MEMORY VIRTUALIZATION: SEGMENTATION AND PAGING

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Academic Misconduct Policy
- 0 points on P1 if have copied code from someone else
- 0 on projects P1-P8 if you repeat problem on 2 projects
- Notifying people next week; send me email before then if you have something to explain
- Project 2 Due Monday evening; Handin dirs available; tests soon
- Project 3 (Shell in Linux): Available soon; Due > 1 week after P2
  - Continue to work alone; P4 will be xv6 with partner
- Homeworks
  - One due today, two due next Tuesday, more to come…

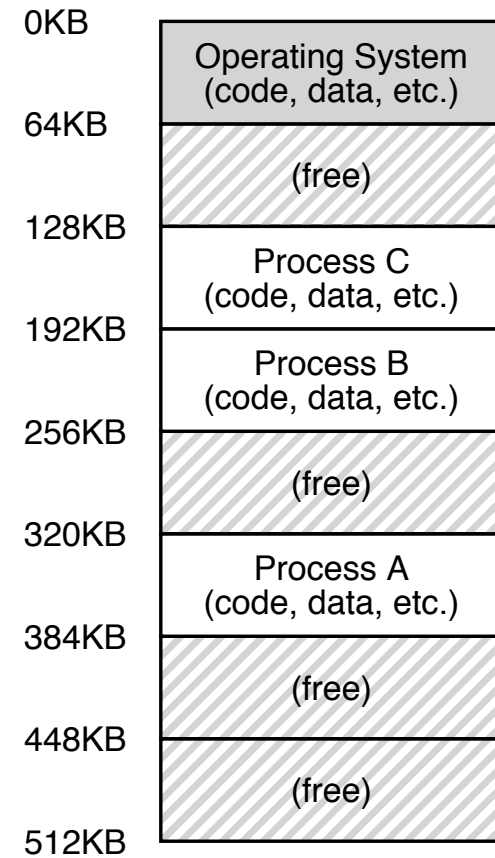# AGENDA / LEARNING OUTCOMES
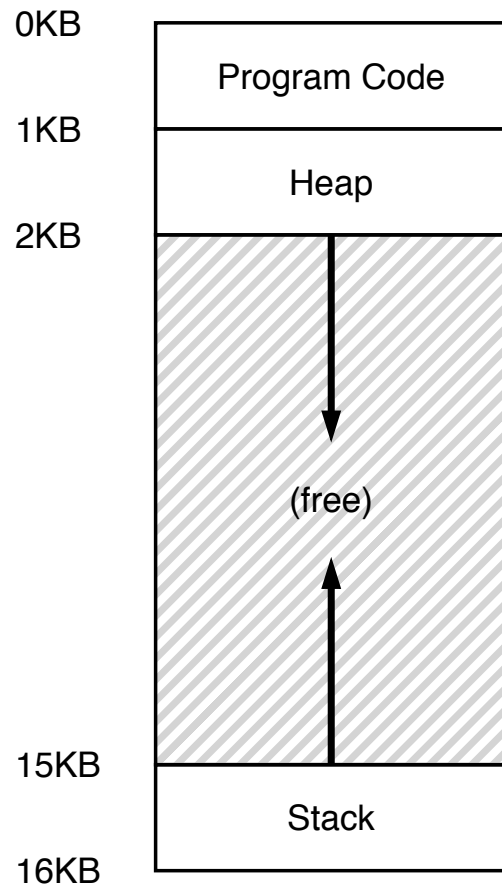
Memory virtualization

What is segmentation?

What is paging?

Pros and cons of each?

Where are page tables stored and how are they organized?

ABSTRACTION: ADDRESS SPACE

0KB	Program Code
1KB	Heap
2KB	
	(free)
15KB	
	Stack
16KB	

0KB	Operating System (code, data, etc.)
64KB	(free)
128KB	Process C (code, data, etc.)
192KB	Process B (code, data, etc.)
256KB	(free)
320KB	Process A (code, data, etc.)
384KB	(free)
448KB	(free)
512KB	

REVIEW: MEMORY ACCESS (LOGICAL)

Initial %rip = 0x10
%rbp = 0x200


```
0x10: movl  0x8(%rbp), %edi
0x13: addl  $0x3, %edi
0x19: movl  %edi, 0x8(%rbp)
```

%rbp is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch instruction at addr 0x10
Exec:
        load from addr 0x208

Fetch instruction at addr 0x13
Exec:
        no memory access

Fetch instruction at addr 0x19
Exec:
        store to addr 0x208

5 total memory accesses

REVIEW: HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

Logical addresses are "hardcoded" into process binaries

How to avoid collisions?

Possible Solutions for Mechanisms (covered previously):
1. Time Sharing
2. Static Relocation
3. Dynamic Relocation: Base Register
4. Base+Bounds

# REVIEW: 3) DYNAMIC RELOCATION

Goal: Protect processes from one another

Requires hardware support
- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference
- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses

Process runs here   OS can control MMU

```
┌──────────────┐          ┌──────────────┐          ┌──────────┐
│              │          │              │          │          │
│              │ <──────> │     MMU      │ <──────> │          │
│     CPU      │     ↑    │              │     ↑    │ Memory   │
│              │    Logical address      │   Physical address  │
│              │                         │          │          │
└──────────────┘                         └──────────────┘          └──────────┘
```

