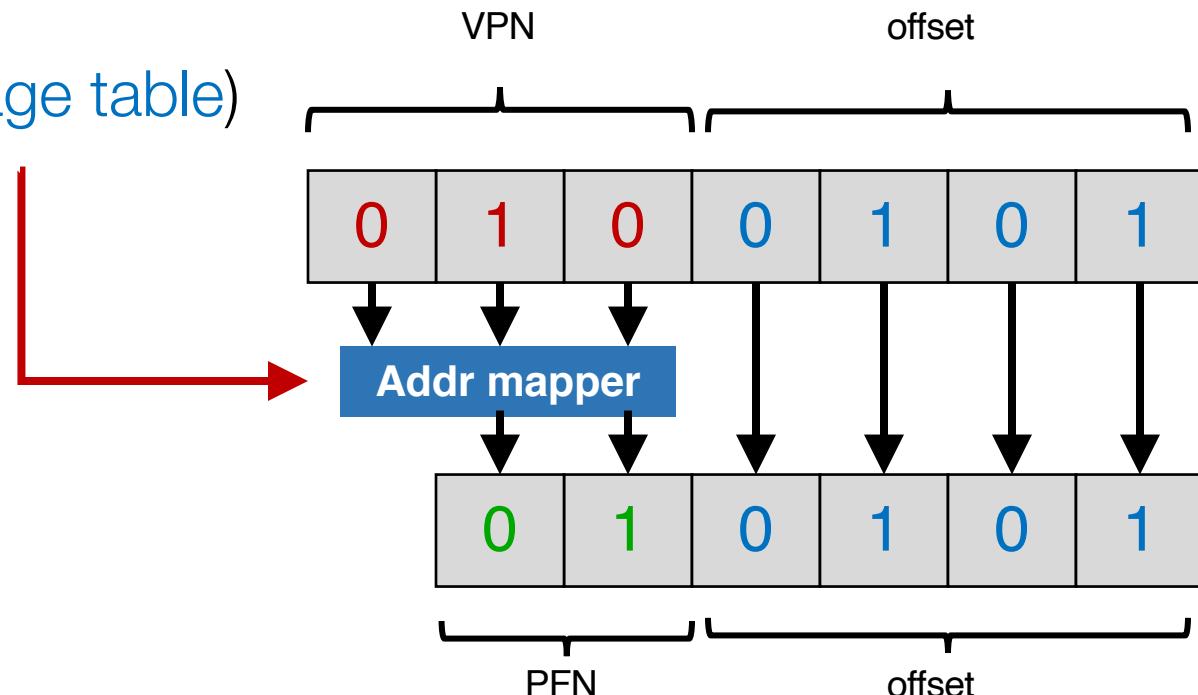
# Virtual => Physical Addr Mapping

- We need a general mapping mechanism
- What data structure is good?
  - Big array

