Memory Management: Paging

CS 571: Operating Systems (Spring 2020)

Lecture 7a

Yue Cheng

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

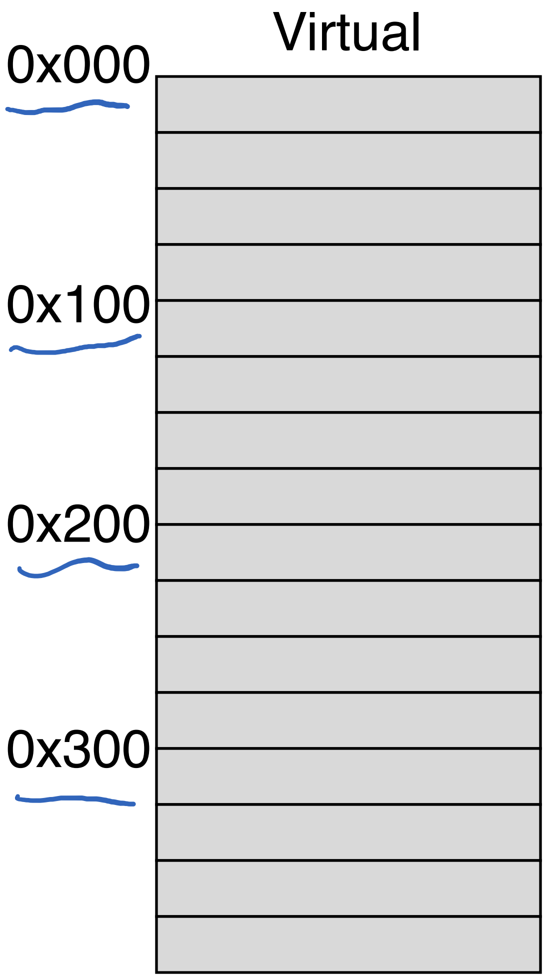
Segmentation Example

- Assume a 10-bit virtual address space
  - With the high 2-bit indicating the segment

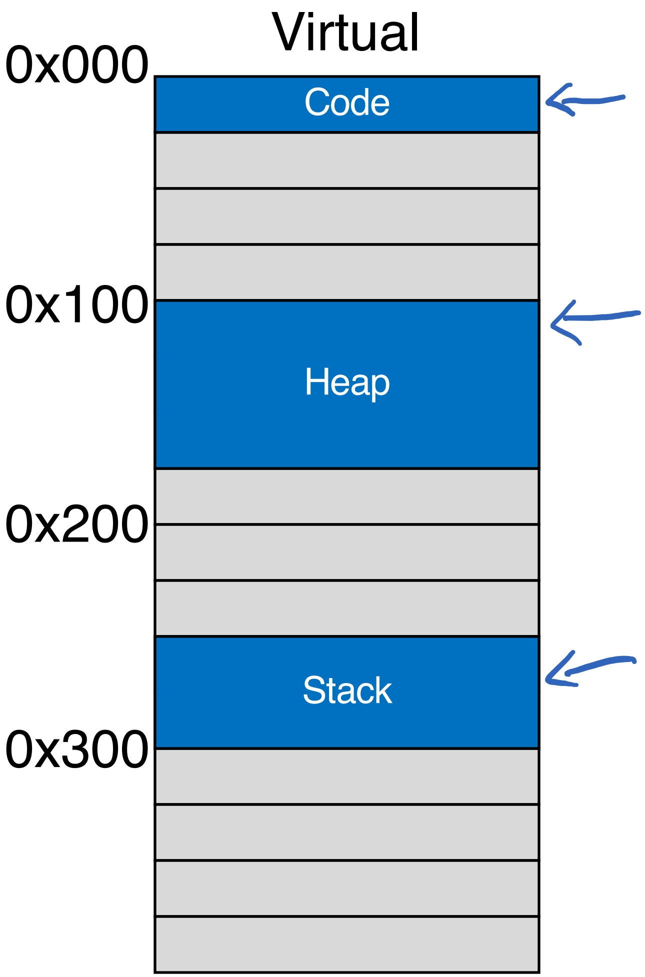
- Assume
  - 0 => code+data
  - 1 => heap
  - 2 => stack

16 mem blks.

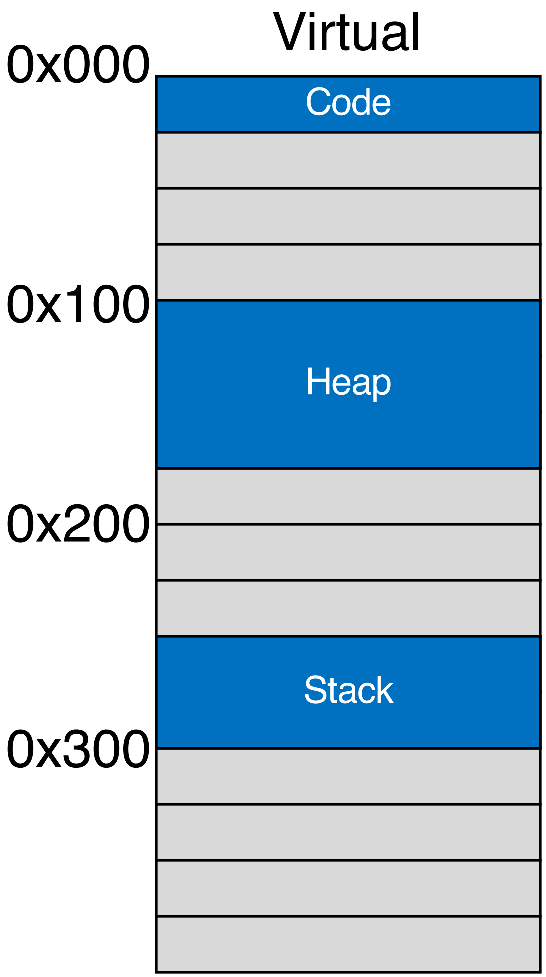
Segmentation Example



Segmentation Example



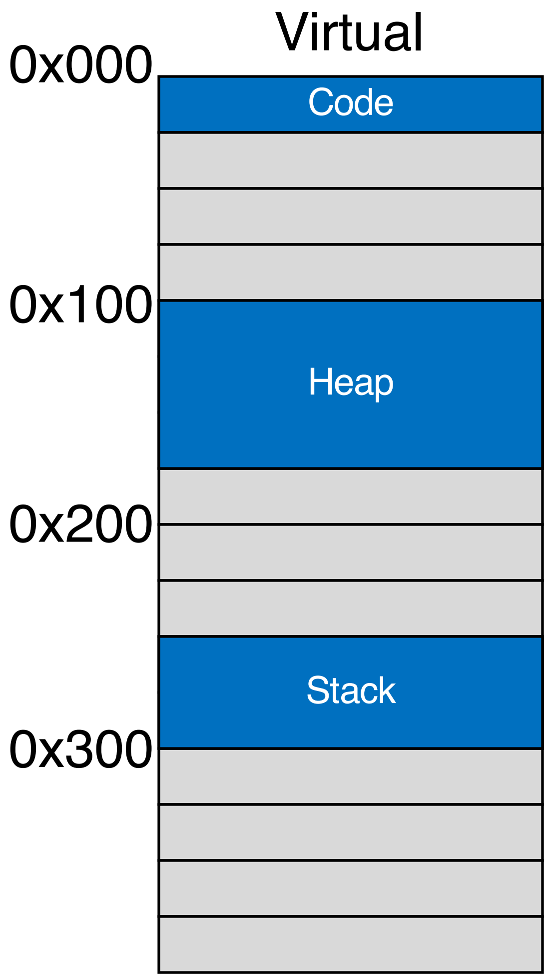
Segmentation Example



Segment table

	base	bounds
code	?	?
heap	?	?
stack	?	?

Segmentation Example

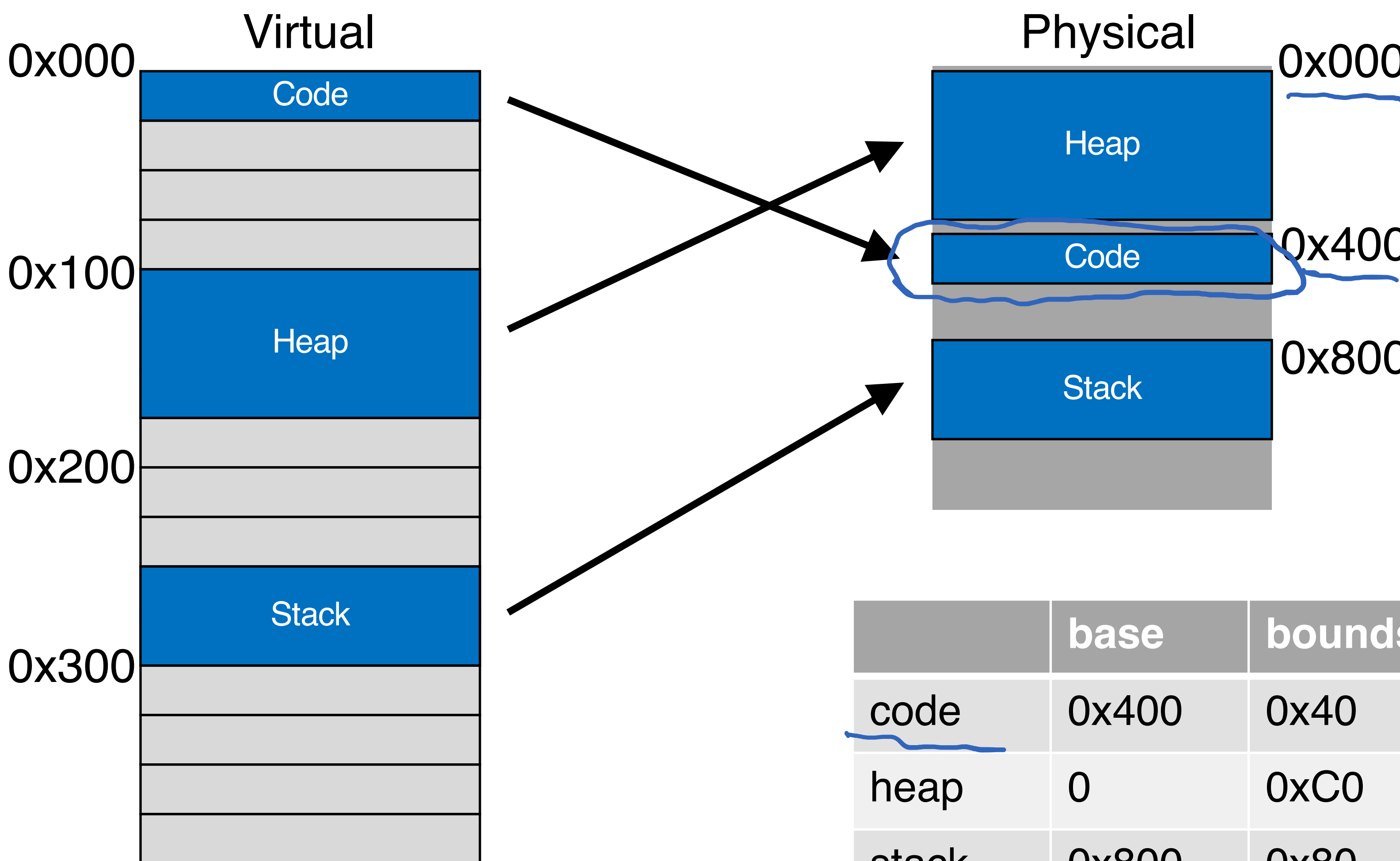


0x40 x 3
= 0x C0

physical
addr

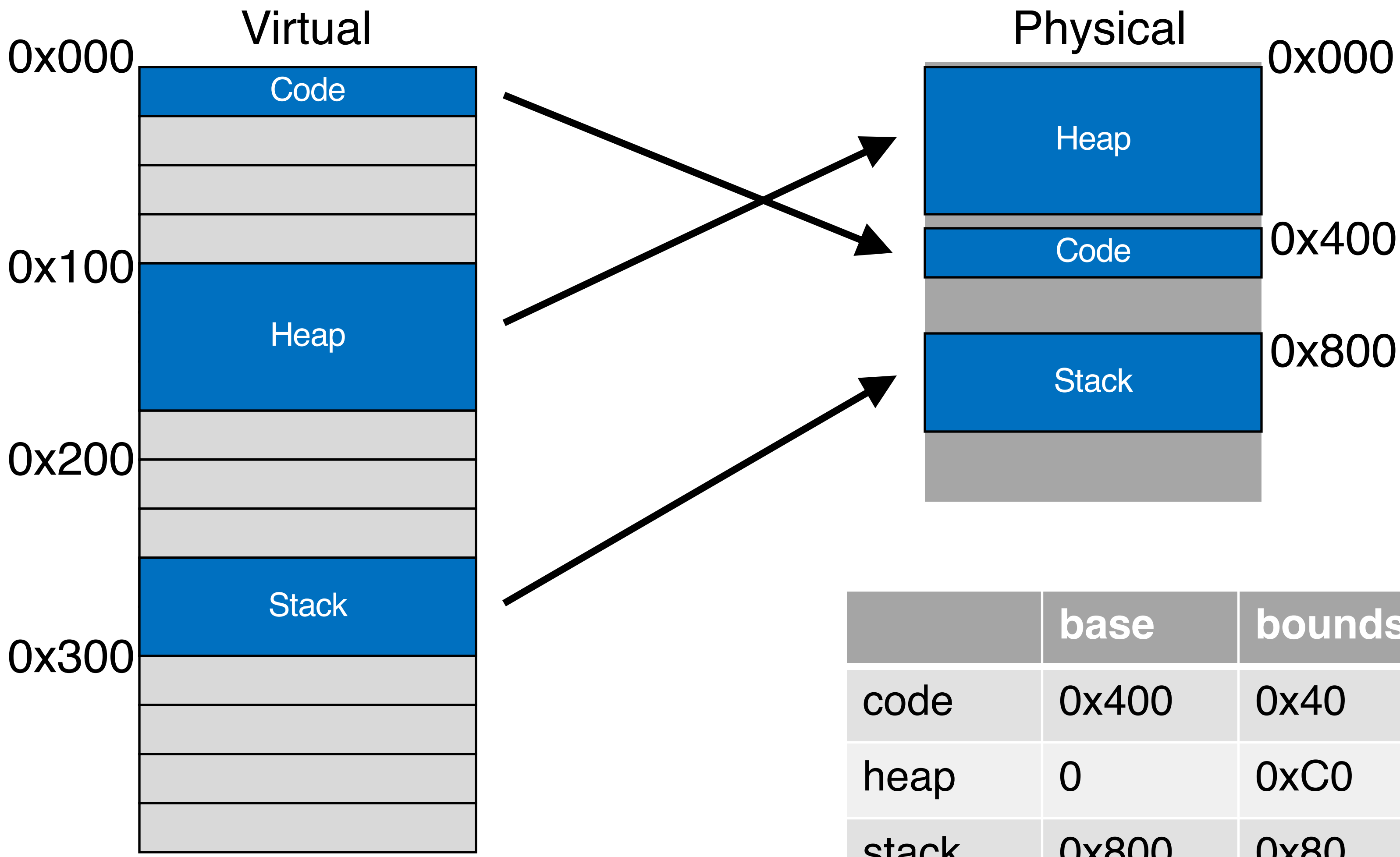
	base	bounds
code	?	0x40
heap	?	0xC0
stack	?	0x80

Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

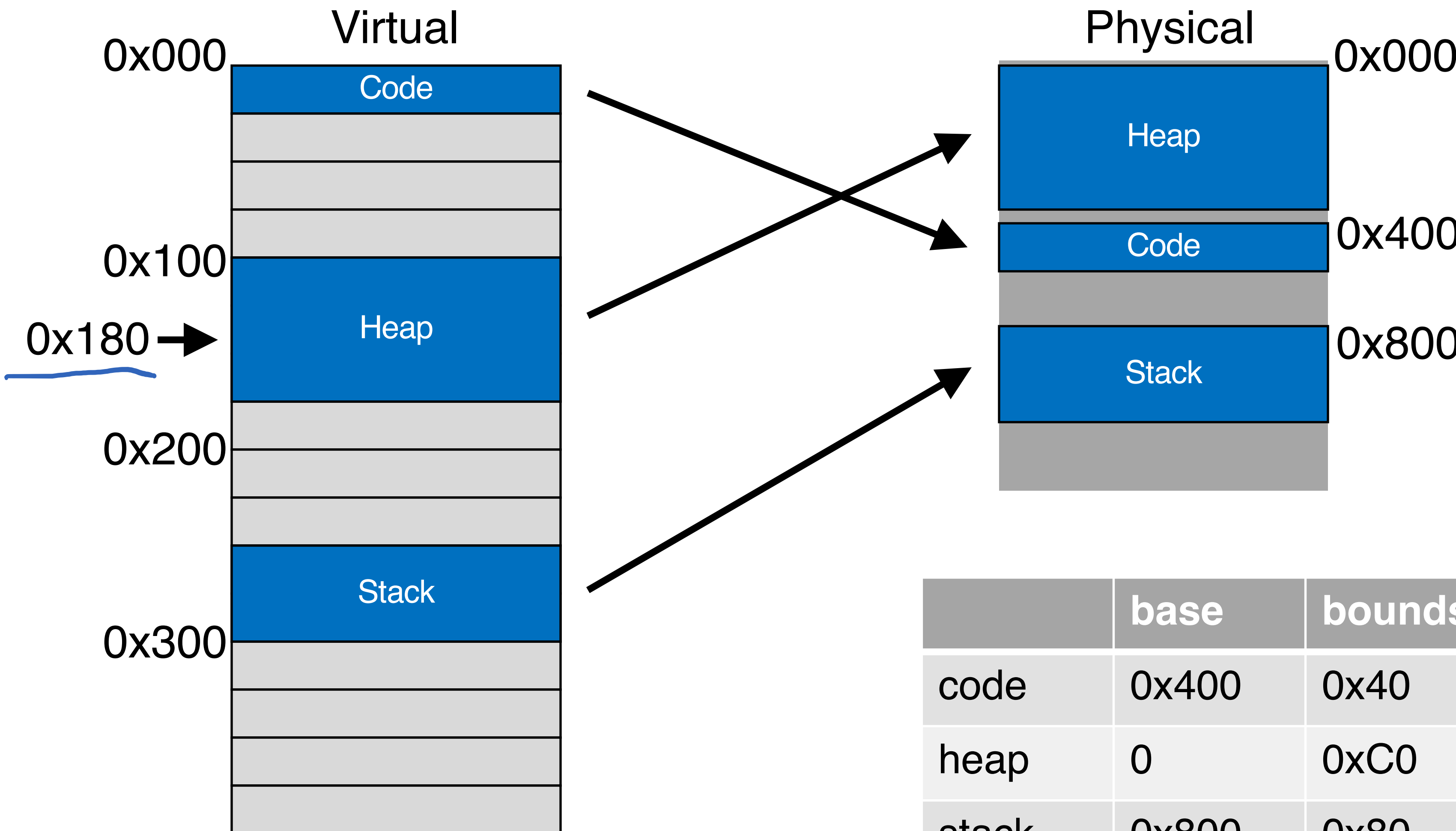
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = virt_offset + base`

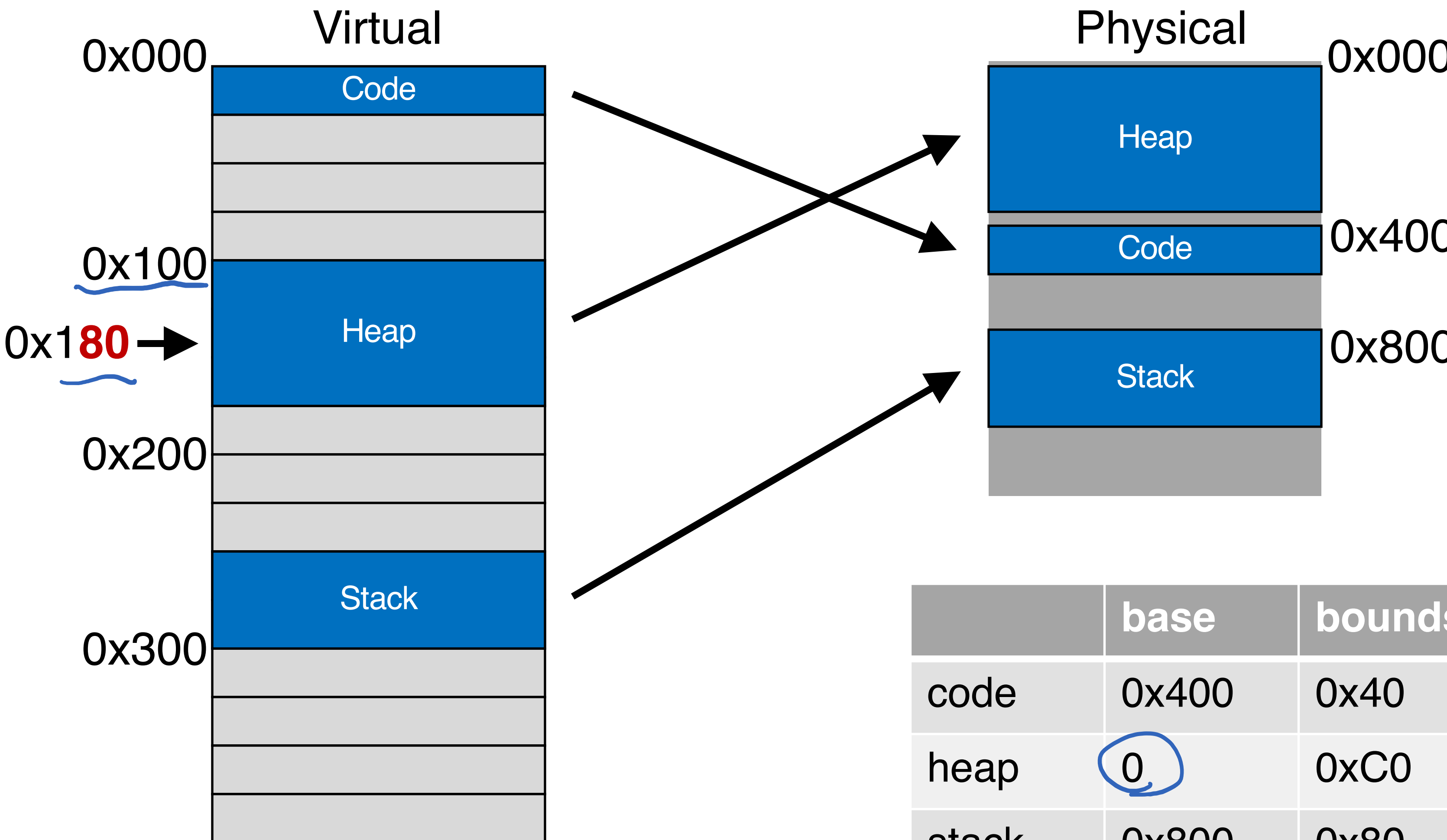
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = virt_offset + base`

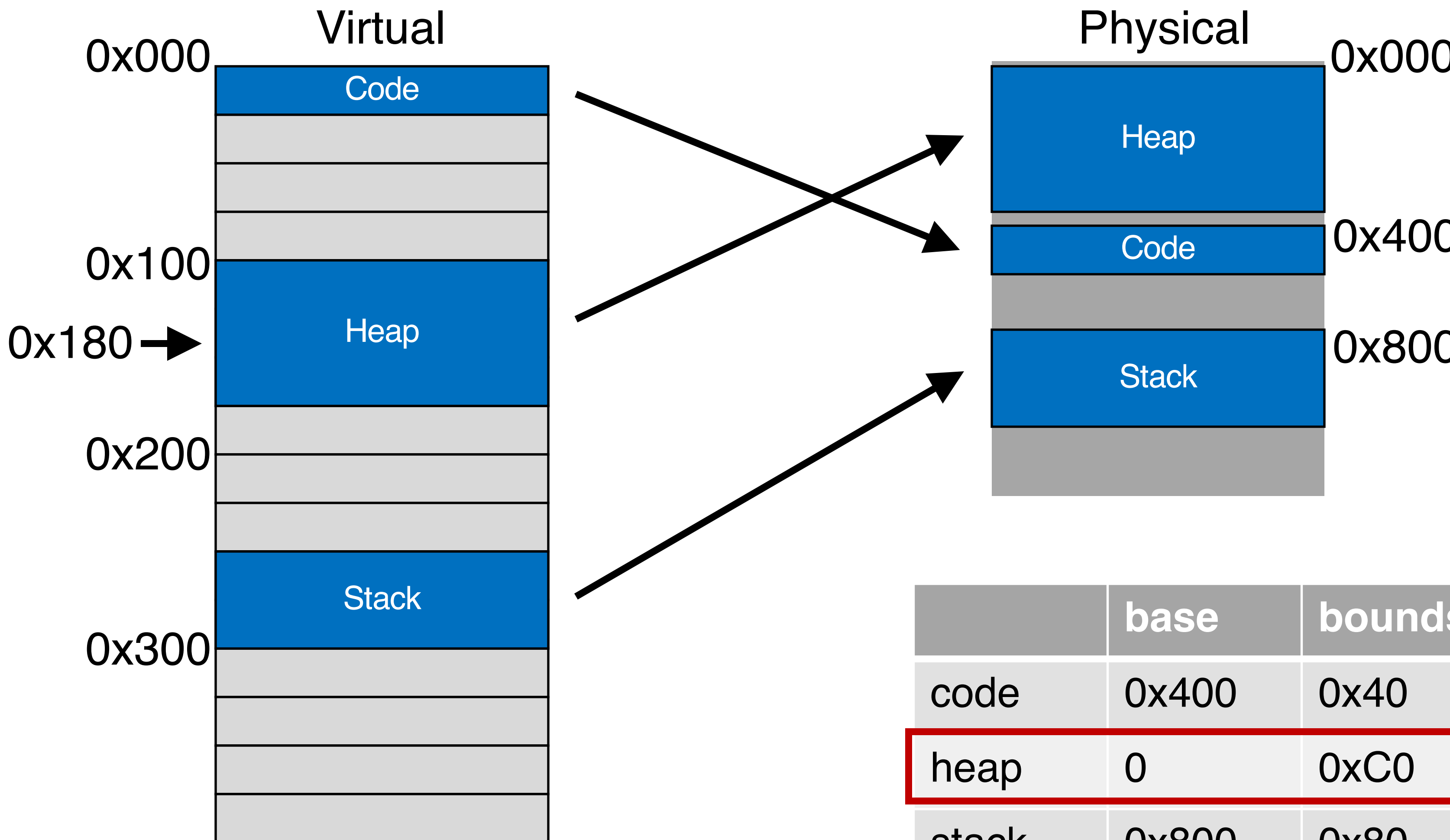
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = 0x80 + base`