Logical address   Physical address
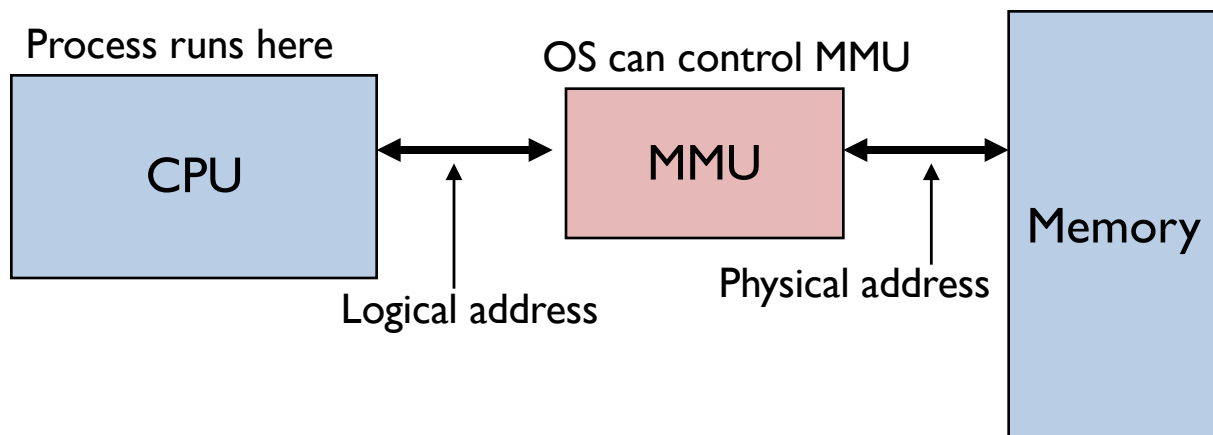
REVIEW: IMPLEMENTATION WITH BASE REG

Translation on every memory access of user process
MMU adds base register to logical address to form physical address

MMU
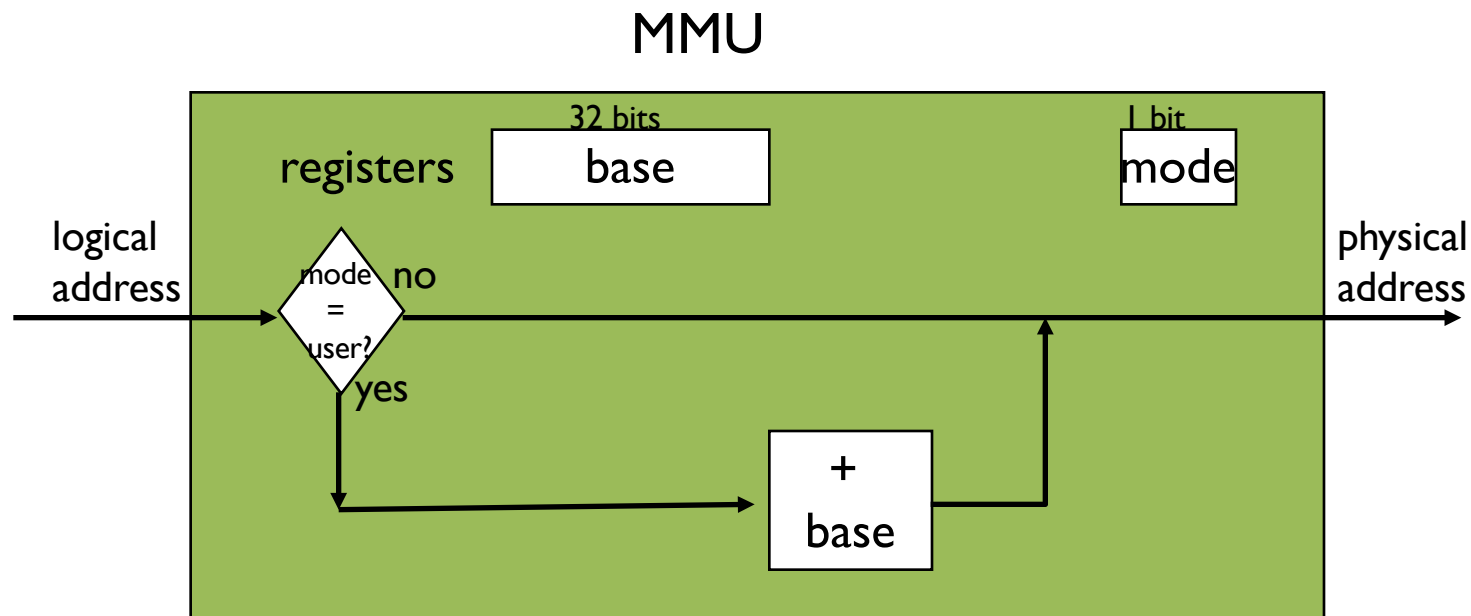
TWO-MINUTE NEIGHBOR CHAT

- Why must it be hardware that adds the base register to each logical address to form physical address?

- Does the base register contain a logical or a physical address?

- One big problem with base register approach?
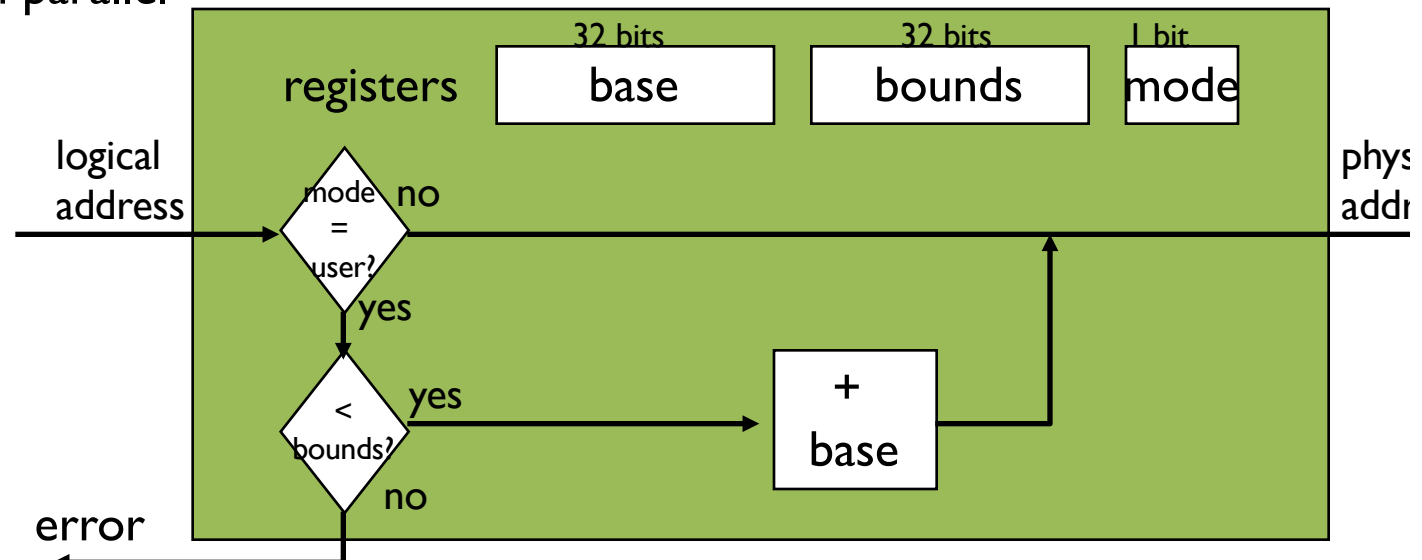
REVIEW: 4) BASE AND BOUNDS ADVANTAGES