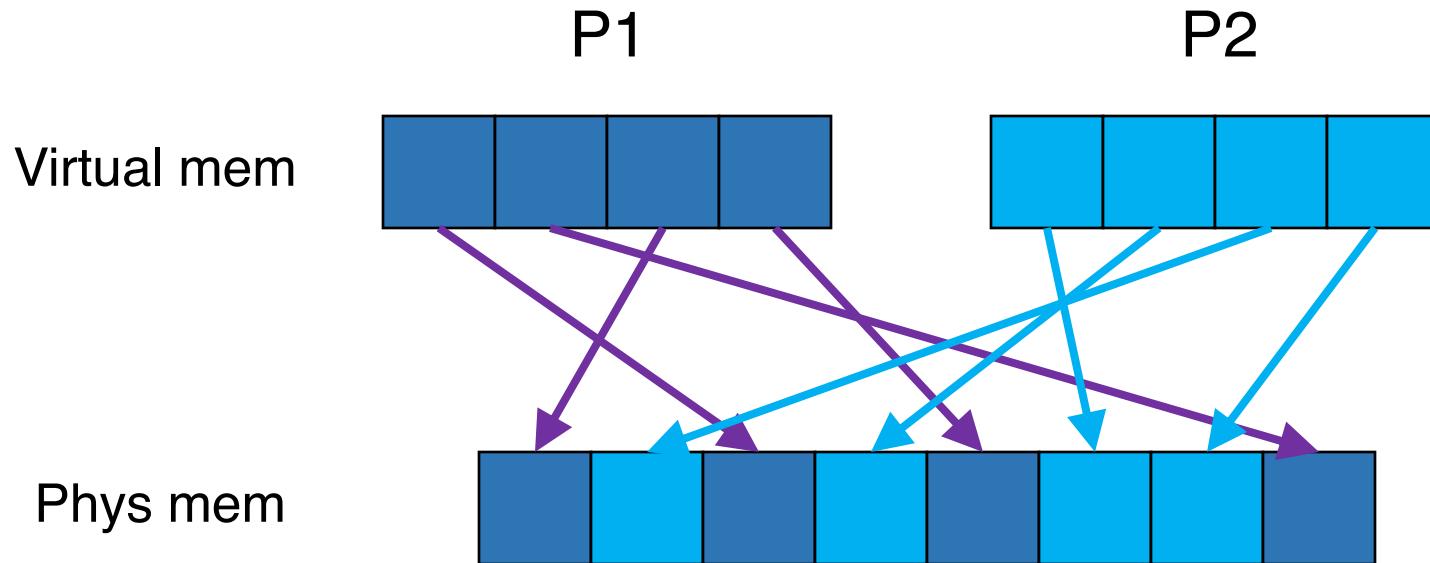


# Virtual => Physical Addr Mapping

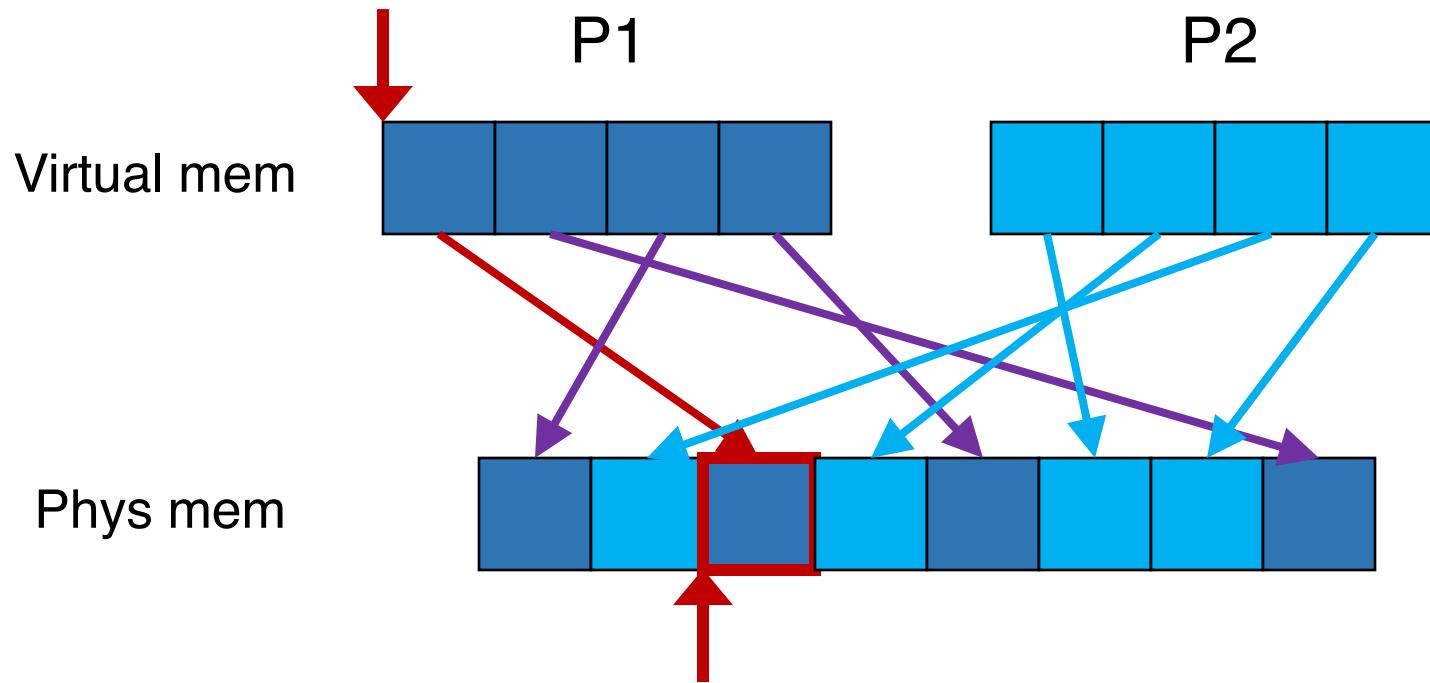
- We need a general mapping mechanism
- What data structure is good?
  - Big array
  - (aka [linear page table](#))