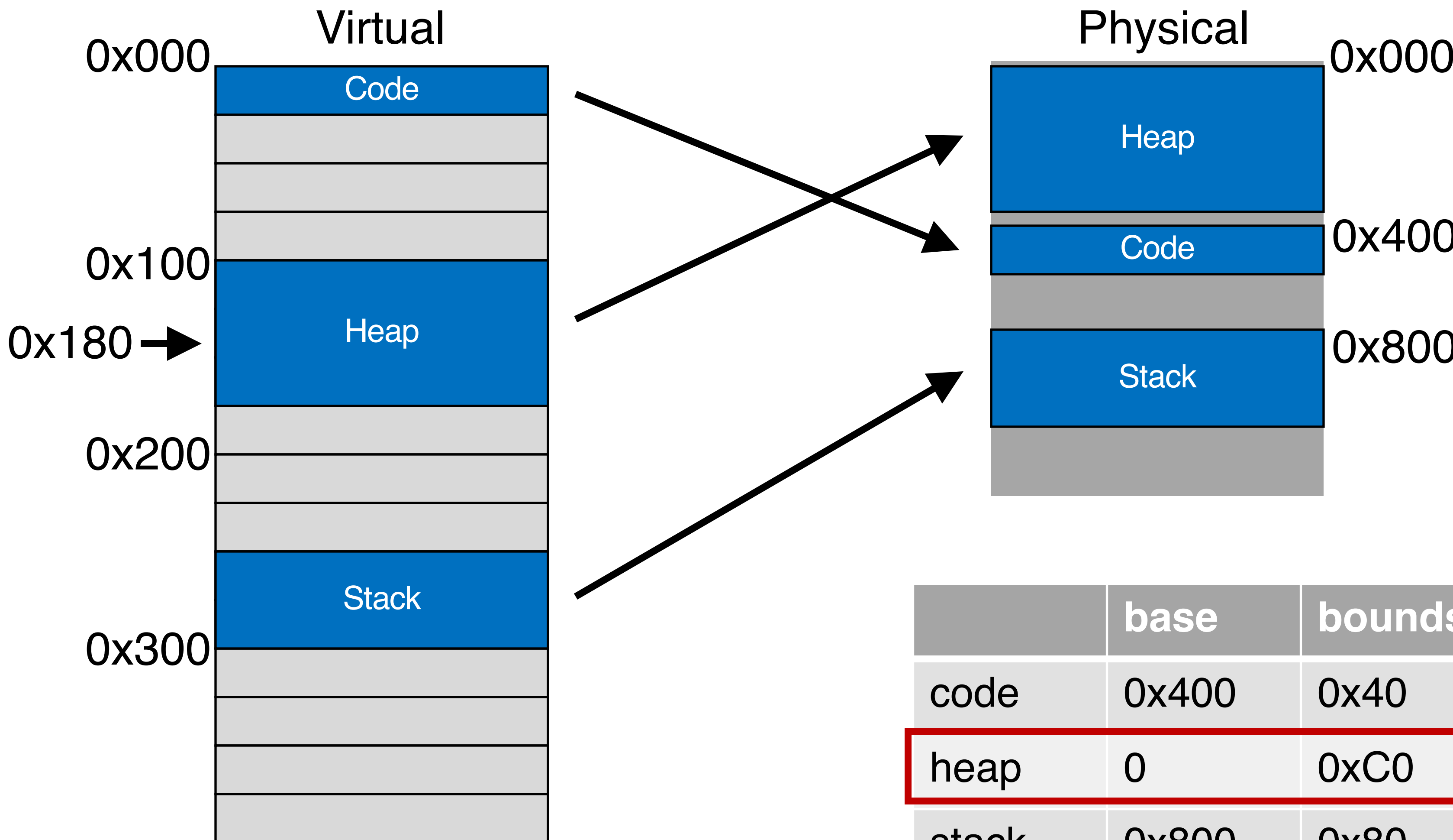
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = 0x80 + base`

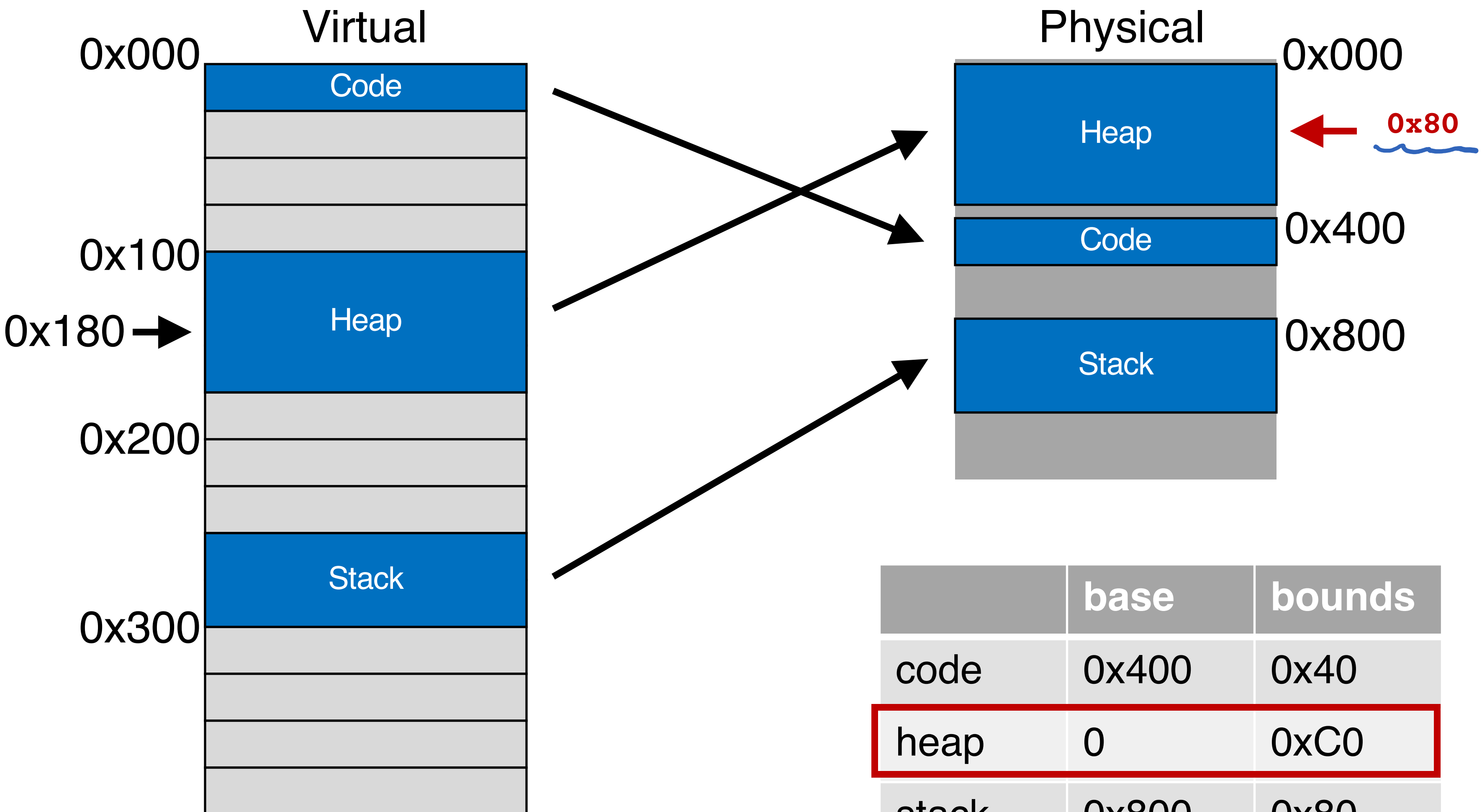
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = 0x80 + 0`

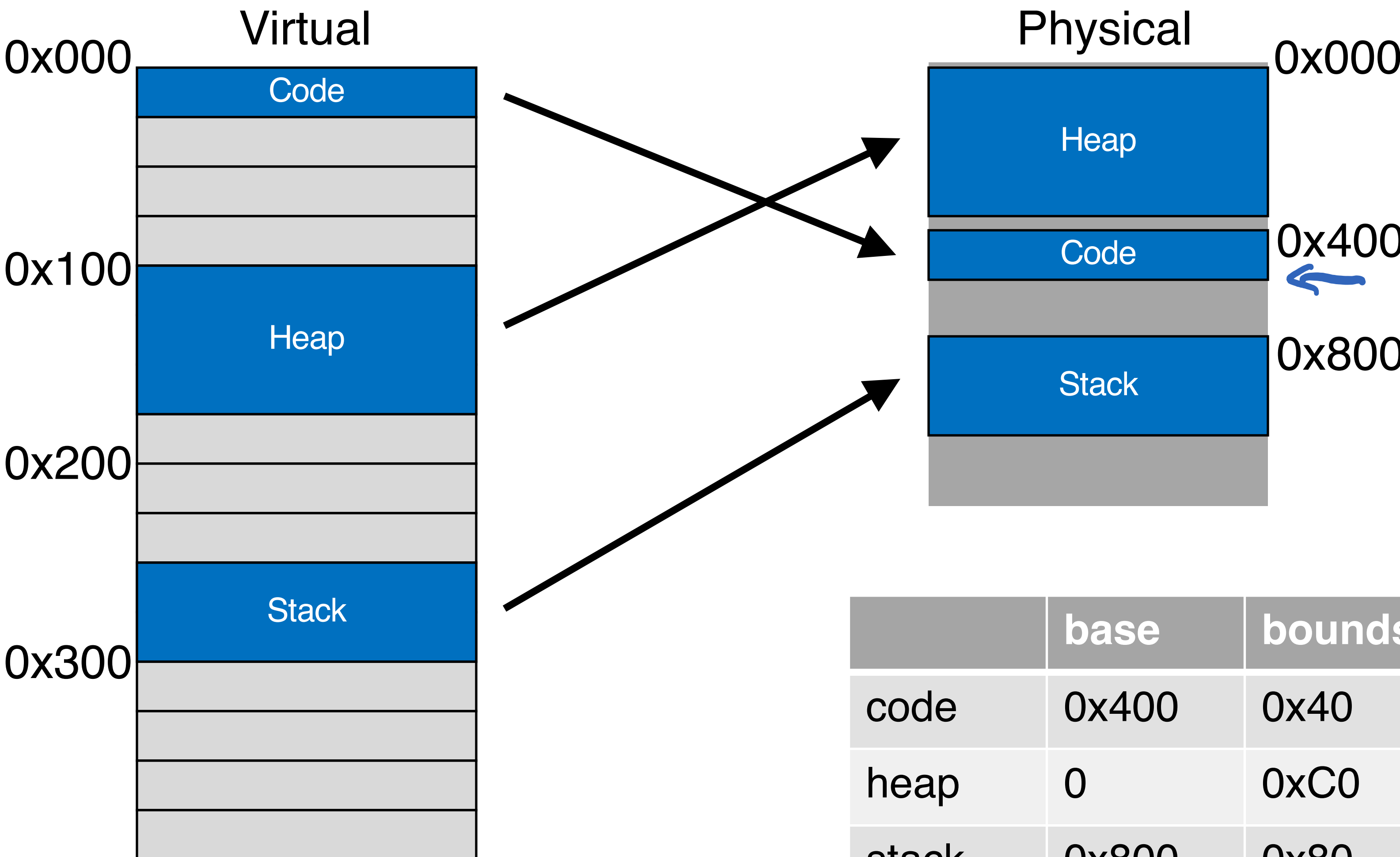
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

Most segments: `phys = 0x80 + 0 = 0x80`

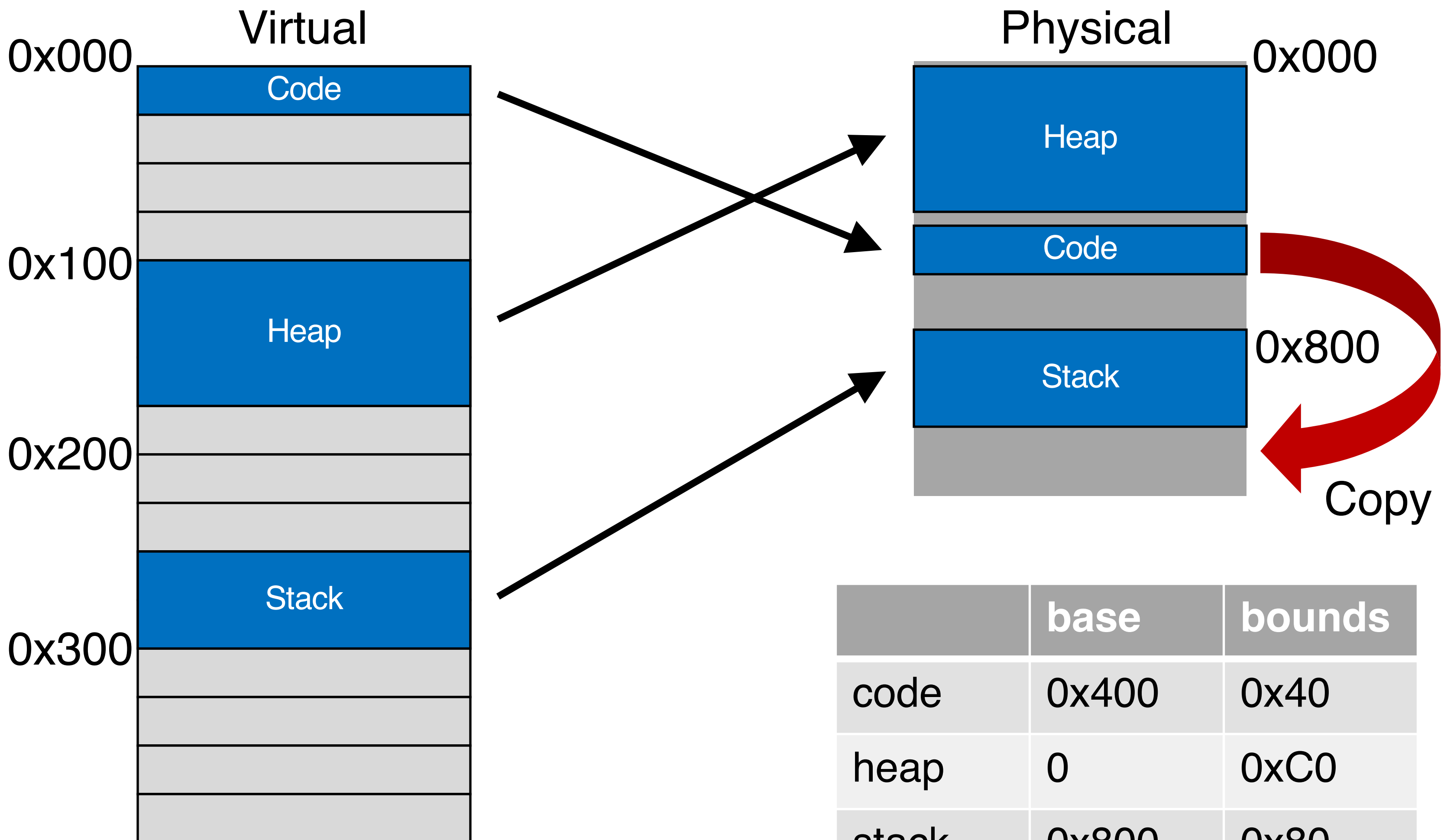
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

What if heap needs to grow?