Provides protection (both read and write) across address spaces

Supports dynamic relocation

   Can place process at different locations initially and also move address spaces

Simple, inexpensive implementation: Few registers, little logic in MMU

Fast: Add and compare in parallel

REVIEW: BASE AND BOUNDS DISADVANTAGES

Disadvantages
- Each process must be allocated contiguously in physical memory
  Must allocate memory that might not be used by process

- No partial sharing: Cannot share limited parts of address space

| 0 |
|---|
| Code |
| Heap |
| ↓ |
| ↑ |
| Stack |
| 2^n-1 |
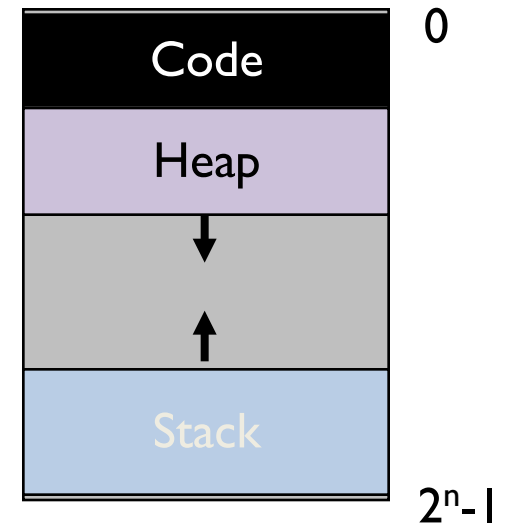
5) SEGMENTATION

Divide logical address space into logical segments
- Each segment corresponds to logical entity in address space
    (code, stack, heap)

Each segment has separate base + bounds register

Each segment can independently:
- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute bits)

0

| Code |
| Heap |
| ↓ ↑ |
| Stack |

2^n-1
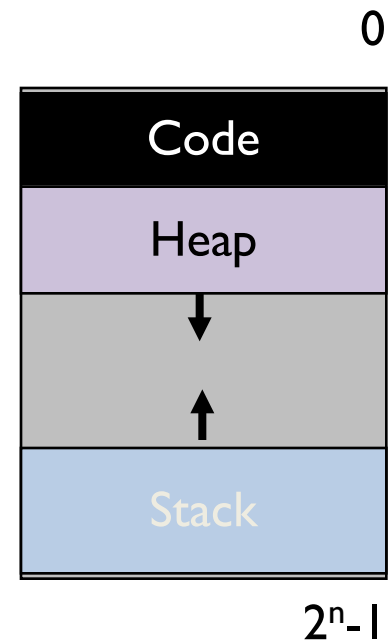
SEGMENTED ADDRESSING

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical address
    - Top bits of logical address select segment
    - Low bits of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

SEGMENTATION IMPLEMENTATION

MMU contains Segment Table (per process)
- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments;

How many bits
for segment?

How many bits
for offset?

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

remember:
1 hex digit → 4 bits

CHAT: ADDRESS TRANSLATIONS WITH SEGMENTATION

14 bit logical address, 4 segments;

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Remember:
1 hex digit→4 bits

Translate logical (in hex) to physical

0x0240:

0x1108:

0x265c:

0x3002:

VISUAL INTERPRETATION

0x00	
0x400	
	heap (seg1)
0x800	
0x1200	
0x1600	
	stack (seg2)
0x2000	
0x2400	

Virtual (hex)   Physical
load 0x2010, R1
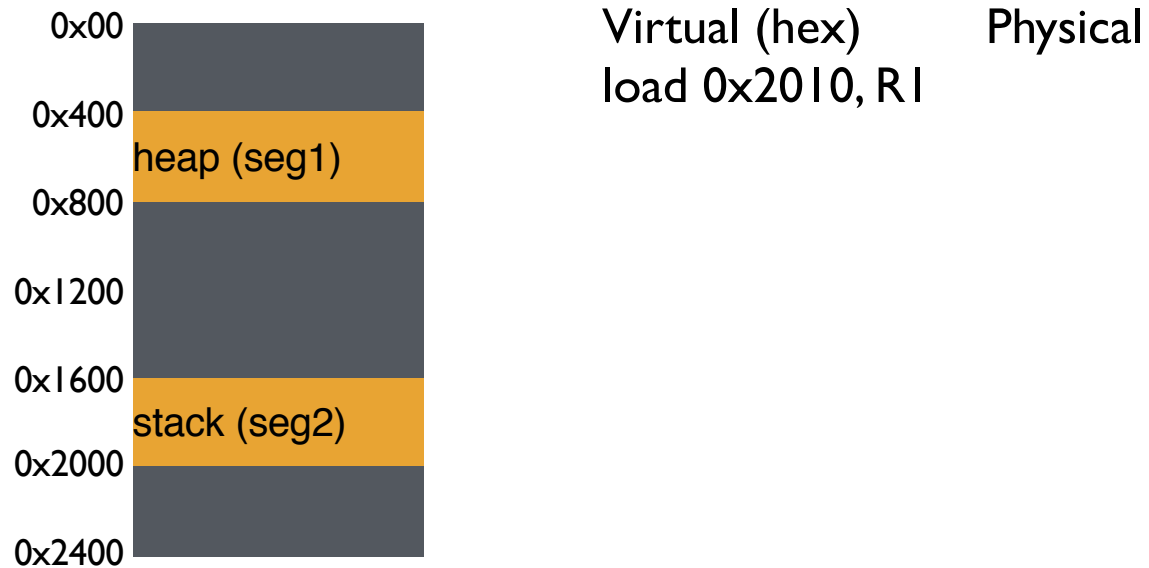
Segment numbers:
    0: code+data
    1: heap
    2: stack