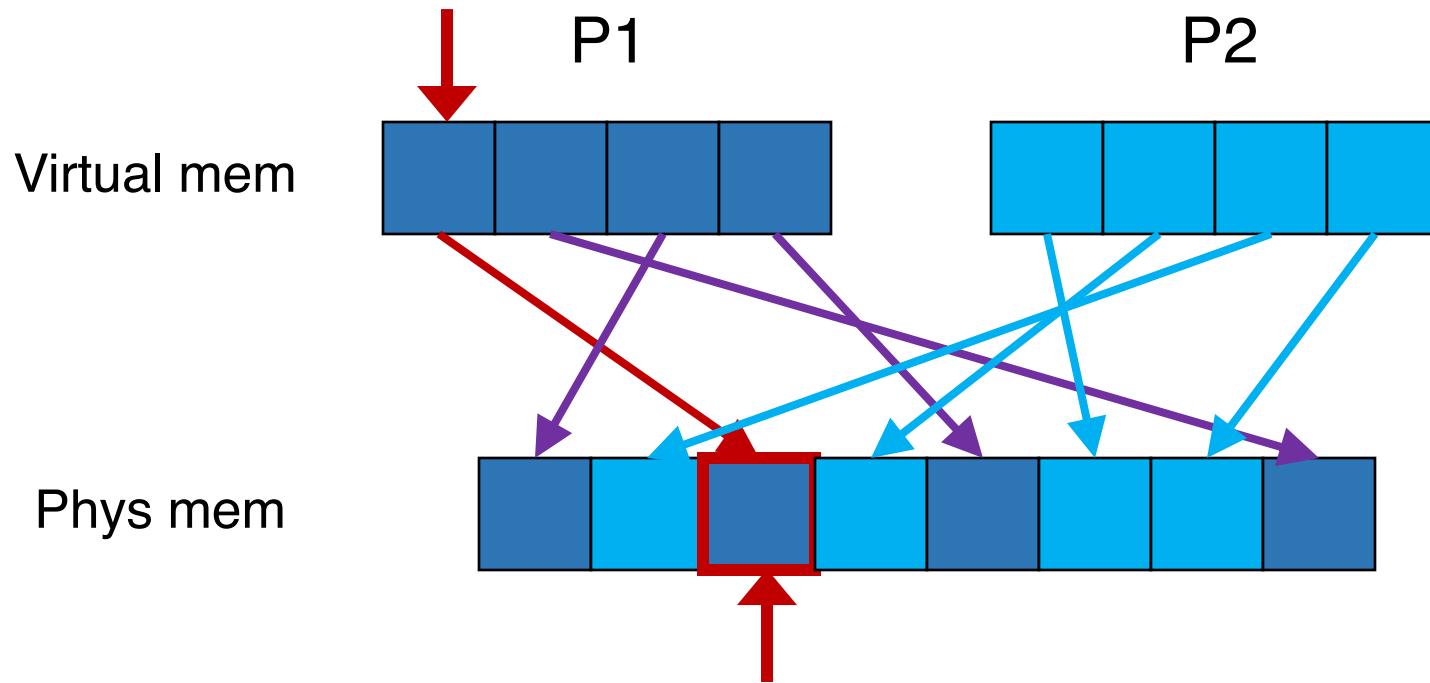
# Mapping Example



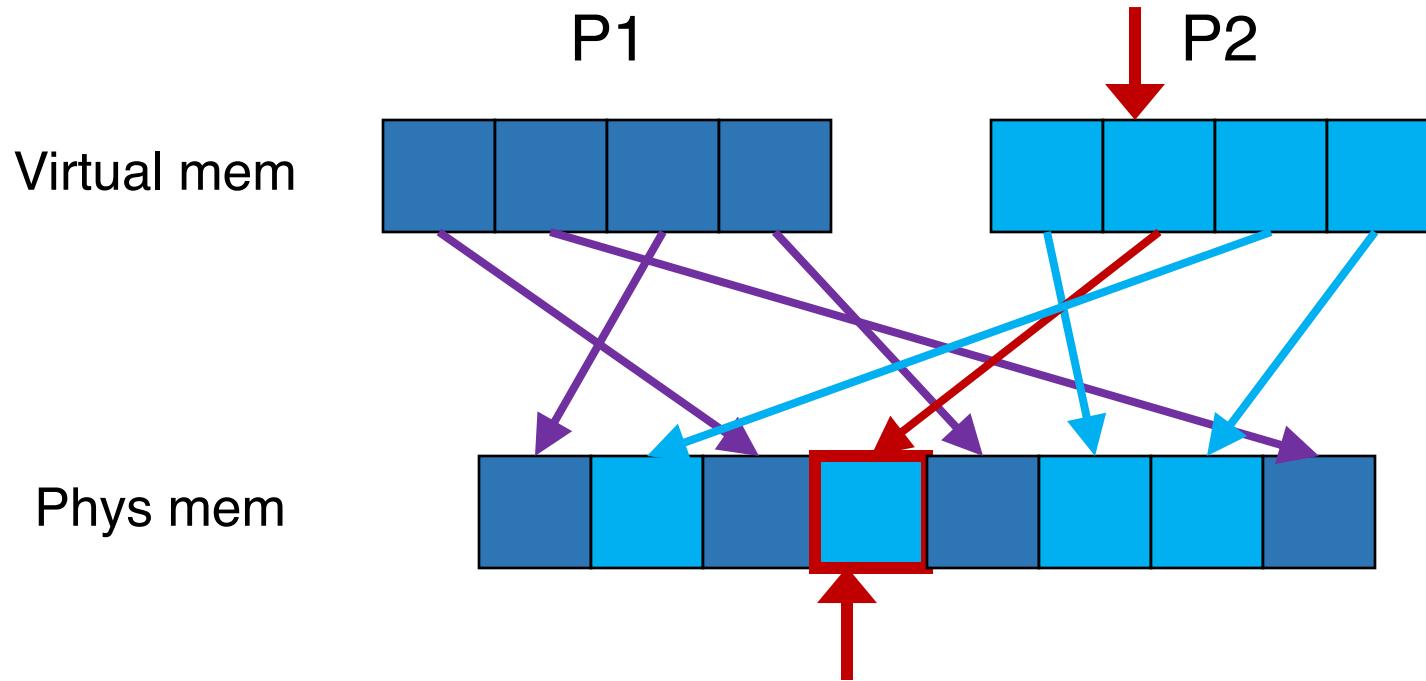
# Mapping Example



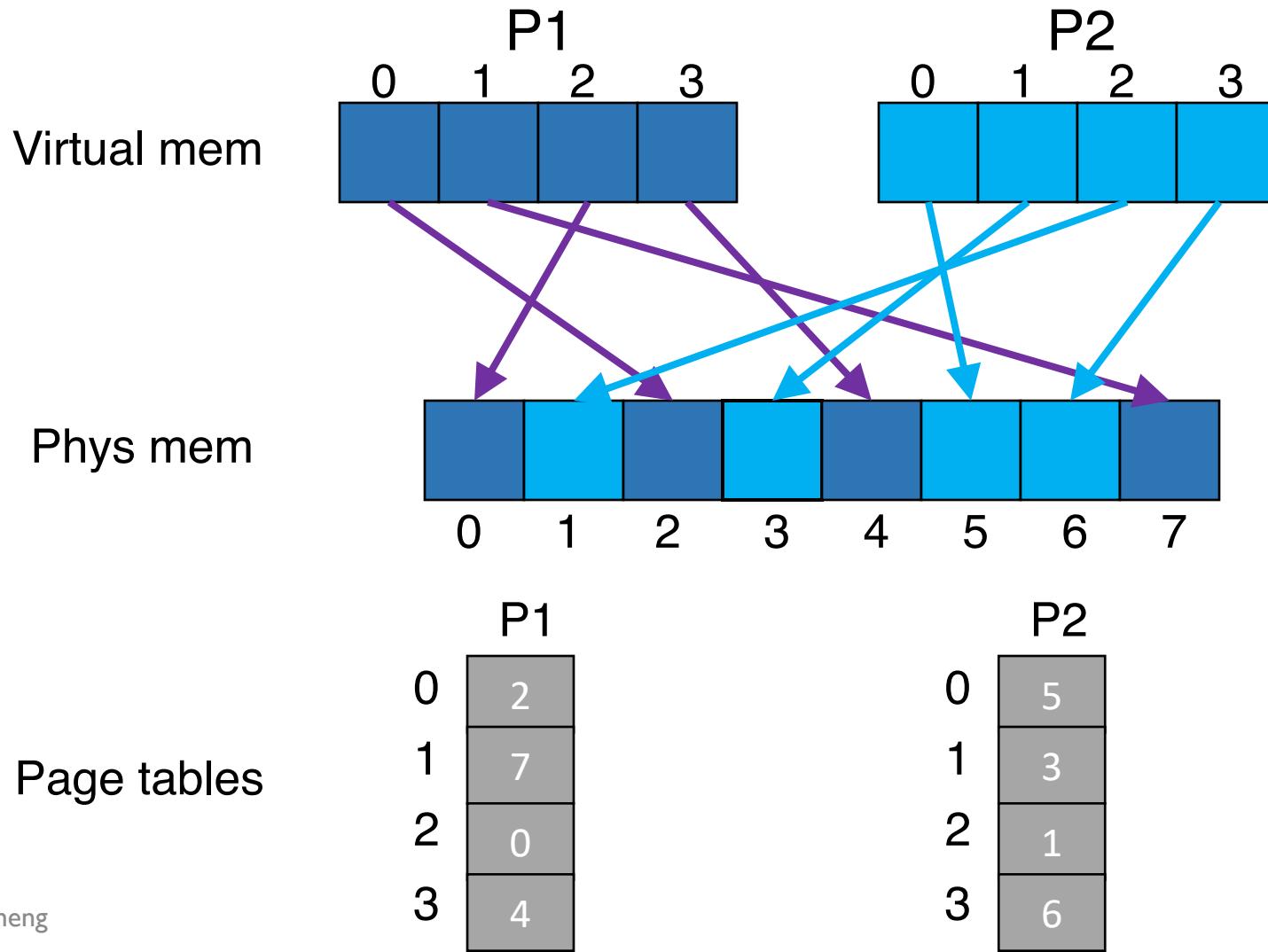