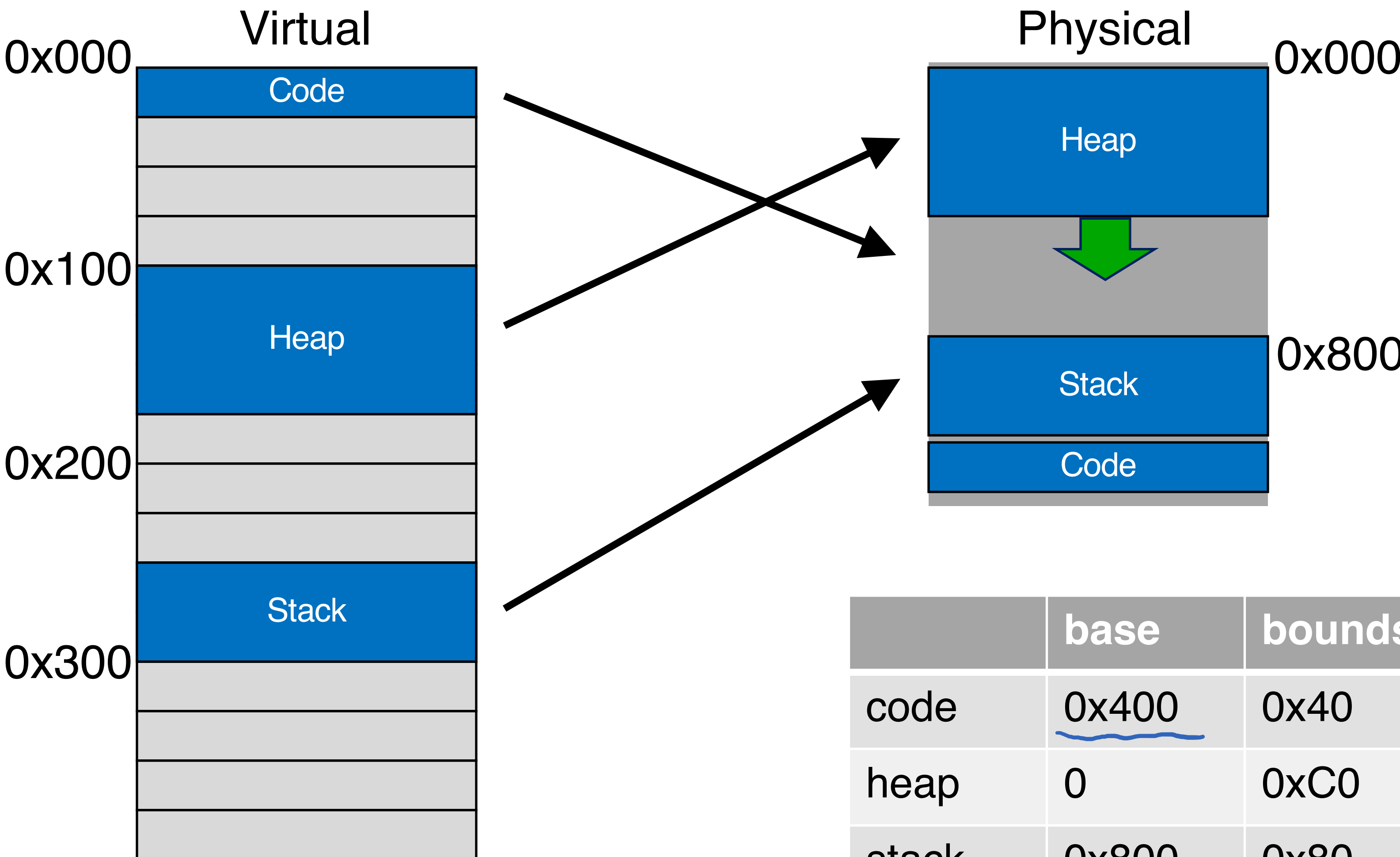
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

What if heap needs to grow?

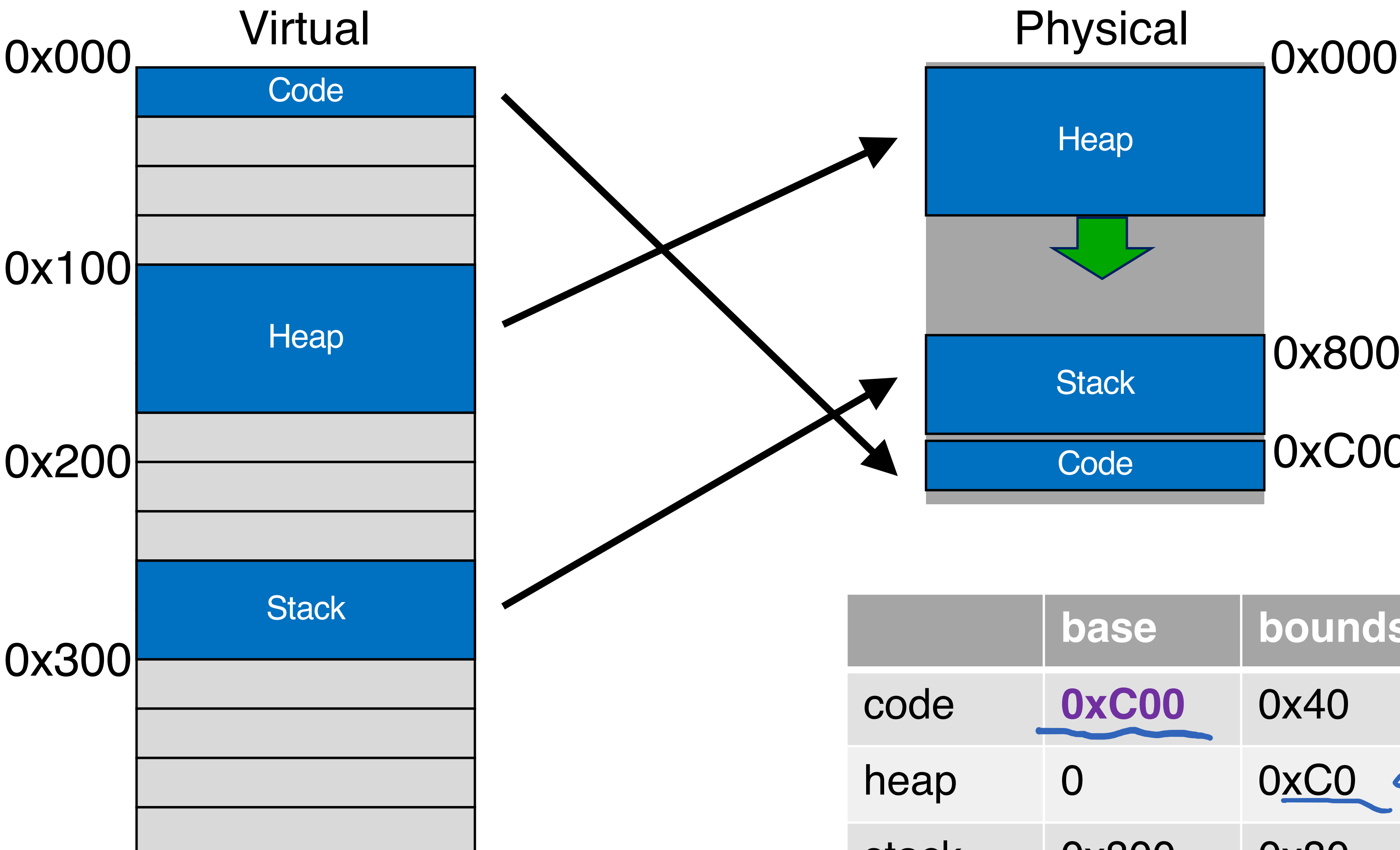
Segmentation Example



	base	bounds
code	0x400	0x40
heap	0	0xC0
stack	0x800	0x80

What if heap needs to grow?

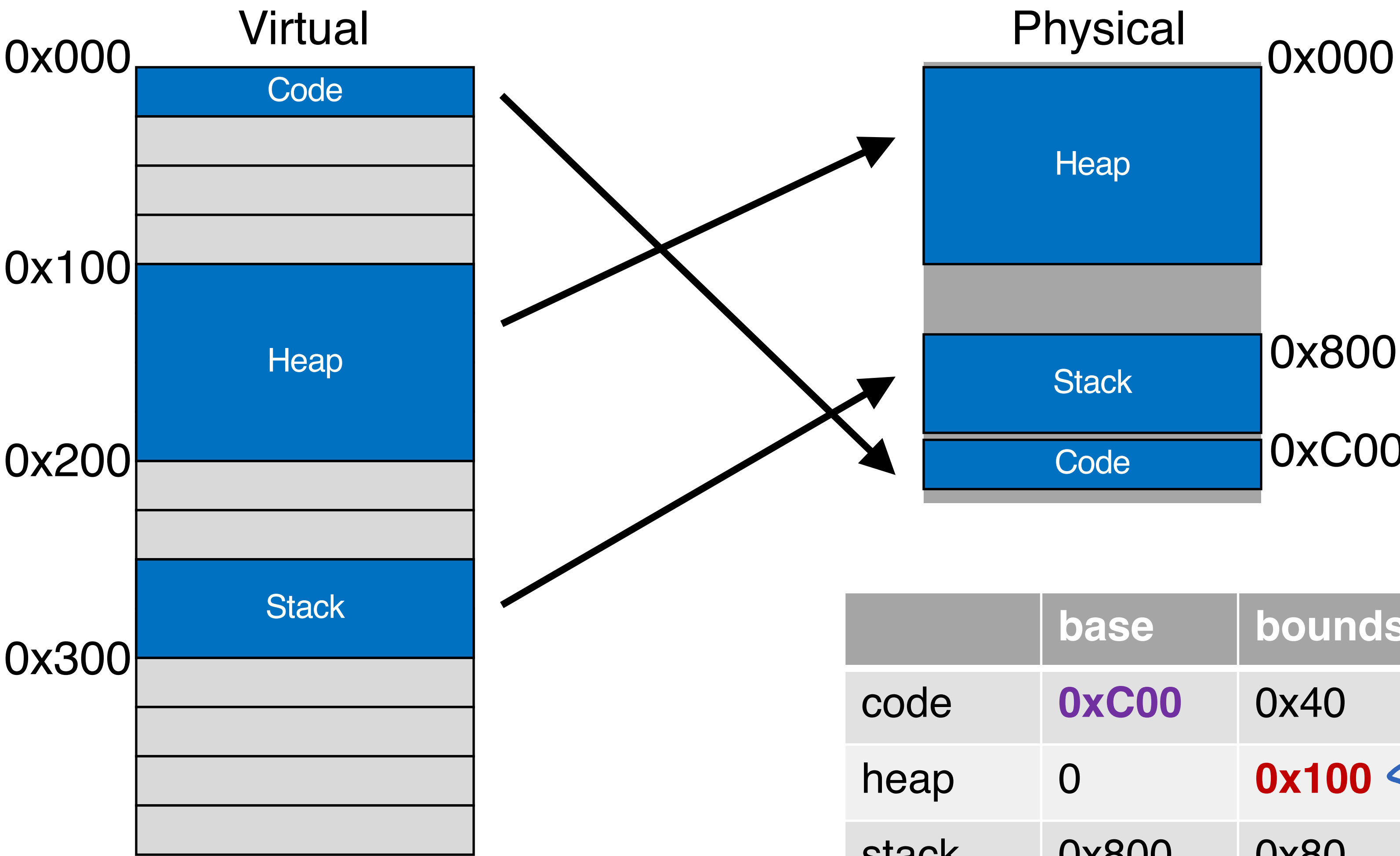
Segmentation Example



	base	bounds
code	**0xC00**	0x40
heap	0	0xC0
stack	0x800	0x80

What if heap needs to grow?