VISUAL INTERPRETATION

Virtual (hex)	Physical
load 0x2010, R1	0x1600 + 0x010 = 0x1610

0x00

0x400

heap (seg1)

0x800

0x1200

0x1600

stack (seg2)

0x2000

0x2400

Segment numbers:
   0: code+data
   1: heap
   2: stack

VISUAL INTERPRETATION

0x00	
0x400	
0x800	heap (seg1)
0x1200	
0x1600	
0x2000	stack (seg2)
0x2400	

Virtual   Physical
load 0x2010, R1   0x1600 + 0x010 = 0x1610

load 0x1010, R1

load 0x1100, R1

Segment numbers:
    0: code+data
    1: heap
    2: stack

PHYSICAL MEMORY ACCESSES

```
0x0010: movl  0x1100, %edi
0x0013: addl  $0x3, %edi
0x0019: movl  %edi, 0x1100
```

%rip: 0x0010

Seg	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

1. Fetch instruction at logical addr 0x0010

   Physical addr: _____

2. Exec, load from logical addr 0x1100

   Physical addr: _____

3. Fetch instruction at logical addr 0x0013

   Physical addr: _____

4.   Exec, no load

5.   Fetch instruction at logical addr 0x0019

   Physical addr: _____

6. Exec, store to logical addr 0x1100

   Physical addr: _____

HOW DO STACKS GROW ?

0KB	Program Code
1KB	Heap
2KB	(free)
15KB	Stack
16KB	
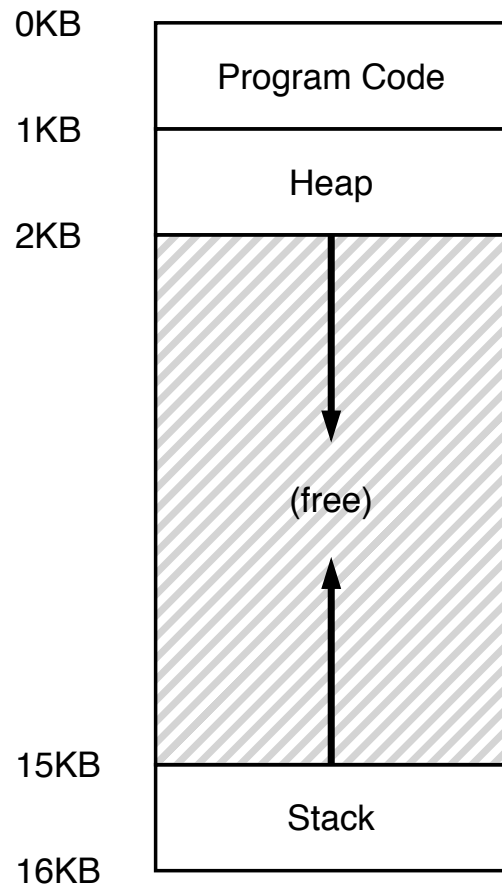
Stack goes 16K → 12K, in physical memory is 28K → 24K

Segment base is at 28K

Virtual address 0x3C00 = 15K

   → top 2 bits (0x3) segment ref, offset is 0xC00 = 3K

How do we make CPU translate that ?

Negative offset = subtract max segment from offset

                     = 3K – 4K = -1K

Add to base   = 28K – 1K = 27K

ADVANTAGES OF SEGMENTATION

Enables sparse allocation of address space

Stack and heap can grow independently

- Heap: If no data on free list, dynamic memory allocator requests more from OS
    (e.g., UNIX: malloc library makes sbrk() system call)
- Stack: OS recognizes reference near outside legal segment, extends stack implicitly

Different protection for different segments

- Enables sharing of selected segments
- Read-only status for code

Supports dynamic relocation of each segment

0

| Code |
| Heap |
| ↓ |
| ↑ |
| Stack |

2^n-1
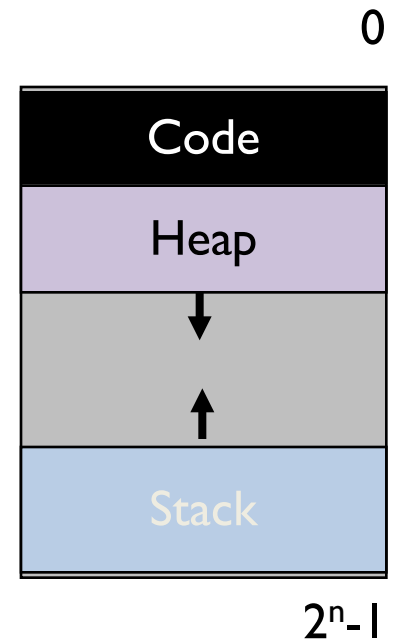
DISADVANTAGES OF SEGMENTATION

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation

Not Compacted

0KB	
	Operating System
8KB	
16KB	
	(not in use)
24KB	
	Allocated
32KB	
	(not in use)
	Allocated
40KB	
48KB	(not in use)
56KB	
	Allocated
64KB	