# Mapping Example



# Mapping Example



# Mapping Example



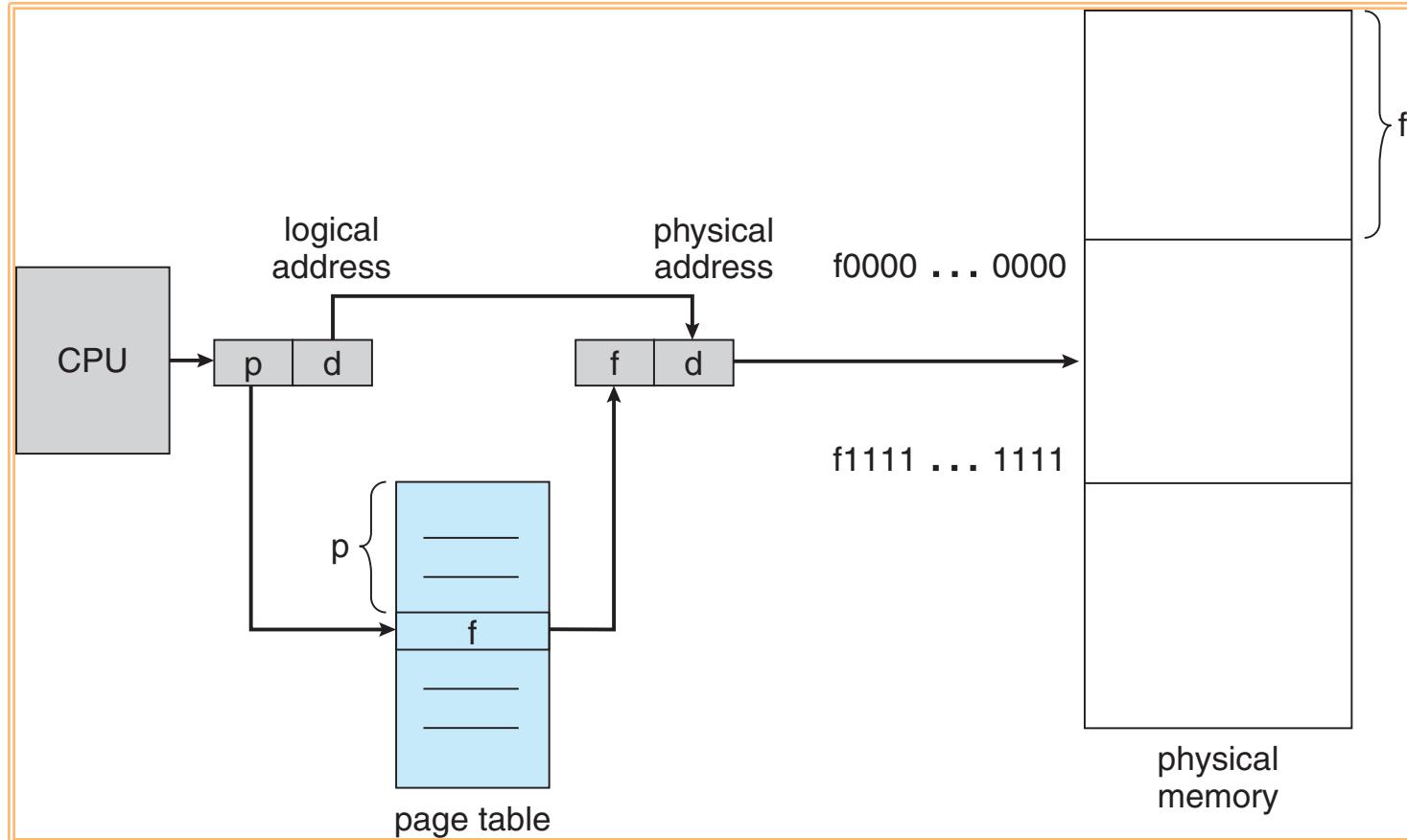
# Page Table

- A **per-process** data structure used to keep track of virtual page to physical frame mapping