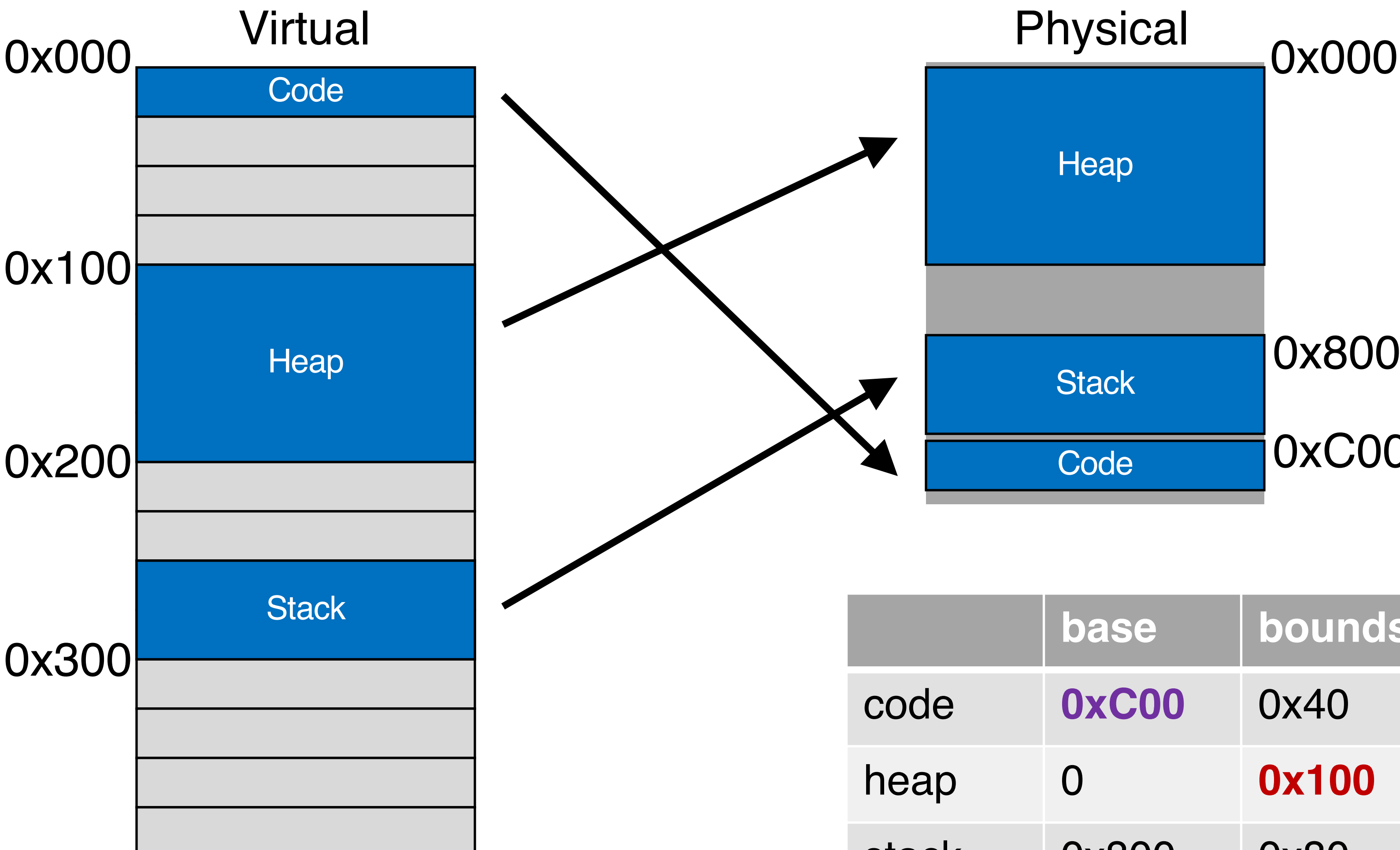
Segmentation Example



	base	bounds
code	**0xC00**	0x40
heap	0	**0x100**
stack	0x800	0x80

What if heap needs to grow?

Segmentation Example



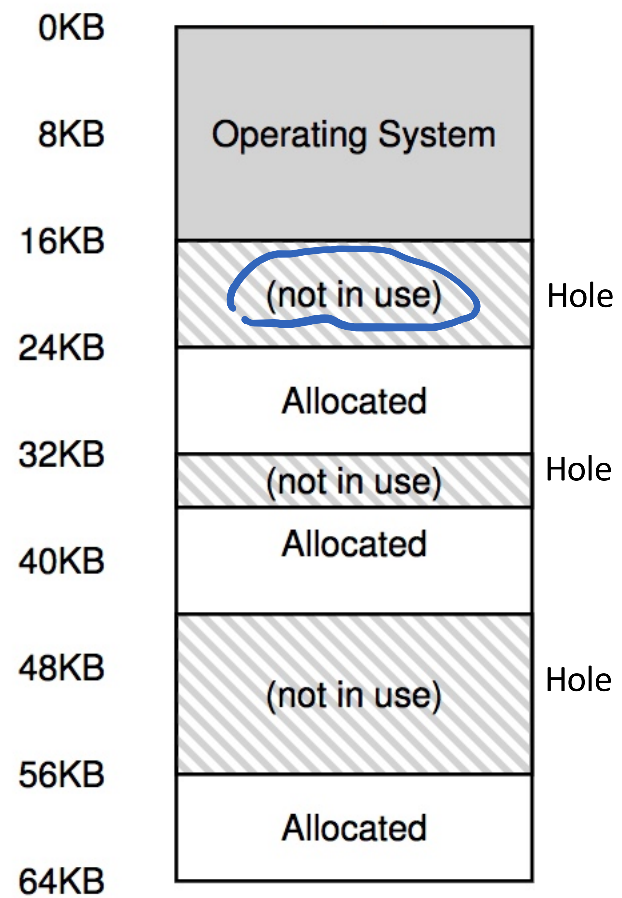
	base	bounds
code	**0xC00**	0x40
heap	0	**0x100**
stack	0x800	0x80

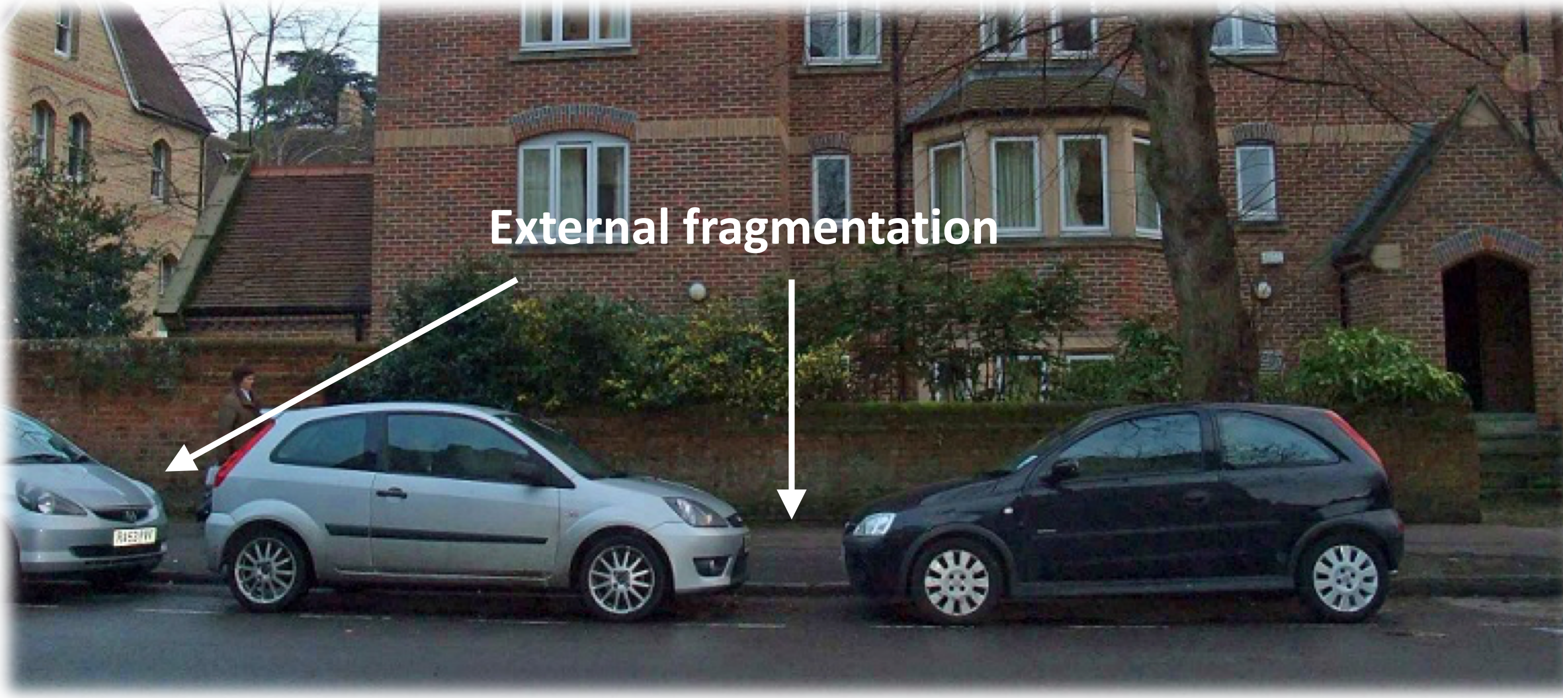
dilemma: must **(a) waste space** or **(b) waste time**

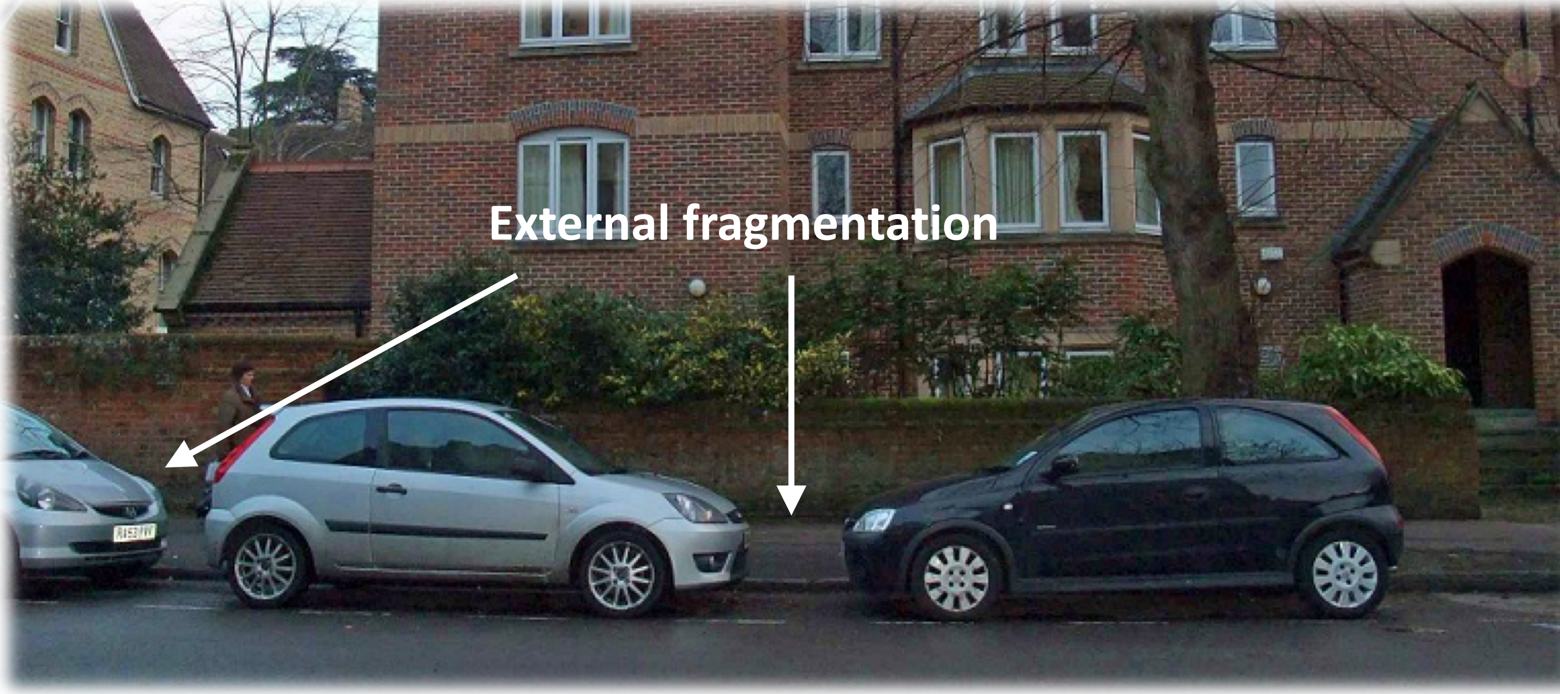
Issues: External Fragmentation

- 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!