FRAGMENTATION

Not Compacted

0KB	
8KB	Operating System
16KB	
24KB	(not in use)
	Allocated
32KB	(not in use)
40KB	Allocated
48KB	(not in use)
56KB	
64KB	Allocated

Compacted

0KB	
8KB	Operating System
16KB	
24KB	Allocated
32KB	
40KB	
48KB	(not in use)
56KB	
64KB	
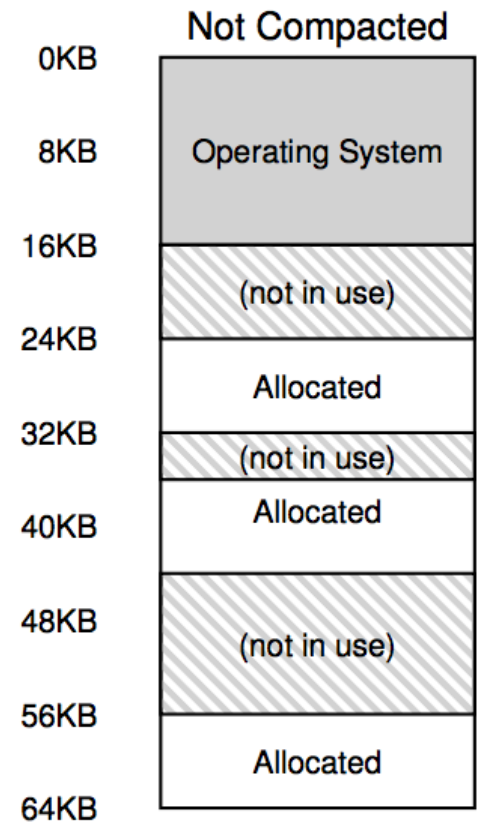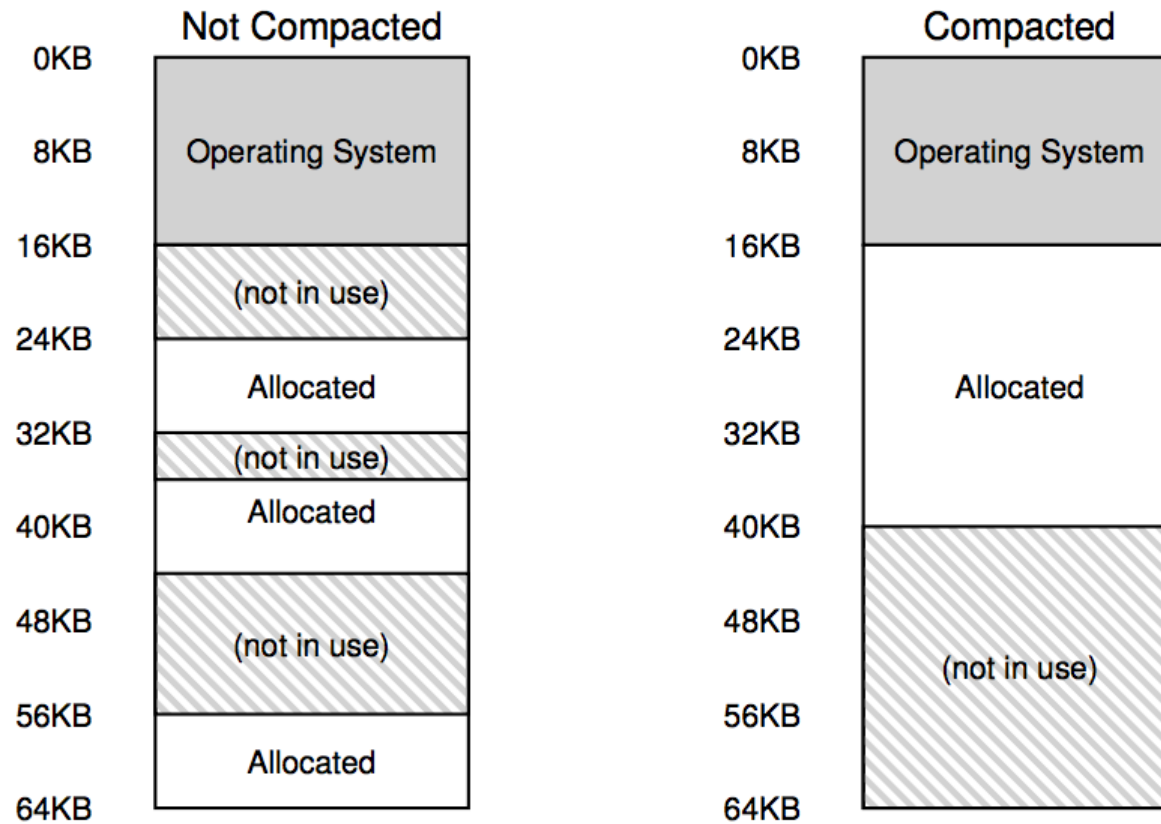
1-MINUTE CHAT: MATCH DESCRIPTION

Description

1. one process uses RAM at a time
2. rewrite code and addresses before running
3. add per-process starting location to virt addr to obtain phys addr
4. dynamic approach that verifies address is in valid range
5. several base+bound pairs per process