- Major role: store **address translation**

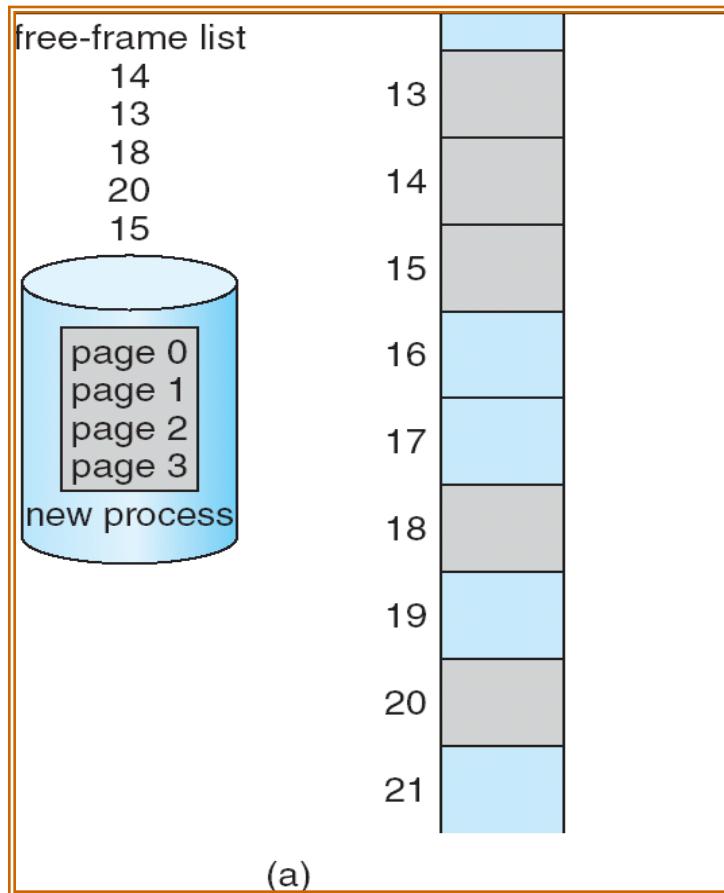
# Address Translation Scheme

- Observe: The simple limit/relocation register pair mechanism is no longer sufficient
- m-bit virtual address generated by CPU is divided into:
  - Virtual Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