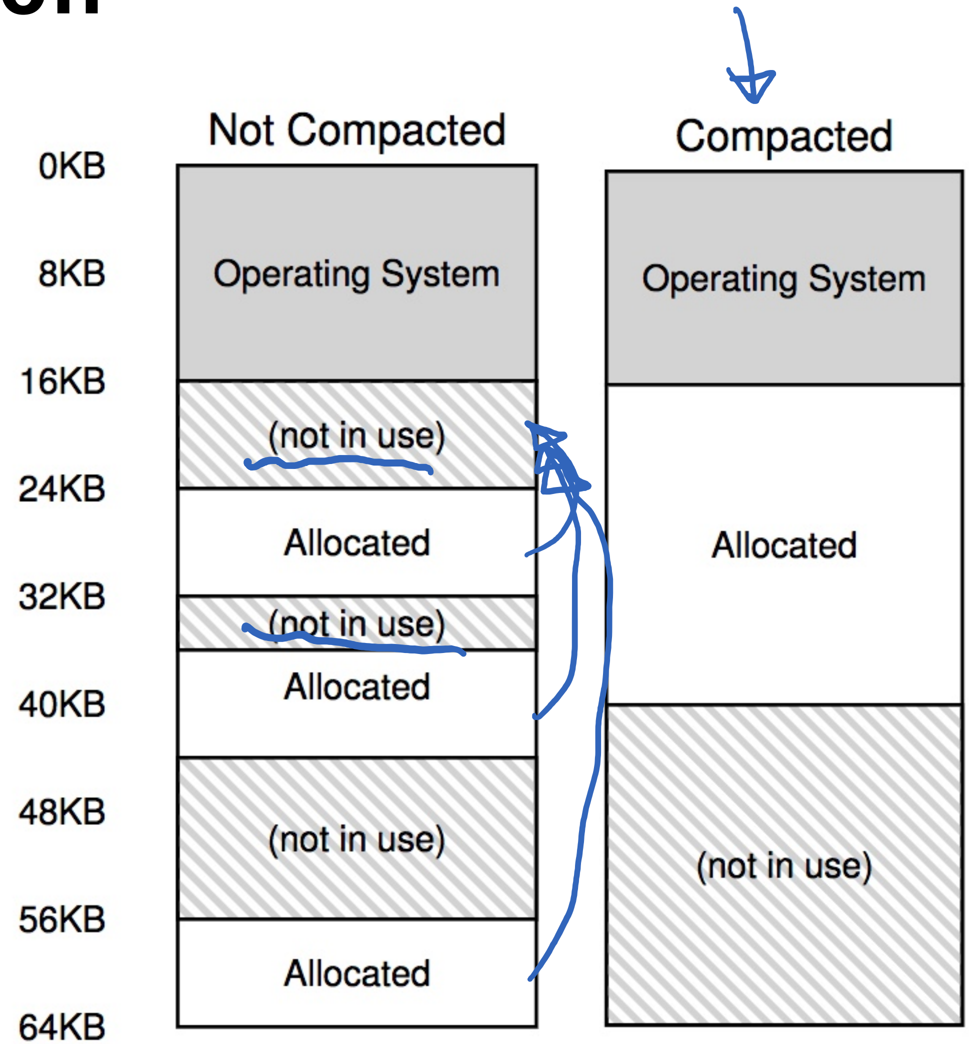

External fragmentation



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

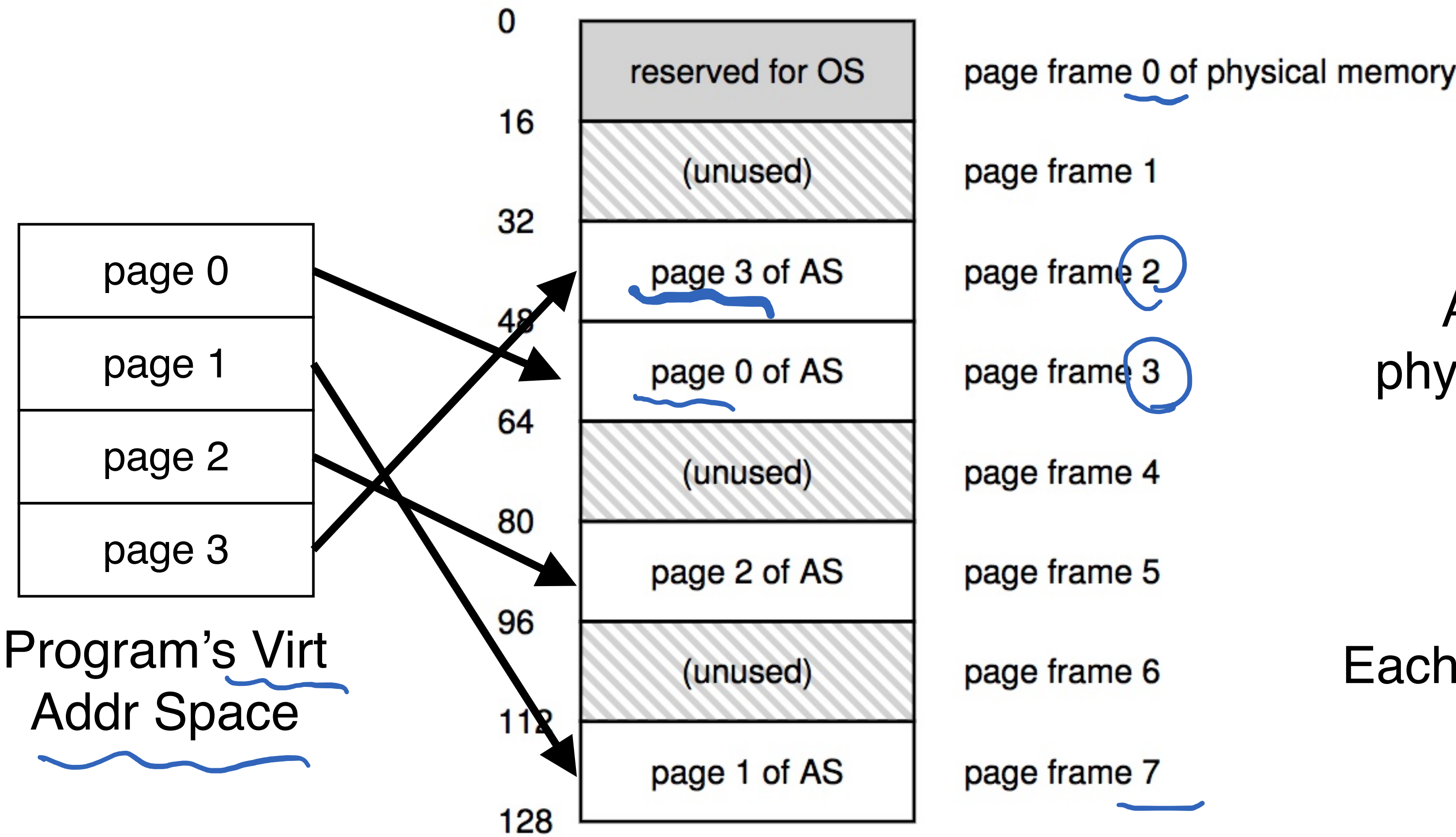
- Motivation: Segmentation is too coarse-grained
  - Either waste space (external fragmentation) or
  - copy memory often (compaction)

- 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!

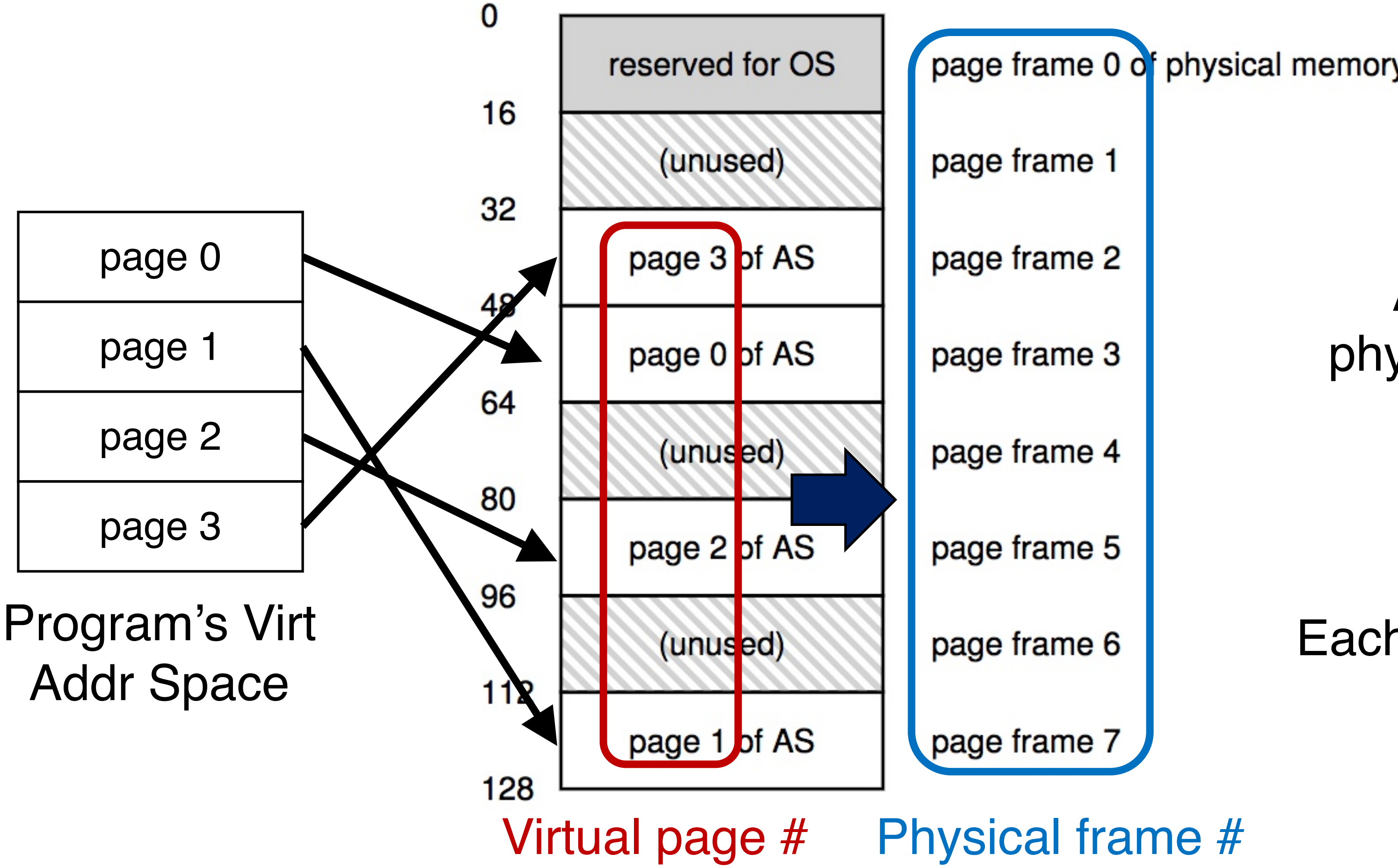
A Simple Example



An 8-frame physical memory (128B)

Each page is of 16B

A Simple Example



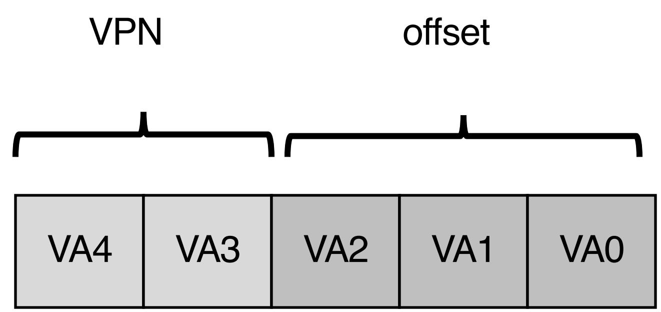
An 8-frame physical memory (128B)

Each page is of 16B

Addressing Basics

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

Virt. page number



Q: How many offset bits do we need?

A: `log(page_size)`

Address Examples

$lg(16) = 4$

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

$lg(2KB) = 10+1 = 11$

$lg(4MB) = 20+2 = 22$

Address Examples

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

Address Examples

length(VA) - length(low bits)

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

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	$2^6 = 64$
2 KB	11	20	9	
4 MB	22	32	10	
256 Bytes	8	16	8	
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}

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!

VA virt. addr.

Question: An x86_64 Linux OS with 4KB page size. How many pages can we have assuming the maximum memory limit?

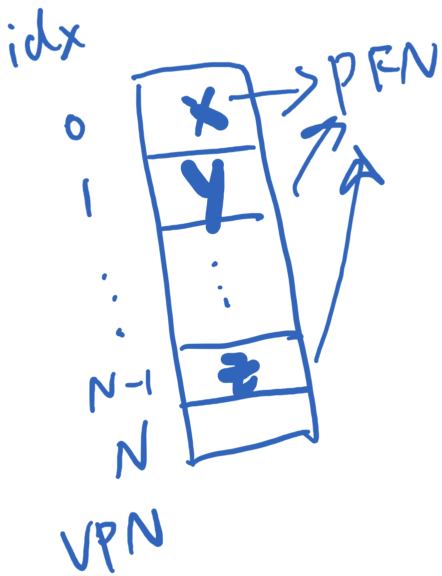
$$64 - \lg(4KB)$$
$$= 64 - (10+2)$$
$$= 64 - 12$$
$$= \underline{52}$$

2^{52} pages.

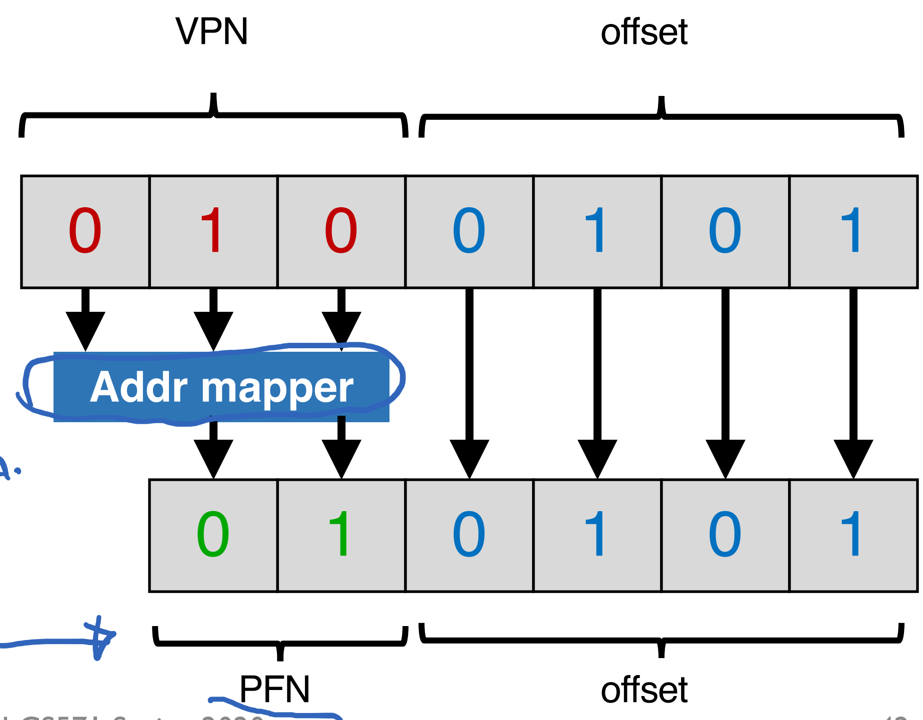
Virtual => Physical Addr Mapping

- We need a general mapping mechanism

- What data structure is good?
    - Big array



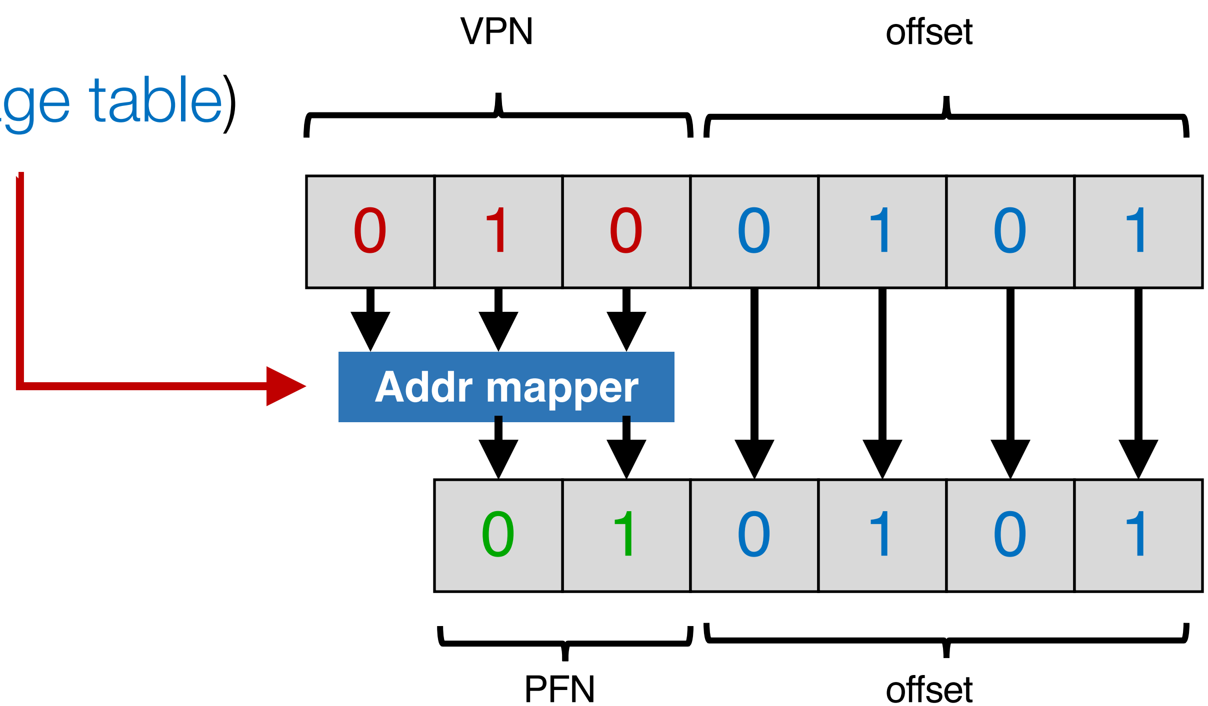
physical frame num.