Name of approach

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing

PAGING

FRAGMENTATION

Not Compacted

Address	Region
0KB	Operating System
8KB	Operating System
16KB	(not in use)
24KB	Allocated
32KB	(not in use)
40KB	Allocated
48KB	(not in use)
56KB	Allocated
64KB	
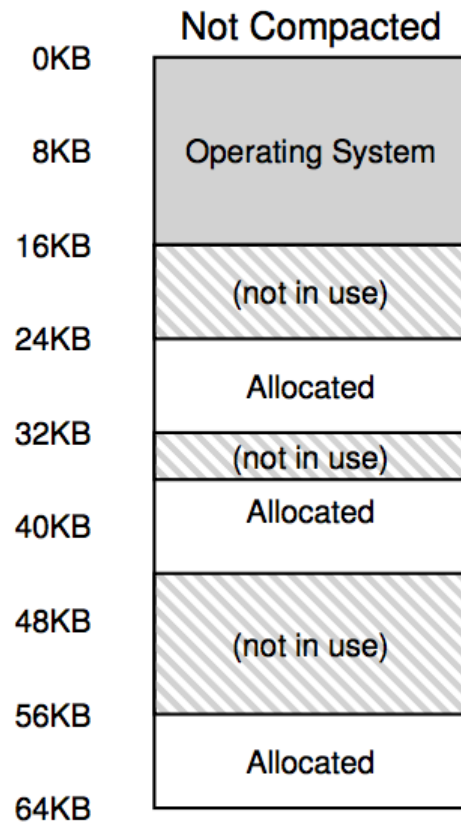
Definition: Free memory that can't be usefully allocated

Types of fragmentation
    External: Visible to allocator (e.g., OS)
    Internal: Visible to requester

Internal

useful
free
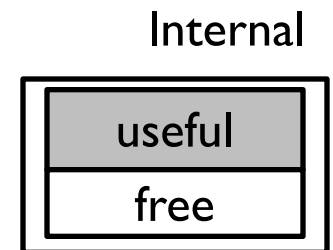
PAGING

Goal: Eliminate requirement that address space is contiguous
 Eliminate external fragmentation
 Grow segments as needed

Idea:
Divide address spaces and physical
memory into fixed-sized pages

Size: 2^n, Example: 4KB

Process 1 Process 2 Process 3

Logical View

Physical View
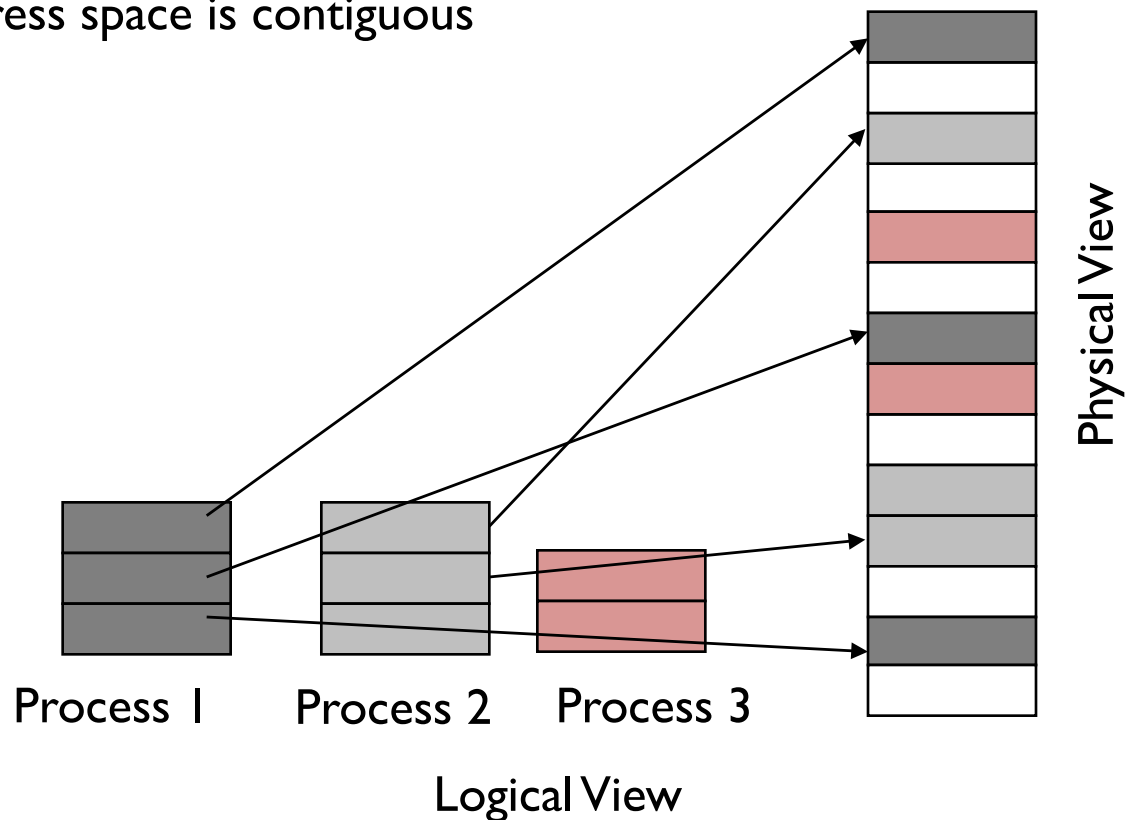
TRANSLATION OF PAGE ADDRESSES