# Address Translation Architecture

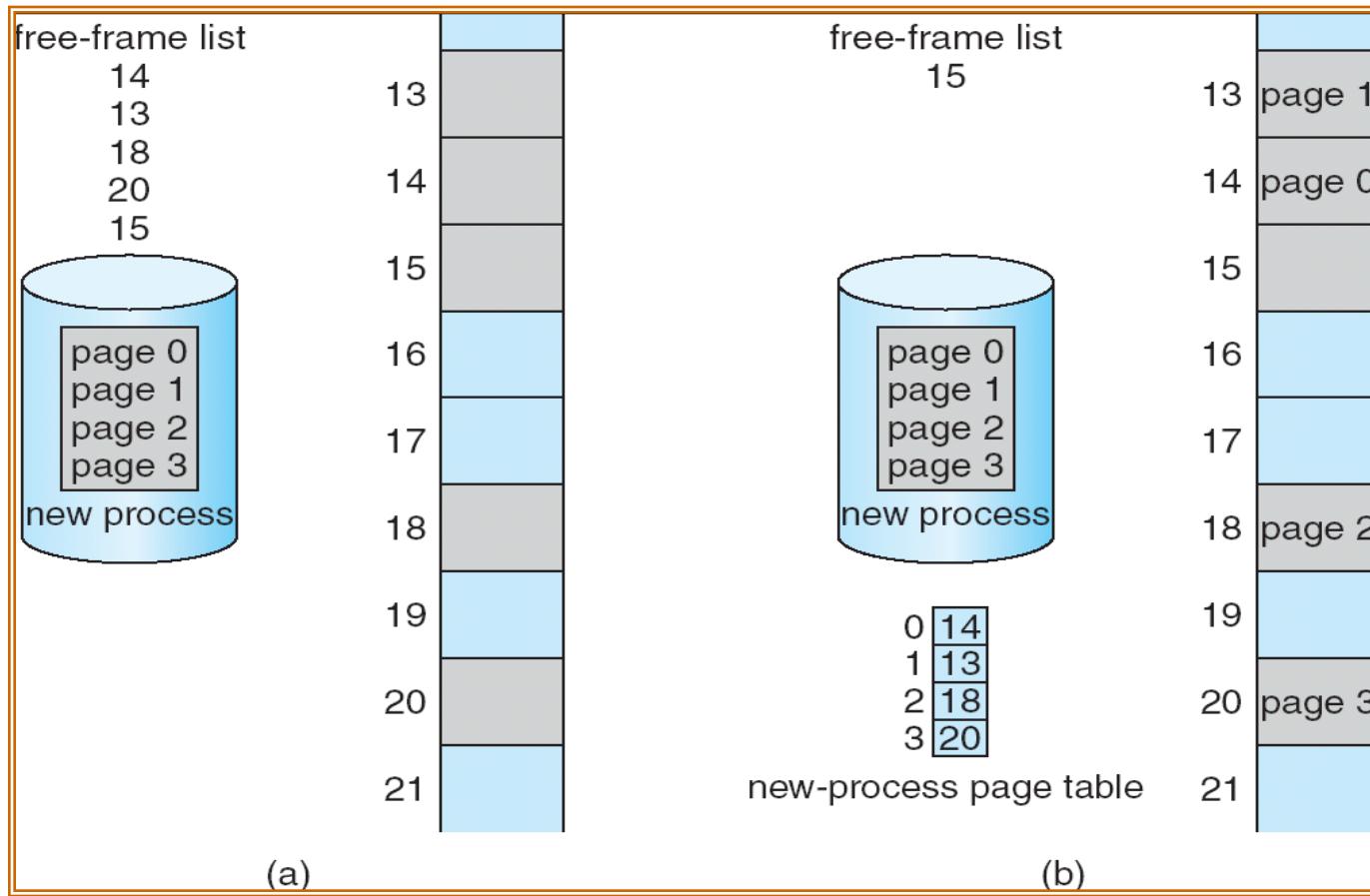


# Free Frames



Before allocation

# Free Frames



Before allocation

After allocation

# More on Page Table

- The page table data structure is kept in main memory
- Each **page table entry** (PTE) holds  
`<physical translation + other info>`
- **Page-table base register** (PTBR) points to the page table
  - E.g., CR3 on x86
- **Page-table length register** (PTLR), if it exists, indicates the size of the page table

# Page Table Entry (PTE)

- The simplest form of a page table is a **linear page table**
  - Array data structure
  - OS indexes the array by virtual page number (VPN)
  - To find the desired physical frame number (PFN)

An 32-bit x86 page table entry (PTE)