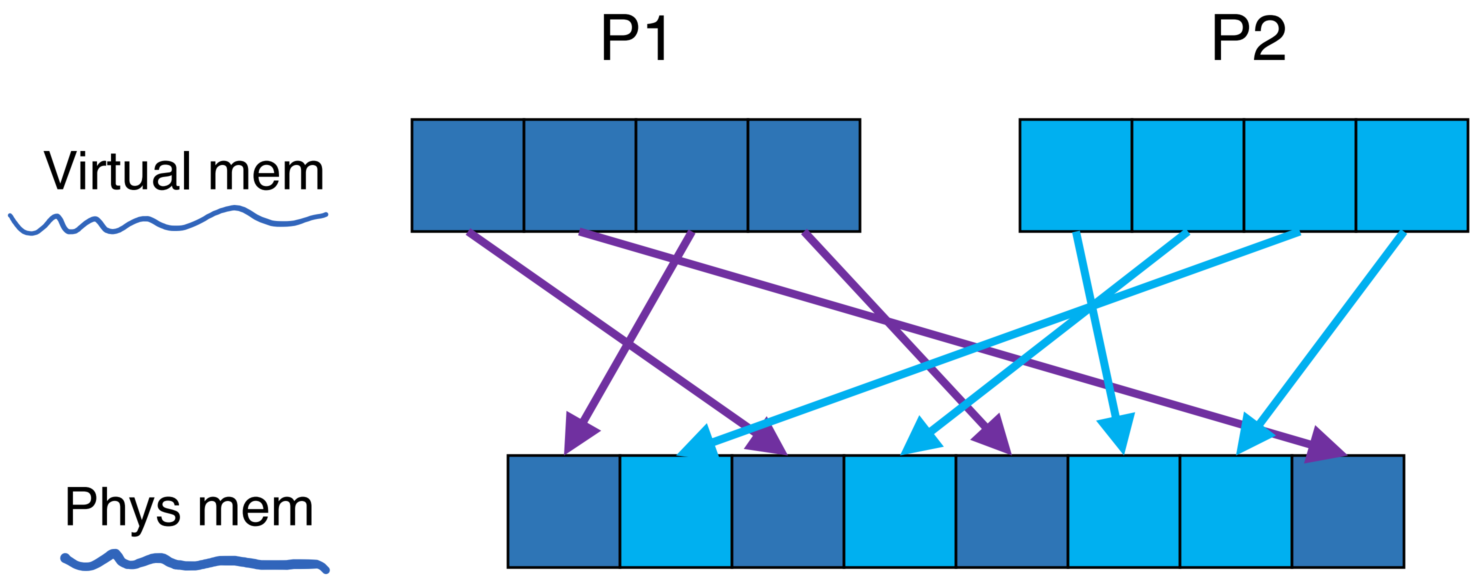
Virtual => Physical Addr Mapping

- We need a general mapping mechanism

- What data structure is good?
  - Big array
  - (aka linear page table)