How to translate logical address to physical address?
- High-order bits of address designate page number
- Low-order bits of address designate offset within page
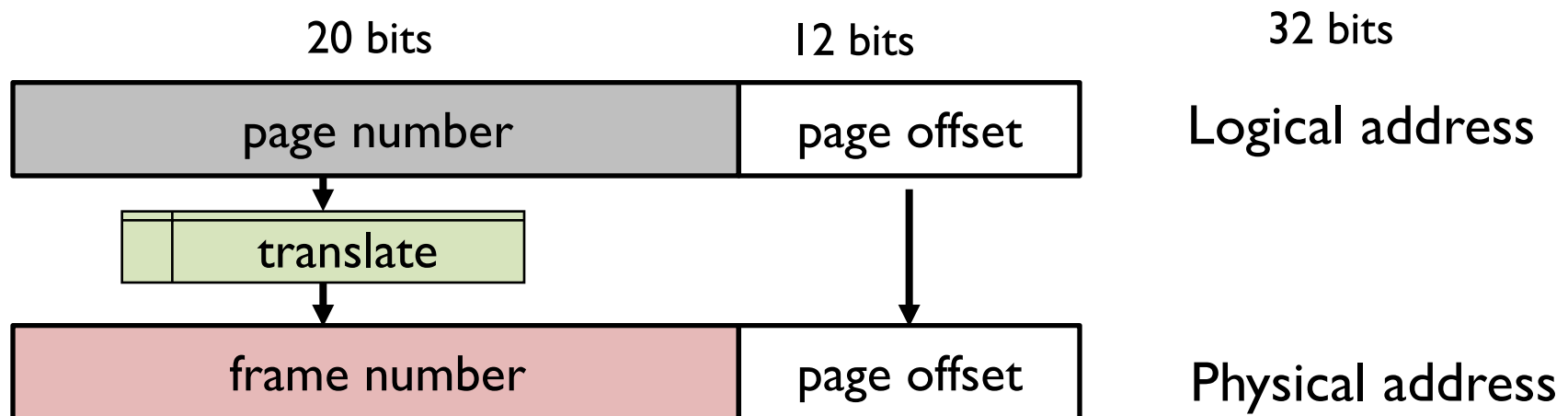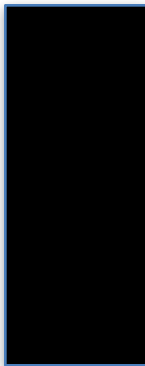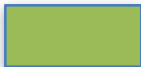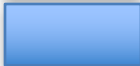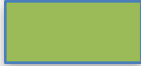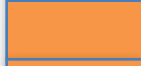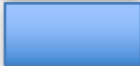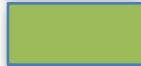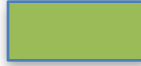
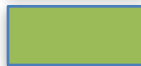No addition needed (unlike segmentation); just append bits correctly…

ADDRESS FORMAT

Given known page size, how many bits are needed in address to specify offset in page?

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Given number of bits for vpn, how many virtual pages can there be in an address space?

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	# of Virt Pages
16 bytes				
1 KB				
1 MB				
512 bytes				
4 KB				

VIRTUAL → PHYSICAL PAGE MAPPING

Number of bits in
virtual address

need not equal

number of bits in
physical address

VPN   offset

| 0 | I | 0 | I | 0 | I |

Addr Mapper

| I | 0 | I | I | 0 | I | 0 | I |

PPN   offset
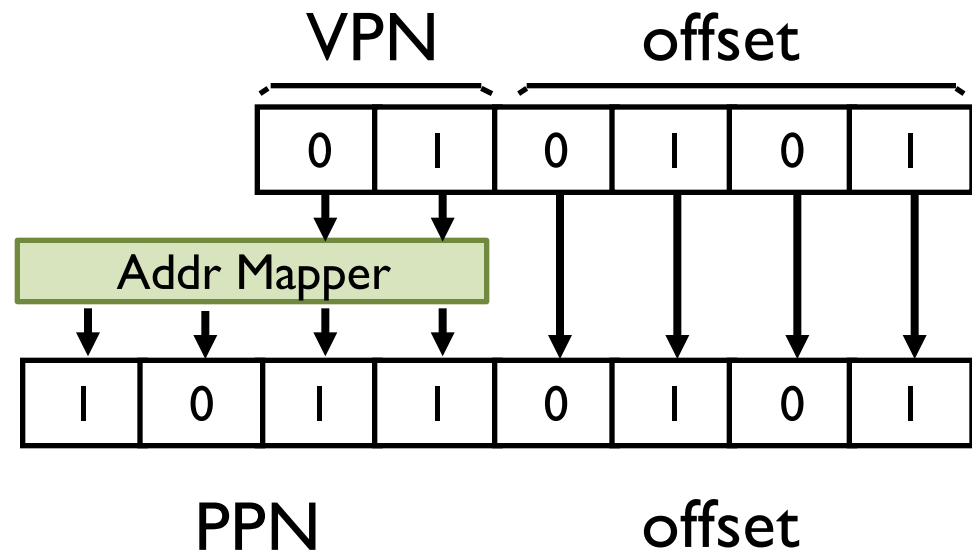
How should OS translate VPN to PPN?

For paging, OS needs general mapping mechanism
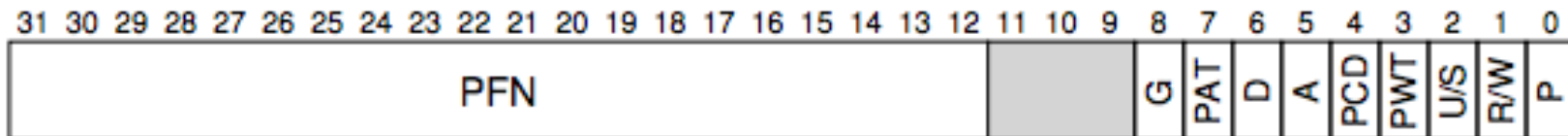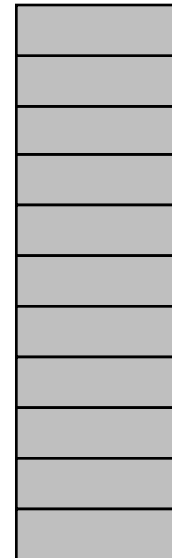What data structure is good?

LINEAR PAGETABLES

VPN

0

2^n

What is a good data structure ?