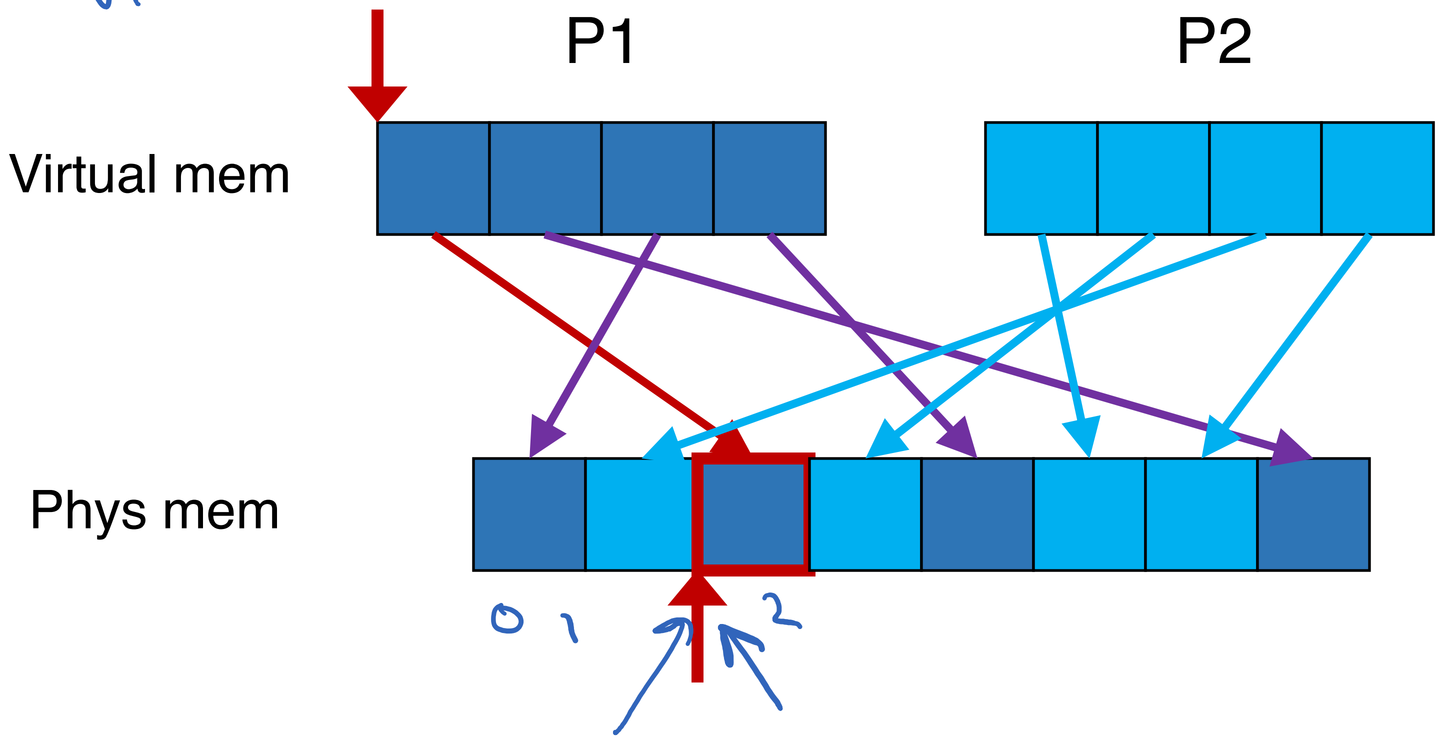


Mapping Example

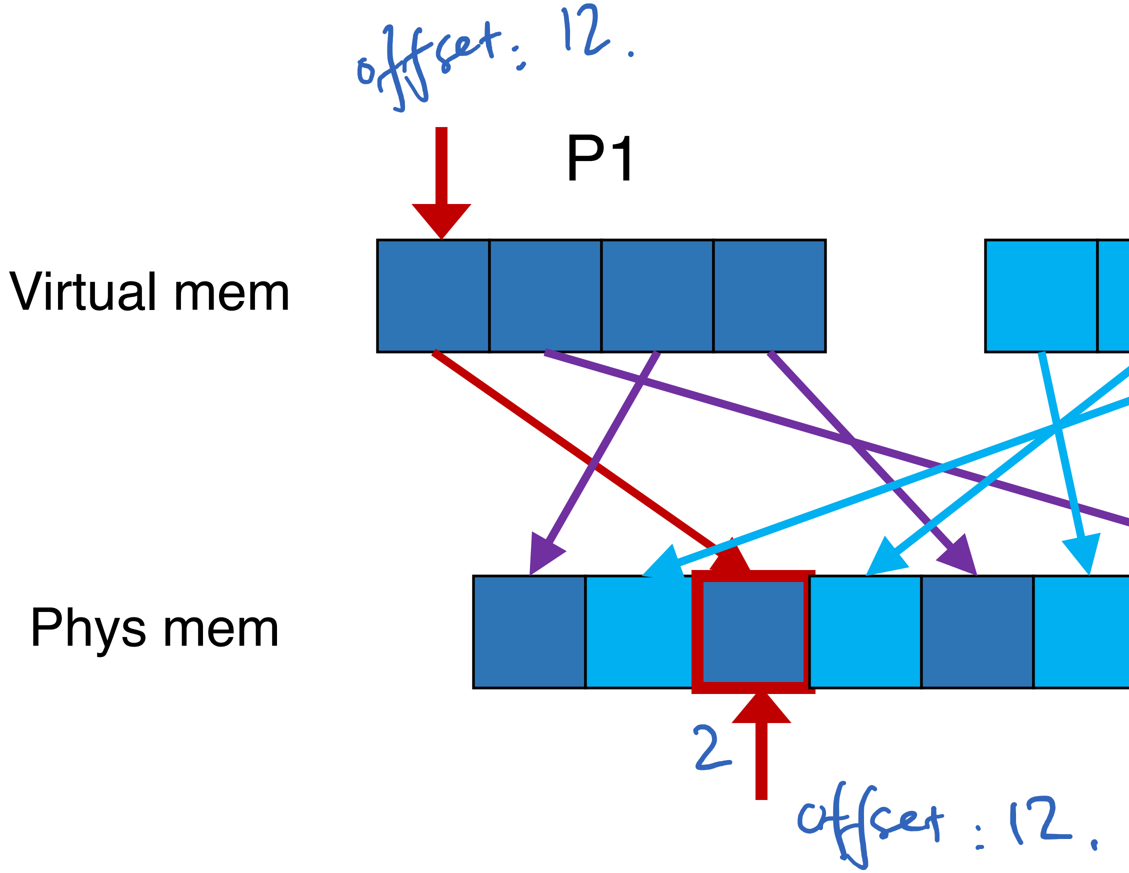


Mapping Example

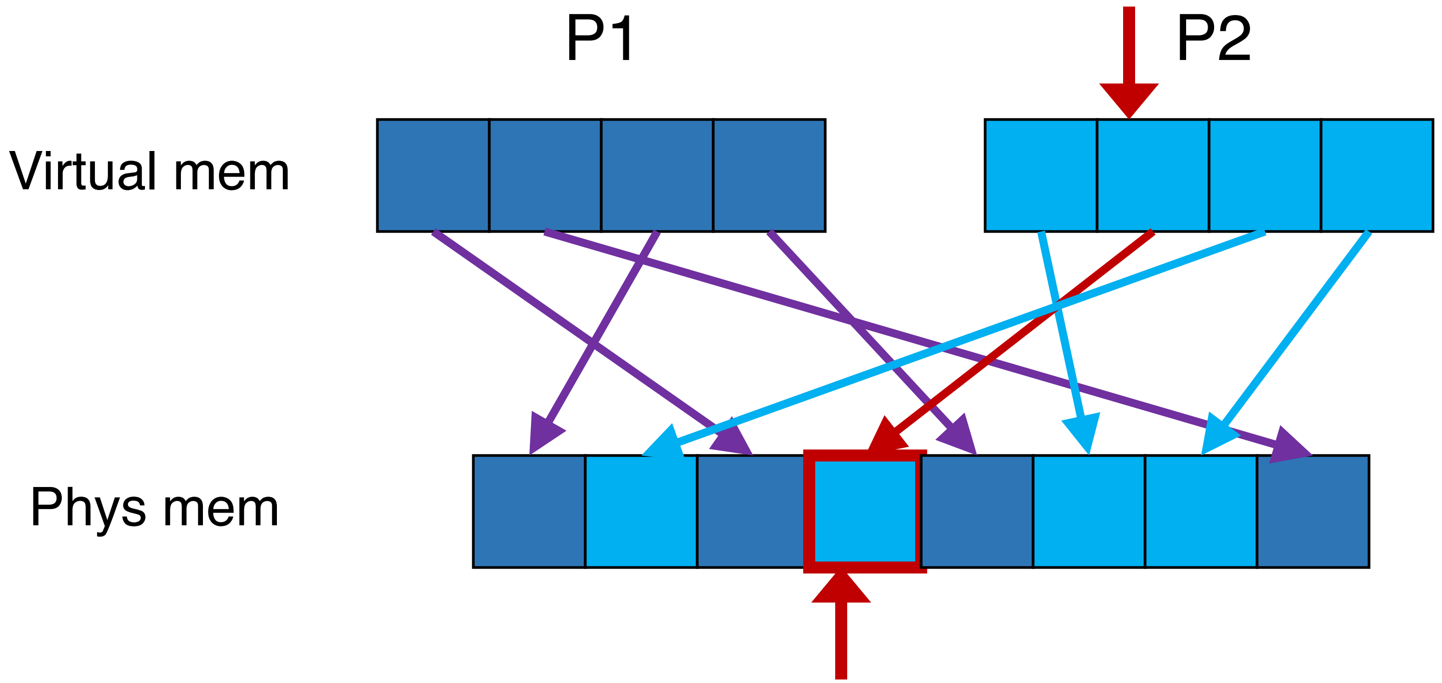
VP: 0 offset: 0



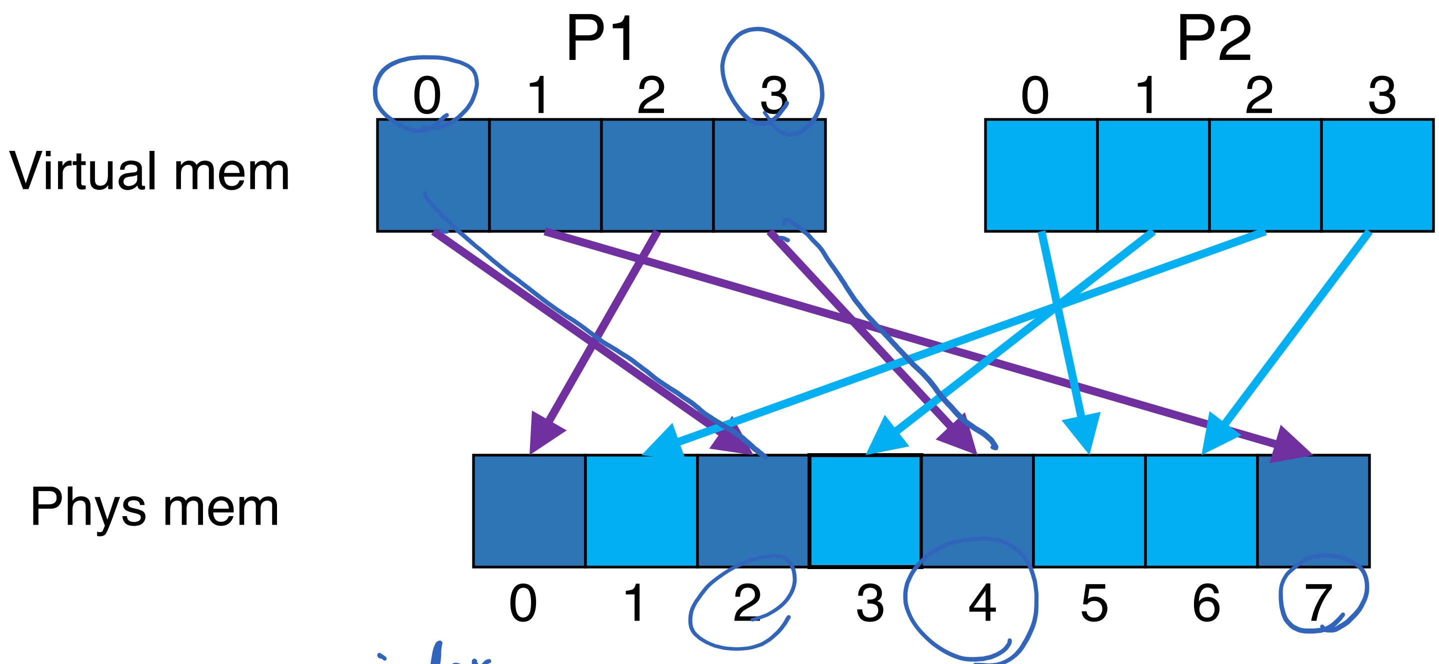
Mapping Example



Mapping Example



Mapping Example



Page tables

index ↓ → PFN.

P1

0	2
1	7
2	0
3	4

P2

0	5
1	3
2	1
3	6

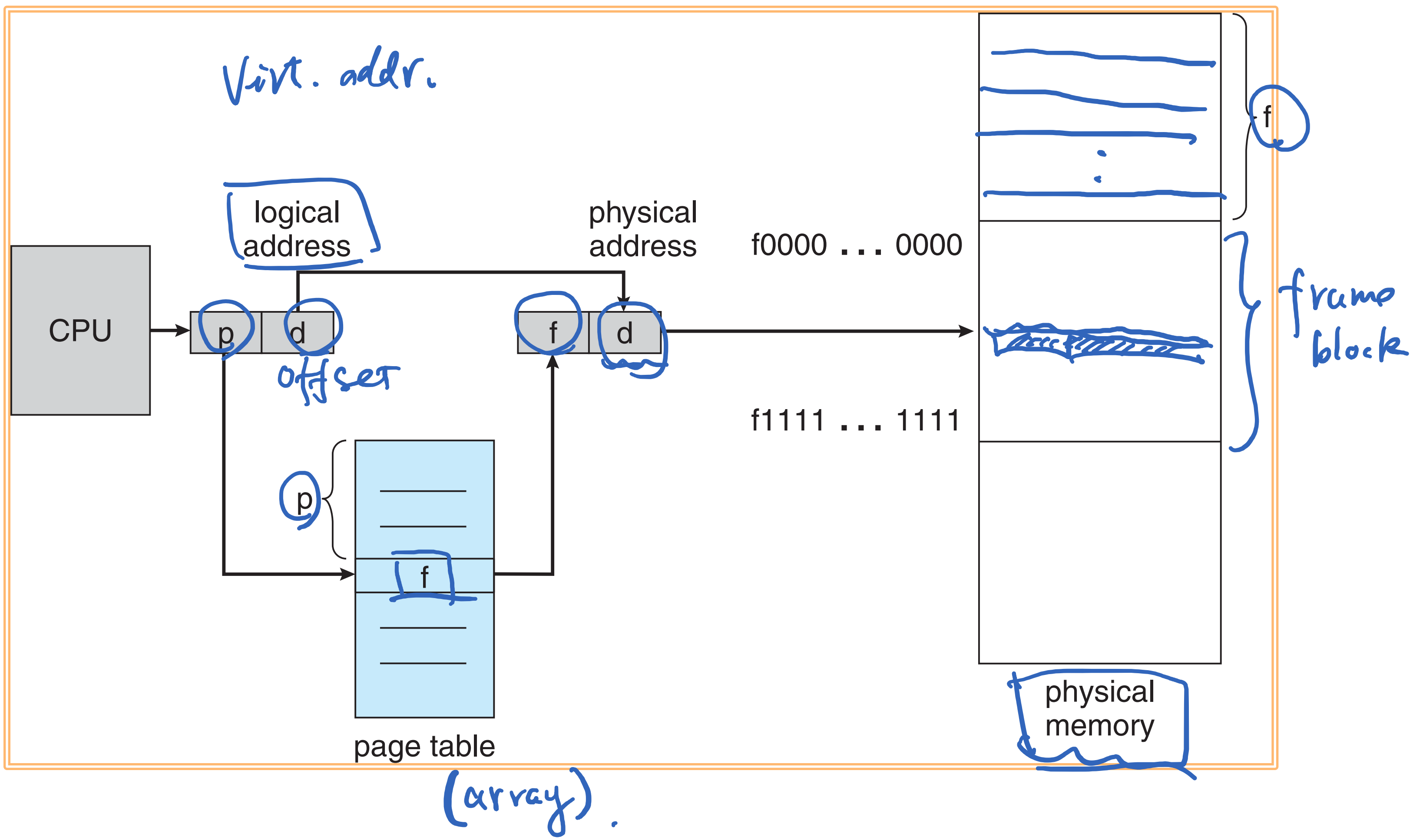
Page Table

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

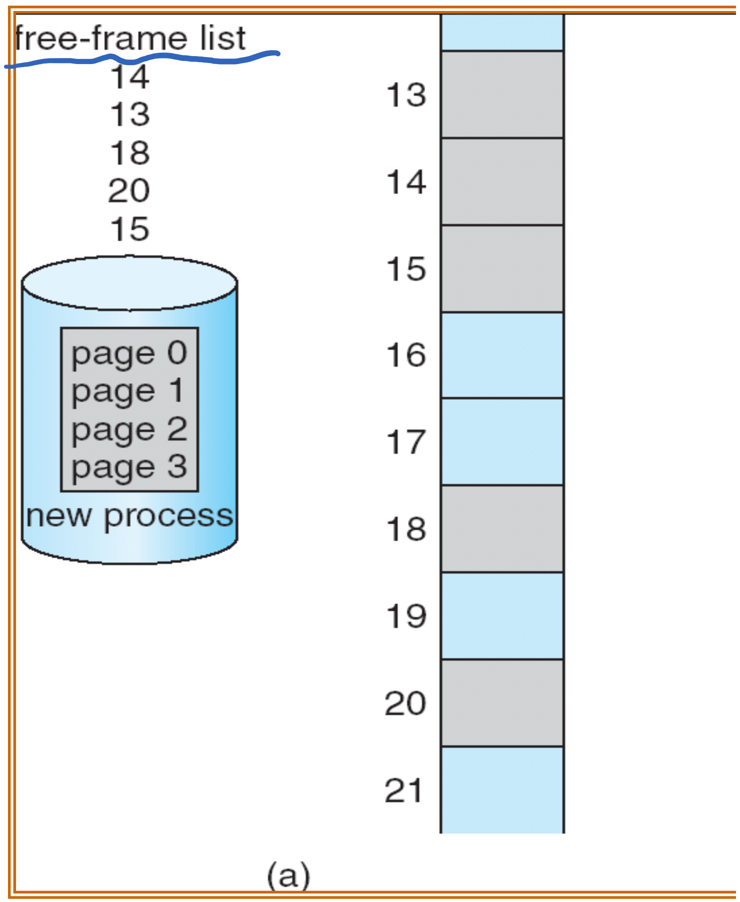
Address Translation Scheme

- Observe: The simple limit/relocation register pair mechanism is no longer sufficient
- m-bit virtual address generated by CPU is divided into: VPN
  - 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

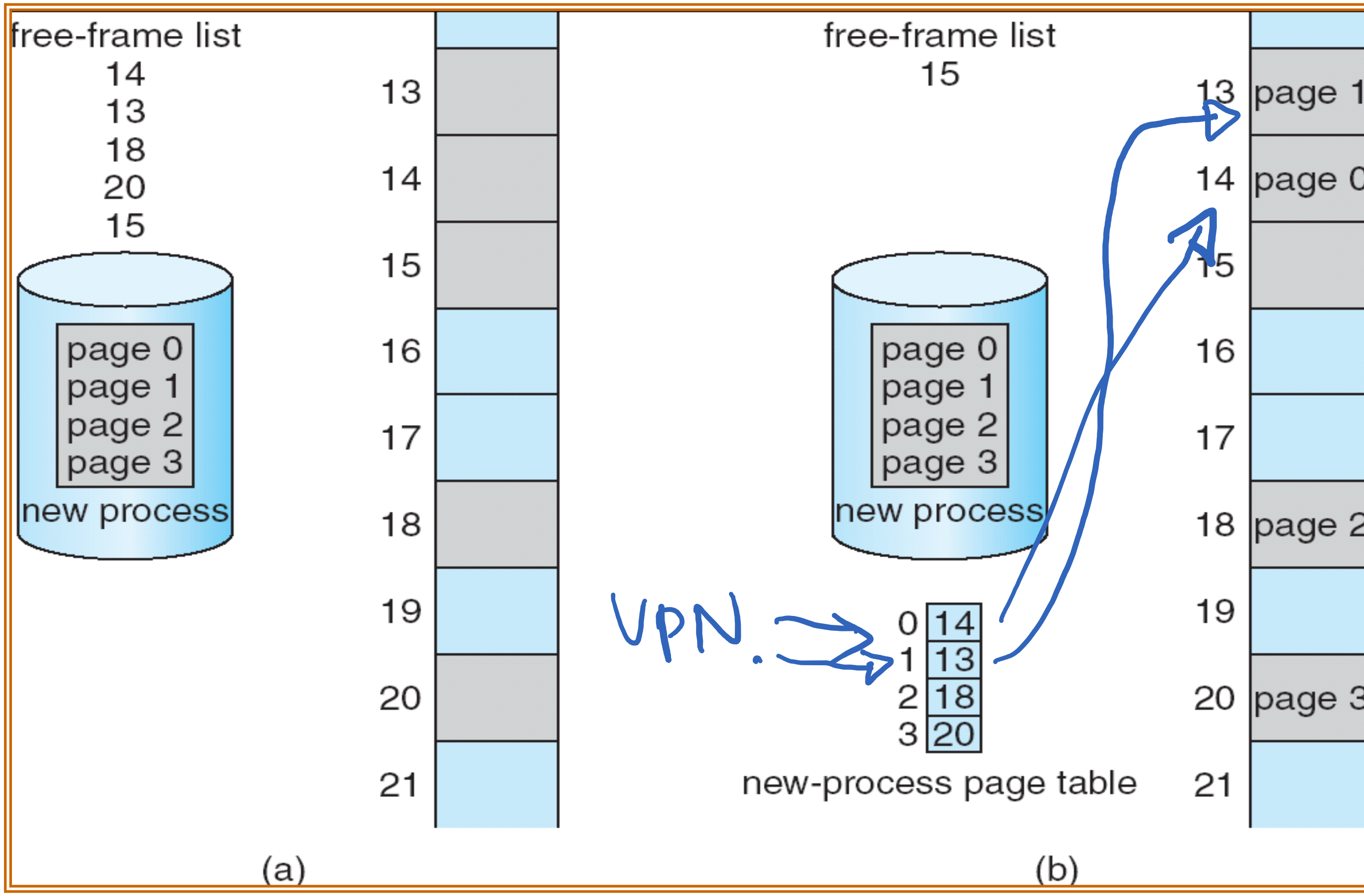


Free Frames



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)