Simple solution: Linear page table aka *array*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																							G	PAT	D	A	PCD	PWT	U/S	R/W	P

PER-PROCESS PAGETABLE
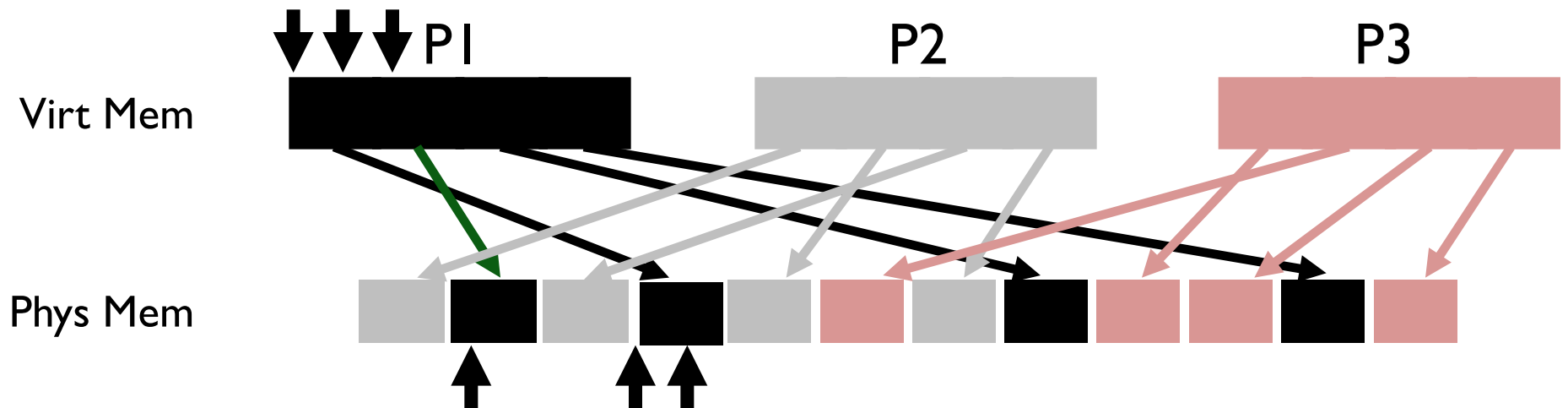
FILL IN PAGETABLE
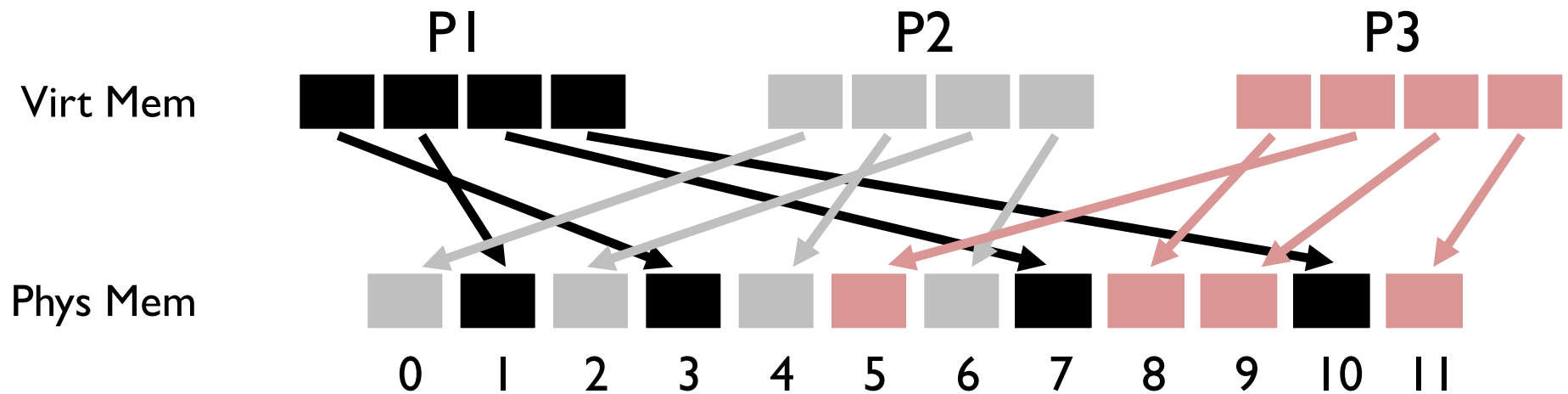
NEIGHBOR CHAT: HOW BIG IS A PAGETABLE?

How big is a typical page table?
- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Key: Figure out how many PTEs (Page table entries)

HOW BIG IS A PAGETABLE?

How big is a typical page table?
    - assume **32-bit** address space
    - assume 4 KB pages
    - assume 4 byte page table entries (PTEs)

WHERE ARE PAGETABLES STORED?

Implication: Store each page table in memory
    Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?
    Change contents of page table base register to newly scheduled process
    Save old page table base register in PCB of descheduled process

OTHER PAGETABLE INFO

What other info is in pagetable entries besides translation?

- – valid bit

- – protection bits

- – present bit (needed later)

- – reference bit (needed later)

- – dirty bit (needed later)

Pagetable entries are just bits stored in memory

- – Agreement between hw and OS about interpretation

MEMORY ACCESSES WITH PAGING

`0x0010: movl  0x1100, %edi`

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view
of page table

2
0
80
99

Pagetables are slow! Double memory references

Fetch instruction at logical addr 0x0010

- Access page table to get ppn for vpn 0

- Mem ref 1: 0x5000

- Learn vpn 0 is at ppn 2

- Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100

- Access page table to get ppn for vpn 1

- Mem ref 3: 0x5004

- Learn vpn 1 is at ppn 0

- Movl from 0x0100 into reg (Mem ref 4)

ADVANTAGES OF PAGING

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add "present" bit to PTE

DISADVANTAGES OF PAGING

Internal fragmentation: Page size may not match size needed by process
- – Wasted memory grows with larger pages
- – **Tension? Why not make pages very small?**

Additional memory reference to page table → Very inefficient
- – Page table must be stored in memory
- – MMU stores only base address of page table
- – Solution: Add TLBs (future lecture)

Storage for page tables may be substantial
- – Linear page table: Requires PTE for all pages in address space
- – Entry needed even if page not allocated
- – Page tables must be allocated contiguously in memory
- – Fix with segmentation and paging page tables (next)

Code
Heap
↓ ↑
Stack